Отзывы о книге

Резник, Крейн и Боуэн исследовали самую суть платформы веб-служб, разработанной корпорацией Microsoft. Неважно, читаете вы о WCF в первый или в пятьдесят первый раз, все равно из этой книги вы узнаете что-то новое для себя.

– Николас Аллен, менеджер по разработке ПО,
отделение веб-служб, корпорация Microsoft

Нас, разработчиков, часто просят выступить в роли экспертов во многих областях. Рано или поздно приходит время для разработки распределенных систем и механизмов обмена сообщениями на новой платформе Microsoft .NET 3.x, и тут неизбежно сталкиваешься с очередным монстром, который называется Windows Communication Foundation (WCF). Когда день наступит, эта книга должна лежать у вас на столе.

– Рон Ландерс, старший технический консультант,
компания IT Professionals, Inc.

Проектирование и написание распределенных приложений всегда было одной из самых сложных задач, стоящих перед архитекторами и разработчиками на платформе .NET. На какой технологии остановиться? Выбор так широк, а времени на кодирование так мало. Windows Communication Foundation (WCF) решает эту проблему, предлагая единую унифицированную платформу для построения распределенных приложений в .NET. Как и любой инструмент для создания распределенных систем, WCF поддерживает много разных возможностей. Подход, принятый в этой книге, позволит вам без особого труда освоить их спектр и получить на свои вопросы ответы, подкрепленные примерами из реальной практики. Начав с основ WCF, автор показывает, как применять эту технологию сегодня. Книгу обязательно должны прочитать все архитекторы и разработчики распределенных приложений любого типа.

– Том Роббинс, директор по управлению продуктами
на платформе .NET, корпорация Microsoft

Книга «Основы Windows Communication Foundation (WCF)» – это всестороннее и вместе всем легкое для усвоения описание технологии. Она безусловно будет полезна как знающим читателям, так и тем, кто только начинает знакомиться с WCF.

– Вилли-Петер Шауб, технический специалист,
компания Barone, Budge, and Dominick Ltd., Microsoft MVP

Очевидно, что авторы посвятили много лет разработке распределенных приложений, что и позволило им выявить и изложить самую суть WCF. В результате на свет появилась книга, содержащая массу практически полезной информации; она сэкономит ваше время и подскажет, как приступить к своему WCF-проекту. Одна только глава о диагностике избавит вас от многочасовых бдений, проведенных за отладкой системы. Настоятельно рекомендую.

– Ясер Шохуд, технический директор,
Microsoft Technology Center, Dallas

Стив Резник, Ричард Крейн, Крис Боуэн

WINDOWS COMMUNICATION FOUNDATION

for .NET Framework 3.5

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Стив Резник, Ричард Крейн, Крис Боуэн

ОСНОВЫ WINDOWS COMMUNICATION FOUNDATION

для .NET Framework 3.5

ДМК ПРЕСС
издательство

Москва, 2008

УДК 004.4
ББК 32.973.26-018.2
Р34

С34 **Стив Резник, Ричард Крейн, Крис Боуэн**

Основы Windows Communication Foundation для .NET Framework 3.5: Пер. с англ. Слинкина А. А. – М.: ДМК Пресс, 2008. – 480 с.: ил.

ISBN 978-5-94074-465-8

Технология Windows Communication Foundation (WCF) – самый простой способ создания и потребления веб-служб на платформе Microsoft. В версии .NET 3.5 WCF была существенно переработана, а в Visual Studio 2008 включены мощные инструменты для работы с ней. Из этой книги вы узнаете, как выжать максимум возможного из WCF с помощью .NET 3.5 и Visual Studio 2008.

Основываясь на обширном опыте работы с пользователями, раньше других приступившими к изучению WCF, три сотрудника Microsoft систематически рассматривают темы, вызывающие наибольшее количество вопросов у разработчиков. Авторы дают практические рекомендации, рассказывают о проверенных приемах, дают множество полезных советов по ренешию конкретных задач. В книге вы найдете подробные объяснения, подходы к «болевым точкам», свойственным разработке с помощью WCF, и богатый набор примеров повторно используемого кода.

УДК 004.4
ББК 32.973.26-018.2

ISBN 978-0-321-44006-8 (англ.)
ISBN 978-5-94074-465-8 (рус.)

Моим родителям, которые указали мне правильное направление, Замани – за то, что она сделала путь таким приятным, и Ною и Ханне, в которых я вижу будущее.

– Стив

Посвящаю моей любимой жене Ники, сыну Мэттью и дочери Шарлотте. Спасибо за вашу поддержку и понимание. Я всех вас очень люблю и надеюсь, что мы сможем проводить больше времени вместе.

– Рич

Спасибо моей жене Джессике и дочерям Деборе и Рэчел за любовь и понимание, ведь я снова посвящал долгие вечера и выходные работе. Получившейся книгой можно гордиться, и теперь я смогу вернуть семье украденное у нее время.

– Крис

Содержание

Предисловие

Я пишу это предисловие в декабре 2007 года, спустя чуть больше года после выхода в свет первой редакции Windows Communication Foundation в составе каркаса .NET Framework 3.0 и чуть меньше месяца с момента внесения существенных изменений в редакцию, поставляемую вместе с .NET Framework 3.5. Сказать, что в этих двух редакциях содержится много материала для изучения, – все равно что ничего не сказать.

Одной из целей WCF была унификация технологий создания всех видов распределенных приложений на платформе Microsoft. Мы стремились выработать базовый набор концепций – простых и доступных, но вместе с тем достаточно выразительных для моделирования семантики всех технологий, которые мы собирались заменить. У предшествующих технологий (ASMX, Remoting, COM+, MSMQ и WSE) были как сильные стороны, так и значительные ограничения, и мы поставили перед собой задачу не отбрасывать идеи, хорошо зарекомендовавшие себя в прошлом, и извлечь уроки из ошибок. Если мы достигли своей цели, то разработчики смогут писать разнообразные распределенные приложения, не утруждая себя изучением многочисленных (и часто совершенно не стыкующихся между собой) моделей программирования.

Чтобы наше видение унифицированной технологии разработки не разбилось о практические трудности, нам была нужна очень гибкая архитектура исполняющей среды, которая отвечала бы богатству модели программирования. Необходимо было выявить основные места, в которых возможны изменения, и изолировать их в виде обобщенных механизмов расширяемости, чтобы не ограничивать без нужды возможности новой платформы. Если предлагаемое по умолчанию поведение не отвечает потребностям конкретного приложения или мы не предусмотрели функцию, требующуюся в конкретном сценарии, то в исполняющей среде должно быть естественное место, в которое внешний разработчик мог бы подключить свой код и тем самым решить задачу.

Для меня самым захватывающим аспектом WCF стало широчайшее разнообразие ситуаций, в которых можно применить эту технологию. И самым убедительным доказательством этого факта служит набор функций, включенных в .NET 3.5. При подготовке этой редакции мы держали в уме два очень разных типа распределенных приложений. Во-первых, мы стремились интегрировать WCF с громадными возможностями технологии Windows Workflow Foundation и тем самым заложить основу для декларативной реализации распределенных бизнес-процессов, протяженных во времени. Во-вторых, мы хотели, чтобы WCF отвечала потребностям сегодняшней активно развивающейся сети Web. Эти две задачи предъявляют совершенно разные требования к исполняющей среде и к модели программирования, и тот факт, что мы сумели удовлетворить все требования за счет расширений WCF *без существенной модификации имеющейся реализации*, говорит о том, что архитектура WCF сможет поддерживать изменяющиеся потребности распределенных приложений и в отдаленной перспективе.

Теперь, по прошествии года с момента выпуска первой редакции, нам приятно видеть, что реальные клиенты многое поставили на нашу платформу. А еще приятнее слышать о том, как много они при этом выиграли с точки зрения производительности труда разработчиков, эффективности и интероперабельности. Успех нашей работы мы оцениваем, прежде всего, по успехам наших клиентов, и по этому критерию WCF действительно оказалась очень удачной платформой. В общем, я веду к тому, что время, потраченное на изучение WCF, будет потрачено не зря. И с этой точки зрения вам очень повезло с этой книгой. Рич, Крис и Стив проделали огромную работу, чтобы выделить и объяснить те базовые элементы WCF, которые вам необходимо понимать для продуктивной работы. Присущее авторам уникальное сочетание технической осведомленности, практического опыта и тесных связей с командой разработчиков позволило им написать книгу, которая по праву займет почетное место на книжной полке любого разработчика на платформе WCF. Я искренне рад, что именно эти парни рассказали техническую историю о нашем продукте. Уверен, что прочитав книгу, вы будете думать так же.

Стив Мэйн
Сиэтл, штат Вашингтон
Декабрь 2007

Вступление

Windows Communication Foundation (WCF) – это унифицированная модель программирования распределенных приложений на платформе Microsoft. Она инкорпорирует предшествующие технологии – ASMX, .NET Remoting, DCOM и MSMQ – и предоставляет расширяемый API, отвечающий разнообразным требованиям, которые возникают при создании распределенных систем. До WCF вам приходилось овладевать всеми этими технологиями, чтобы выбрать ту, которая лучше всего подходит в конкретной ситуации. WCF упрощает задачу, предлагая единообразный подход.

В современных распределенных приложениях чаще всего применяются Web-службы на основе XML. С их помощью реализуются разнообразные технические и бизнес-функции как в закрытых, так и в открытых сетях. Иногда при этом используется спецификация SOAP, иногда – нет. Обычно информация передается в виде текстовых документов, размеченных с помощью тегов в угловых скобках, но это необязательно. Как правило в качестве транспортного протокола выбирается HTTP, но опять же не всегда. WCF – это каркас для работы с Web-службами на основе XML, который совместим со многими другими технологиями.

Каждый из нас писал код для платформы .NET с момента ее появления (где-то в 1999 году). Мы работаем в корпорации Microsoft, помогая клиентам применять WCF к решению реальных задач. Среди наших клиентов есть и крупные транснациональные корпорации, и независимые поставщики программного обеспечения и Интернет-«стартапы». У каждого свои проблемы, потребности и приоритеты, с которыми мы должны разбираться индивидуально. Мы рассказываем им о том, что можно сделать, рекомендуем хорошо зарекомендовавшие себя подходы и предостерегаем от неверных решений. У нас накопился опыт построения распределенных приложений, и на его основе мы учим других, как применять WCF.

В этой книге мы хотели изложить WCF так, чтобы разработчики программ сразу могли приступить к использованию этой технологии в своих проектах. Материал излагается достаточно детально, чтобы вы поняли, как и для чего применять те или иные возможности. В большинстве случаев мы заходим несколько дальше, описывая некоторые тонкости, но не настолько подробно, как в документации по API.

Различные детали WCF можно найти в блогах. Часть из них принадлежат членам команды разработчиков .NET, а часть – сторонним разработчикам, которые описывают то, что раскопали по ходу работу. Блоги были для нас важным источником информации. В этой книге мы стремились организовать материал так, чтобы его легко было усваивать и лежа на диване, и сидя за столом и вообще в любом месте, где вам сподручно читать.

Для кого написана эта книга?

Мы писали книгу для тех разработчиков программного обеспечения, которые хотят создавать распределенные приложения на платформе .NET. Будучи и сами разработчиками, мы знаем, как важны при освоении новой технологии заслуживающие доверия рекомендации и хорошие примеры. Мы исходили вдоль и поперек блогосферу, прочесали внутренние списки рассылки в Microsoft и написали километры кода, чтобы предложить вам самые лучшие примеры того, как решать стоящие перед вами задачи.

Архитекторы, которым необходимо понимать суть WCF, также найдут для себя много интересного. В главах, посвященных основным понятиям, привязкам, каналам, поведению, размещению, потокам работ и безопасности, описываются важные аспекты проектирования и реализации служб в контексте WCF. Прочитав две-три страницы введения к каждой из этих глав, вы составите представление о технологии с высоты птичьего полета.

Работая над этой книгой, мы ставили себе целью сократить кривую обучения WCF. Мы рассказываем и показываем, как решать типичные задачи, излагаем как основы, так и более сложные темы. Весь материал излагается в виде последовательности подлежащих решению проблем. Мы не дублируем документацию по API, а описываем, как эти проблемы решаются с применением WCF.

Предварительных условий для чтения этой книги не так уж много. Если вас интересует WCF, то, наверное, с основами .NET вы уже знакомы. Вероятно, вы программируете на языке C# либо Visual Basic или хотя бы писали на них раньше. И, разумеется, вы ориентируетесь в Visual Studio. Таким образом, мы предполагаем, что вы умеете писать приличный код для .NET и хотите оптимально потратить время на профессиональное овладение WCF.

Требования к программной среде

WCF – это один из основных компонентов Microsoft .NET Framework 3.x. Впервые WCF была включена в состав .NET 3.0, а в версии .NET 3.5 доработана. Различие между двумя редакциями не слишком велики: усовершенствованы Web-службы, не основанные на протоколе SOAP, улучшена интеграция между WCF и WF и включен пакет обновлений, устраняющих ошибки. В этой книге рассматривается редакция, включенная в .NET 3.5. Мы настоятельно рекомендуем использовать именно ее, если нет особых причин для работы с прежней версией.

Есть два вида дистрибутивов .NET: библиотеки времени исполнения, пригодные для последующего распространения, и комплект средств для разработки ПО (SDK). Библиотеки времени исполнения устанавливаются на конечные машины, а не на машины, где ведется разработка. К этой категории относятся компьютеры для тестирования, подготовки данных и промышленной эксплуатации. SDK предназначен для самих разработчиков. В него включены примеры кода, документация и полезные инструменты. И тот и другой дистрибутив можно загрузить с сайта Microsoft MSDN по адресу http://msdn2.microsoft.com/en-us/netframework/default.aspx. SDK для .NET 3.5 поставляется также с Visual Studio 2008. Продукт

Microsoft .NET Framework 3.5 можно установить на Windows XP SP2, Windows Vista, Windows Server 2003 и Windows Server 2008.

Организация материала

Мы не ожидаем, что вы будете читать книгу от корки до корки. Если вы раньше не работали с WCF, то имеет смысл сначала прочитать главу 1 «Основные понятия» и проработать приведенные в ней примеры. В каждой из последующих глав рассматриваются какие-то конкретные особенности WCF. Каждая глава начинается с краткого введения, в котором описывается, для чего нужна та или иная возможность и какие цели преследовались в процессе ее проектирования. Далее рассматриваются различные вопросы, относящиеся к теме главы.

Глава 1 посвящена основам создания и потребления WCF-служб. Мы покажем, как реализуются различные типы интерфейсов, и обсудим, какой из них выбрать в конкретной ситуации. К концу этой главы вы сможете написать производителя и потребителя WCF-службы.

В главе 2 рассматриваются три основных типа контрактов в WCF: контракт о службе, контракт о данных и контракт о сообщениях. Каждый из них позволяет программно определять сложные структуры и интерфейсы. Контракт о данных отображает типы .NET на XML, контракт о службе описывает оконечные точки интерфейса службы в виде WSDL-документа, который может потребляться на любой платформе, а контракт о сообщениях позволяет напрямую работать с XML-кодом сообщения, а не на уровне типов .NET. Для каждого типа контракта входящие в состав WCF инструменты генерируют и экспортируют во внешний мир WSDL-документ, отвечающий принятым стандартам.

В главе 3 рассматриваются каналы и их стеки. Архитектурная модель каналов – это фундамент, на котором возведен весь коммуникационный каркас WCF. Канал позволяет службам и их клиентам отправлять и принимать сообщения. Вы можете построить стек каналов, точно отвечающий вашим потребностям.

В главе 4 описано, как сконфигурировать коммуникационный стек, чтобы использовались необходимые вам протоколы. Например, если система работает внутри предприятия, не выходя за границы брандмауэра, и необходима максимальная эффективность, то лучше всего воспользоваться привязкой `netTcpBinding`. Если же вы хотите обслуживать всех без исключения Web-клиентов, то придется ограничиться протоколом HTTP и текстовыми сообщениями в формате XML; к вашим услугам привязка `basicHttpBinding`. Привязка – это синоним заранее сконфигурированного стека каналов.

В главе 5 описываются поведения служб. В WCF поведением называется механизм воздействия на работу службы вне контекста обработки самого сообщения. Все, что происходит после получения сообщения, но до того, как оно отправлено коду операции службы, относится к сфере поведений. Здесь решаются задачи управления параллелизмом и временем жизни экземпляров, а также обеспечивается поддержка транзакционности. В этой главе мы покажем, как самостоятельно написать поведение для нестандартного управления службой.

В главе 6 описывается процедура сериализации данных, в результате которой тип .NET (класс) превращается в информационный набор XML (XML Infoset), а также способ передачи информационного набора по проводам. Обычно мы представляем себе XML-документ как текст, в котором имена полей и значения окружены угловыми скобками, но информационный набор XML – более фундаментальная структура данных. В этой главе мы обсудим способы преобразования этой структуры в формат, допускающий передачу по сети.

В главе 7 речь идет о различных вариантах размещения WCF-служб. Подробно описывается самый распространенный случай – размещение в IIS, но он **далеко** не единственный. WCF-службы можно размещать в управляемых .NET-приложениях, в составе Windows Activation Services и в любой другой .NET-программе. В этой главе обсуждаются имеющиеся возможности и способы размещения.

Весьма объемная глава 8 посвящена разнообразным средствам обеспечения безопасности. Обсуждаются и демонстрируются различные схемы аутентификации. Сравниваются механизмы безопасности на транспортном уровне и на уровне сообщений, приводятся примеры того и другого. Описываются сценарии работы в сетях Интернет и Интранет.

В главе 9 рассказано, как пользоваться встроенными в .NET средствами трассировки для отслеживания событий WCF. Описываются классы-прослушиватели (Trace Listener), и на примерах показывается, как конфигурировать параметры для различных событий. Также описывается поставляемый в составе WCF многофункциональный инструмент Trace Viewer, который позволяет трассировать поток сообщений через границы вызова службы.

Глава 10 – это практическое руководство по обработке исключений в WCF. Передача информации об ошибках по протоколу SOAP описывается с помощью контрактов об ошибках, и на примерах демонстрируется, как возбуждать и перехватывать исключения.

Глава 11 посвящена интеграции WCF с технологией Windows Workflow Foundation (WF), поддерживаемой в Visual Studio 2008 и .NET 3.5. Мы покажем, как обратиться к WCF-службам из WF и как сделать потоки работ WF доступными для WCF.

В главе 12 показано, как строятся системы с равноправными клиентами (пиринговые), которые могут обнаруживать друг друга в сети. Мы рассмотрим адресацию в ячеистой сети (mesh network) и способы установления соединения точка-точка после разрешения адресов клиентов.

Глава 13 посвящена использованию WCF для организации Web-служб, не основанных на протоколе SOAP. В качестве примеров рассмотрена технология Asynchronous JavaScript and XML (AJAX) и простой формат данных JSON, ориентированный на JavaScript-программы. Описываются классы, предназначенные специально для протоколов, отличных от SOAP. Как и интеграция WCF с WF, эти возможности появились только в .NET 3.5.

Наконец, в приложении рассматриваются более сложные вопросы, для которых не нашлось места в других главах. Мы решили не «закапывать» их там, где им не место, а рассмотреть отдельно.

Из-за масштабности самого предмета мы не смогли раскрыть все темы одинаково полно. Цель книги состоит в том, чтобы помочь разработчикам радикально повысить производительность своего труда с помощью WCF. Если нам это удалось, то читатели будут пользоваться этой книгой по ходу изучения технологии. Мы не пытались документировать WCF; эту задачу успешно решили технические писатели в Microsoft, подготовившие оперативную справку и материалы в MSDN. Сочетание документации с хорошим руководством, которое вы найдете на страницах этой книги, – залог того, что разработчики смогут быстро приступить к созданию надежных приложений на основе WCF.

Продолжение кода на следующей строке. Если строка кода не умещается в одной строке текста, мы разбиваем ее. В таком случае в конце строки ставится значок стрелочки ➥.

Благодарности

В подготовке этой книги принимало участие множество людей. Мы начали писать ее свыше двух лет назад, когда была выпущена первая публичная бета-версия проекта «Indigo». На протяжении этого времени мы готовили примеры, тестировали и дорабатывали их с выходом каждого обновления. Последний раз это было проделано после выпуска .NET 3.5 и Visual Studio 2008. Попутно мы писали книгу, которую вы сейчас держите в руках. Работа с такой быстро изменяющейся технологией доставляла нам истинное удовольствие.

Эта книга не вышла бы в свет без мощной поддержки со стороны команды разработчиков WCF и других знающих людей в Microsoft. Все они внимательно просматривали наш текст и код и поправляли, когда мы сбивались с пути. Мы хотим поблагодарить следующих людей за то, что они тратили свое время, делились мыслями и были неизменно терпеливы: Вен Лон Донга (Wenlong Dong), Билла Эвьена (Bill Evjen), Стива Мейна (Steve Maine), Дуна Пэрди (Doug Purdy), Рави Рао (Ravi Rao), Яссера Шохуда (Yasser Shohoud) и Дэвида Стэмпфли (David Stampfli).

Мы также благодарны техническим редакторам, которые читали книгу, высказывали свои замечания, спорили с нами и в конечном итоге сделали книгу гораздо лучше. Нам посчастливилось работать с некоторыми из наиболее авторитетных специалистов по WCF. Выражаем искреннюю признательность Николаса Аллену (Nicholas Allen), Джеффу Барнсу (Jeff Barnes), Рону Ландерсу (Ron Landers), Соуми Сринивасану (Sowmy Srinivasan), Тому Фуллеру (Tom Fuller) и Вилли-Петеру Шаубу (Willy-Peter Schaub). Особая благодарность Джону Джастису (John Justice), который отыскал для нас рецензентов в команде разработчиков. Отдельное спасибо также Тому Роббинсу (Thom Robbins), который научил нас писать на литературном английском языке.

Мы также благодарим Лиам Спает (Liam Spaeth) и весь коллектив технического центра Microsoft (Microsoft Technology Center) за поддержку. Идеи поступали от сотрудников отделений MTC по всему миру и в особенности бостонского отделения.

Помимо сотрудников корпорации Microsoft, мы хотим выразить благодарность Киту Брауну (Keith Brown) и Мэтту Милнеру (Matt Milner) из компании PluralSight за скрупулезную проверку материалов по безопасности и потокам работ. Эти две тема весьма глубоки и новы для нас, поэтому их опыт оказался неоценимым подспорьем. И не забудем сотрудников издательства Addison-Wesley, которые собрали все это воедино. Возможно, мы знаем, как программировать и описывать программы, но они знают, как сделать книгу. Поэтому мы благодарим Джоан Мюррей (Joan Murray), Бэтси Харрис (Betsy Harris) и всех прочих.

Об авторах

Стив Резник работает в корпорации Microsoft с середины 1990-х годов. Он примерил на себя роли архитектора, разработчика и евангелиста. Специализируется в Интернет-технологиях, разработке архитектуры и проектировании нагруженных Web-приложений. Стив занимает должность национального технического директора в технических центрах Microsoft в США, в его обязанности входит выработка стратегии и направление усилий своей команды на решение самых сложных задач, с которыми обращаются клиенты. Он работает на платформе .NET с самого момента ее появления и является экспертом по Web-службам, BizTalk-серверу, обработке транзакций и смежным технологиям. Получил ученые степени магистра и бакалавра по информатике соответственно в Бостонском университете и в Университете штата Делавэр.

Рич Крейн – технический архитектор в отделении Технического центра Microsoft в Уолтхоме, штат Массачусетс. Имея 18-летний опыт архитектурного проектирования и разработки, последние шесть лет Рич помогает клиентам создавать решения на платформе Microsoft. Он работал с самыми разными продуктами и технологиями Microsoft и является экспертом в области BizTalk, SQL Server, SharePoint, Compute Cluster Server и, разумеется, Visual Studio и .NET Framework. Он выступал с докладами на различных конференциях, в частности TechEd и Code Camp. С отличием закончил Дрексельский университет, получив степень бакалавра по электронике и вычислительной технике.

Крис Боуэн занимает в корпорации Microsoft должность Developer Evangelist по северо-восточным штатам и специализируется на инструментах разработки, платформах и распространении передового опыта. Архитектор и инженер с 15-летним стажем, Крис перед приходом в Microsoft работал на ведущих должностях в компаниях Monster.com, VistaPrint, Staples и IDX Systems, а также оказывал консультационные услуги по присутствию в Web и по проектам в сфере электронной коммерции. Он является соавтором книги *Professional Visual Studio 2005 Team System* (2006, WROX), имеет степени магистра по информатике и бакалавра по информационно-управляющим системам, которые получил в Ворчестерском политехническом институте.

Глава 1. Основные понятия

Технология Windows Communication Foundation (WCF) – это службы. Их создание, размещение, потребление и обеспечение безопасности. Это стандарты и интероперабельность. Это производительность труда разработчиков. Короче говоря, это средства организации распределенных вычислений, предоставленные в распоряжение профессиональных разработчиков.

В этой главе мы рассмотрим основные понятия, которыми вы должны владеть, если хотите работать с WCF-службами. Мы сконцентрируем внимание на наиболее употребительных возможностях. Проработав текст и примеры, вы сможете создавать и потреблять службы, работающие как локально, так и в сети.

Почему именно WCF?

Прежде чем отвечать на вопрос *как*, нужно понять, *почему*. Итак, почему технология WCF так важна? Ответ прост: потому что службы составляют основу глобальной распределенной сети, а WCF – это самый простой способ предоставлять и потреблять службы на платформе Microsoft. Пользуясь WCF, разработчики могут сосредоточиться на приложениях, а не на коммуникационных протоколах. Это классический пример инкапсуляции технологии в инструментальных средствах. Труд разработчика будет более производительным, если применяемые им инструменты всюду, где возможно, инкапсулируют (но не скрывают) технические детали. WCF в сочетании с Visual Studio 2008 именно это и делает.

В современных архитектурах приложений принимаются во внимание устройства, клиентское программное обеспечение и службы. Без сомнения, возникшая примерно в 1995 году модель Web-сайта (когда приложение размещается на Web-сервере и доставляет пользовательский интерфейс в виде HTML-разметки любому браузеру) не прекратит существования, но одновременно получают распространение более современные модели, сочетающие локальное ПО с Web-службами. В качестве примеров можно назвать iPod, XBOX 360, RSS, AJAX, Microsoft Office, SharePoint и трехмерные среды с погружением; во всех этих случаях установленное на локальном компьютере ПО работает совместно с Web-службами.

Для приложений-потребителей с 2008 года превалирует интерфейс к Web-службам на основе стандарта REST (Representational Entity State Transfer), который сочетает протокол HTTP и четко определенную схему составления URI для адресации XML-данных. REST предполагает, что манипуляции с данными описываются паттерном CRUD (Create Read Update Delete – «создать-читать-обновить-удалить»); основной отличительной особенностью REST является простота.

Для бизнес-приложений основным интерфейсом к Web-службам в 2008 году стал протокол SOAP (Simple Object Access Protocol). Он обеспечивает наиболее надежную модель обмена сложными данными. Сообщения SOAP состоят из конверта и тела, поэтому их можно зашифровать и безопасно передать по сети Интернет. Если сообщение является частью логического сеанса или транзакции, то эта семантика прописывается в конверте и распространяется вместе с сообщением. Если информацию необходимо защитить, то тело сообщения шифруется, а информация о способе защиты помещается в конверт. Сообщения SOAP строго типизированы, что дает дополнительное удобство для разработчика. Как и в случае REST, сообщения SOAP обычно передаются по протоколу HTTP в текстовом формате.

WCF безразлична к протоколу передачи и формату сообщения. В главе 2 описываются службы, основанные на формате сообщений SOAP. В главе 13 то же самое делается для протокола REST. Хотя между ними существуют некоторые тонкие, но существенные отличия, вы увидите, что в моделях программирования больше сходства, нежели различий.

Какой бы протокол передачи ни выбрать, для написания хорошего кода требуются солидные знания и опыт. Те разработчики, которые пишут код служб, включающих бизнес-транзакции, или реализуют клиент с удобным интерфейсом пользователя, обычно предпочитают не работать с XML напрямую. Почему? Потому что за десятилетия исследований в области языков и проектирования компиляторов были созданы гораздо более подходящие инструменты. Работая на уровне объектов, классов и компонентов, вы получите гораздо более надежный код, чем в результате трудоемких манипуляций с XML-разметкой.

При создании приложений для .NET разработчики пользуются средой Visual Studio. В WCF и в Visual Studio есть инструменты для реализации служб. В WCF также встроена модель размещения, позволяющая размещать службы в IIS или в Managed Services на платформе Windows. WCF поддерживает развитую модель многопоточности и ограничения пропускной способности (throttling), которая позволяет управлять созданием экземпляров с минимальными усилиями. Вне зависимости от того, обрабатываются ли одновременно поступающие запросы в одном или в нескольких потоках, модель программирования остается одинаковой, так что разработчик может не вдаваться в детали (которые, однако, остаются ему доступными).

WCF поддерживает различные способы обмена сообщениями, например: запрос-ответ, односторонний и дуплексный поток. Поддерживаются также пиринговые сети, в которых клиенты могут обнаруживать друг друга и обмениваться данными в отсутствие централизованного механизма управления.

Короче говоря, технология WCF важна потому, что современные приложения немыслимы без служб, а именно это и составляет назначение и смысл WCF.

Введение

Будучи всеобъемлющей системой работы со службами, WCF вводит терминологию, с которой вы должны быть знакомы. Термины не обязательно обозначают какие-то новые концепции, однако описывают согласованную систему понятий, которая необходима для обсуждения новой технологии.

В основе своей служба – это множество *оконечных точек* (endpoints), которые предоставляет клиентам некие полезные возможности. Оконечная точка – это просто сетевой ресурс, которому можно посылать сообщения. Чтобы воспользоваться предоставляемыми возможностями, клиент посылает сообщения оконечным точкам в формате, который описывается контрактом между клиентом и службой. Службы ожидают поступления сообщений на адрес оконечной точки, предполагая, что сообщения будут записаны в оговоренном формате. На рис. 1.1 схематически представлено отношение между клиентом и службой.

Рис. 1.1. Обмен данными между клиентом и службой

Чтобы клиент мог передать службе осмысленную информацию, он должен знать АПК: адрес, привязку и контракт.

- **«А» обозначает адрес, то есть «куда»**. Адрес определяет, куда следует отправлять сообщения, чтобы оконечная точка их получила. В случае протокола HTTP адрес будет выглядеть так: http://myserver/myservice/, а в случае TCP так: net.tcp://myserver:8080/myservice.
- **«П» обозначает привязку, то есть «как»**. Привязка определяет *канал* для коммуникаций с оконечной точкой. По каналам передаются все сообщения, циркулирующие в приложении WCF. Канал состоит из нескольких *элементов привязки* (binding element). На самом нижнем уровне элемент привязки – это транспортный механизм, обеспечивающий доставку сообщений по сети. В WCF встроены следующие транспорты: HTTP, TCP, Named Pipes, PerChannel и MSMQ. Элементы привязки, расположенные выше, описывают требования к безопасности и транзакционной целостности. К счастью, WCF поставляется с набором готовых привязок, в которых каналы уже собраны и сконфигурированы, чтобы вы не тратили на этой время. Привязка `basicHttpBinding` применима для доступа к большинству Web-служб, созданных до 2007 года. Она соответствует спецификации WS-I BP 1.1 и включена потому, что обеспечивает максимальную интероперабельность. Привязка `wsHttpBinding` реализует семейство протоколов WS-*, обеспечивающих безопасный, надежный и транзакционный обмен сообщениями.
- **«К» обозначает контракт, то есть «что»**. Контракт определяет набор функций, предоставляемых оконечной точкой, то есть операции, которые она может выполнять, и форматы сообщений для этих операций. Описанные в контракте операции отображаются на методы класса, реализующего оконечную точку, и включают в частности типы параметров, передаваемых каждому методу и получаемых от него.

Как показано на рис. 1.2, WCF-служба может состоять из нескольких оконечных точек, каждая из которых описывается собственным адресом, привязкой и контрактом. Поскольку поток сообщений обычно двунаправленный, клиенты неявно также оказываются контейнерами оконечных точек.

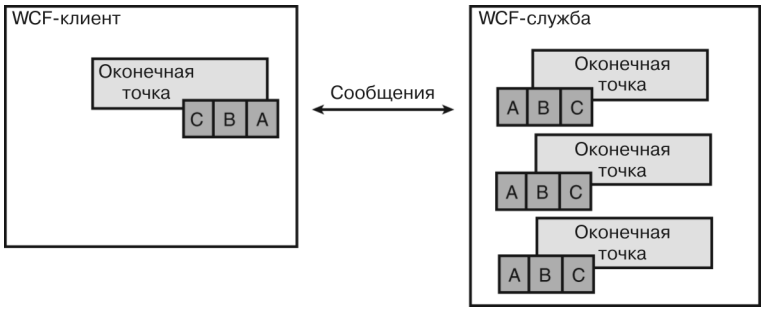

Рис. 1.2. Коммуникация между оконечными точками клиента и службы

Оконечная точка службы не может отвечать на сообщения, если служба не размещена в каком-нибудь работающем процессе операционной системы. Владельцем службы может быть любой процесс, например, работающее без присмотра человека серверное приложение, Web-сервер и даже клиентская программа, представленная полноценным окном на экране ПК или значком в системном лотке Windows. Для служб можно определить поведения, управляющие степенью параллелизма, ограничением пропускной способности, транзакционной целостностью, безопасностью и другими семантическими аспектами. Поведения можно реализовать с помощью атрибутов .NET, путем манипулирования исполняющей средой WCF или в конфигурационных файлах. В сочетании с гибкой моделью размещения поведения заметно упрощают написание многопоточного кода.

Как показано на рис. 1.3, главная программа может создать экземпляр класса `ServiceHost`, который будет отвечать за создание оконечных точек службы.

Чтобы службы можно было найти, предусмотрена специальная инфраструктурная оконечная точка, называемая Metadata Exchange (MEX) (обмен метаданными). Клиенты могут обращаться к ней, если хотят получить описания АПК службы на языке Web Service Description Language (WSDL). Оконечная точка MEX вызывается, когда вы щелкаете по узлу Add Service Reference (Добавить ссылку на службу) в Visual Studio 2008 или запускаете утилиту `svcutil.exe` на этапе проектирования. По получении WSDL-документа генерируются два артефакта: прокси-класс на языке проекта и файл `app.config`. Прокси-класс описывает сигнатуры операций оконечной точки, так что клиентский код может просто «вызывать» точку. Интерфейс прокси-класса не обязательно должен точно повторять сигнатуру службы, но сообщение, доставляемое службе с его помощью, должно быть точно таким, как описано в контракте о службе. Файл `app.config` содержит детальные сведения о привязках.

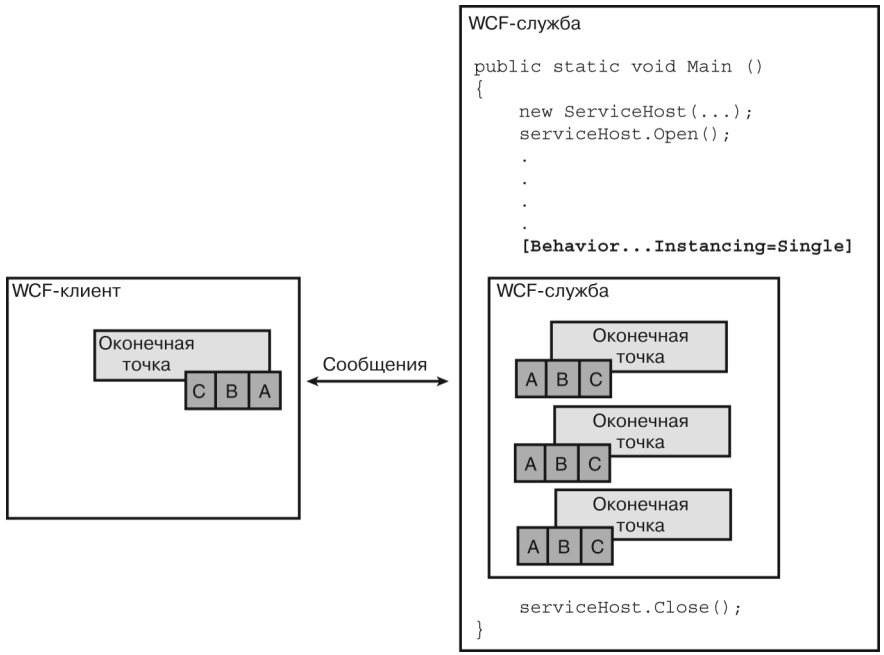

Рис. 1.3. Размещение службы

Реализация WCF-службы

В этом разделе мы опишем реализацию простой службы с помощью WCF. Говоря «простая», мы имеем в виду, что транспортным протоколом является HTTP, а сообщение представлено в виде XML-документа. Предполагается, что безопасность каким-то образом обеспечивает само приложение. Будем считать, что коммуникация представляет собой синхронный диалог типа запрос-ответ и что наша служба поддерживает только одну операцию, которая принимает на входе строку, а возвращает число типа double. В следующих главах все эти допущения будут подвергнуты ревизии, но пока обойдемся без лишних сложностей.

Всего лишь АПК

Чтобы определить оконечную точку службы, нам понадобится АПК: адрес, привязка и контракт. В листингах с 1.1 по 1.3 описываются следующие аспекты:

- **«А» – адрес или *куда***. Наша служба будет ожидать входящие запросы по адресу http://localhost:8000/EssentialWCF;
- **«П» – привязка или *как***. В данном случае воспользуемся привязкой `basicHttpBinding`, которая заставит WCF реализовать спецификацию WS-I Basic Profile 1.1 – самый распространенный набор протоколов для взаимодействия с Web-службами;

- **«К» – контракт или *что***. Это синтаксическое описание операций, на которые отвечает служба, и форматов входных и выходных сообщений. В этом примере контракт определен классом `StockService`.

Мы реализуем эту службу дважды. Сначала покажем, как это можно сделать программно, когда АПК целиком определен в исходном коде. Тем самым мы избегнем каких-либо внешних зависимостей. Затем продемонстрируем, как то же самое можно сделать с помощью конфигурационных файлов. Кода при этом потребуется писать меньше, зато сложность службы возрастет из-за наличия зависимостей между кодом и конфигурацией. На практике вы, скорее, предпочтете второй подход, так как некоторое усложнение с лихвой окупается приобретаемой гибкостью. Гибкость проистекает из того факта, что часть функциональности выносится в конфигурационные файлы, которые может изменять администратор, а часть остается в коде, так что модифицировать ее сможет только программист.

Полностью программная реализация службы

На некоем сверхвысоком уровне написание WCF-службы аналогично написанию любой другой службы и не зависит от ее семантики. Сначала пишется код, реализующий какую-то функцию, затем этот код размещается в процессе операционной системы, а этот процесс ожидает поступления запросов и отвечает на них. WCF в какой-то мере формализует эти шаги, помогая разработчику не допустить ошибок на каждом этапе. Например, с помощью готовых привязок и кодировщиков WCF-службы могут обмениваться стандартными SOAP-сообщениями. Механизмы многопоточности, параллелизма и порождения экземпляров по умолчанию имеют реализации, отличающиеся вполне предсказуемым поведением.

Для реализации WCF-службы требуется написать класс на одном из языков .NET, а затем снабдить его атрибутами из пространства имен `System.ServiceModel`. Это пространство имен устанавливается вместе с .NET 3.0 и содержит большую часть кода WCF. Во время компиляции вашего класса CLR интерпретирует атрибуты, подставляя вместо них исполняемый код. Атрибуты – не новость в .NET; они существуют еще со времен .NET 1.0. В WCF, как и в ASMX в .NET 1.0, 1.1 и 2.0, атрибуты применяются ради упрощения и ускорения процесса написания служб.

В листинге 1.1 приведен полный код WCF-службы, размещаемой в консольном приложении. Вот что мы делаем:

- **Определяем контракт**. Пишется класс, который делает нечто полезное, после чего он снабжается атрибутами WCF. Атрибут `[ServiceContract]` помечает класс, как контракт. В терминах языка WSDL `[ServiceContract]` определяет тип порта `PortType`. Атрибут `[OperationContract]` определяет методы класса, которые можно вызывать через интерфейс службы. Одновременно он определяет, какие сообщения можно передать этим методам и получить от них. С точки зрения WSDL, этот атрибут соответствует разделам Operations и Messages. В листингах 1.1–1.3 определен класс `StockService`, содержащий единственный метод GetPrice.

Примечание. В примерах из этой книги мы намеренно упрощаем интерфейсы, ограничиваясь обычно приемом или возвратом единственной строки или числа. На практике операции ваших служб, скорее всего, должны будут принимать и возвращать данные составных типов. При сетевом обмене следует стремиться к минимизации трафика и задержек, а для этого при каждом обращении нужно передавать как можно больше информации, из-за чего и возникает потребность в составных типах.

- **Определяем оконечную точку**. В определении оконечной точки адрес, привязка и контракт задаются с помощью метода `AddServiceEndpoint` класса `ServiceHost`. Адрес мы оставляем пустым, это означает, что адрес оконечной точки такой же, как адрес самой службы. В качестве привязки указывается `basicHttpBinding`, совместимая со спецификацией WS-I BP 1.1 и обеспечивающая интероперабельность с большинством систем, в которых реализованы Web-службы на базе XML. Спецификация WS-I, или Web Services Interop, появилась в результате совместных усилий крупнейших производителей программного обеспечения – Microsoft, IBM, BEA, Oracle и других, направленных на выработку и опубликование уровней совместимости. WS-I – это не стандарт, а руководство и комплект инструментов для определения того, насколько ПО отвечает существующим стандартам, например HTTP и XML.
- **Размещаем службу в процессе, чтобы она могла прослушивать входящие запросы**. В листингах 1.1–1.3 служба размещается в консольном приложении с помощью класса `ServiceHost`. Служба ожидает поступления запросов на адрес http://localhost:8000/EssentialWCF.

Листинг 1.1. Полностью программная реализация службы

```
using System;
using System.ServiceModel;

namespace EssentialWCF;
{
   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      double GetPrice(string ticker);
   }

   public class StockService : IStockService
   {
      public double GetPrice(string ticker)
      {
         return 94.85;
      }
   }

   public class Service
```

```
   {
      public static void Main()
      {
         ServiceHost serviceHost = new
                         ServiceHost(typeof(StockService)),
                         new Uri("http://localhost:8000/EssentialWCF"));

         serviceHost.AddServiceEndPoint(
                     typeof(IStockService),
                     new BasicHttpBinding(),
                     "");
         serviceHost.Open();

         Console.WriteLine("Для завершения нажмите <ENTER>.\n\n");
         Console.ReadLine();

         serviceHost.Close();
      }
   }
}
```

Реализация службы с помощью кода и конфигурационных файлов

В WCF имеется развитая поддержка для определения атрибутов службы в конфигурационных файлах. Кодировать алгоритм работы службы все равно придется, но задание адресов, привязок и поведений оконечных точек можно перенести из кода в конфигурационные файлы.

Задание оконечных точек и поведения в конфигурационных файлах обеспечивает большую гибкость по сравнению с заданием в коде. Предположим, например, что оконечная точка была написана для работы с клиентами по протоколу HTTP. В листинге 1.1 это реализуется за счет того, что методу `AddServiceEndPoint` передается объект типа `BasicHttpBinding`. А теперь допустим, что вы решили изменить привязку на `WSHttpBinding`, поскольку она обеспечивает повышенную безопасность, добавляя к защите на уровне транспортного протокола еще и защиту на уровне отдельных сообщений. В таком случае пришлось бы изменить и перекомпилировать программу. Если же перенести выбор привязки из кода в конфигурацию, то можно будет обойтись без перекомпиляции. А если вы захотите распространить контракт на оба протокола, достаточно будет определить две оконечные точки: одну для базового протокола HTTP, а другую для протокола WS-Security; ничего изменять в коде при этом не придется. За счет этого сопровождение программы упрощается.

В листинге 1.2 приведен полный код WCF-службы, размещаемой в консольном приложении. Для нее требуется конфигурационный файл, в котором содержится информация о поведении и оконечной точке. В данном примере мы делаем следующее:

- **Определяем контракт**. Пишем класс, выполняющий полезную работу, и снабжаем его атрибутами WCF. Код самой службы одинаков что при программном определении, что при описании в конфигурационных файлах. В листинге 1.2 класс называется `StockService`, как и в листинге 1.1
- **Размещаем службу в процессе операционной системы, чтобы клиенты могли обращаться к ней из сети**. Для этого создается объекта класса `ServiceHost`, определенного в пространстве имен `System.ServiceModel`, и вызывается его метод `Open` точно так же, как в листинге 1.1.
- **Создаем конфигурационный файл, в котором определяется базовый адрес службы и АПК ее оконечной точки**. Отметим, что код в листинге 1.2 никак не ссылается на конфигурационный файл. При вызове метода `ServiceHost.Open` WCF ищет в конфигурационном файле приложения (`app.config` или `web.config`) секцию `<serviceModel>`, из которой читает конфигурационные параметры.

Листинг 1.2. Реализация службы с помощью кода и конфигурационного файла

```
using System;
using System.ServiceModel;

namespace EssentialWCF;
{
   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      double GetPrice(string ticker);
   }

   public class StockService : IStockService
   {
      public double GetPrice(string ticker)
      {
         return 94.85;
      }
   }

   public class Service
   {
      public static void Main()
      {
         ServiceHost serviceHost = new
                          ServiceHost(typeof(StockService));
         serviceHost.Open();

         Console.WriteLine("Для завершения нажмите <ENTER>.\n\n");
         Console.ReadLine();

         serviceHost.Close();
      }
   }
}
```

В листинге 1.3 приведен конфигурационный файл, используемый совместно с программой из листинга 1.2. В секции `<serviceModel>` определена оконечная точка. Для каждой оконечной точки задаются адрес, привязка и контракт. Адрес в данном случае пуст, это означает, что нужно использовать тот же адрес, что для самой службы. Если у службы несколько оконечных точек, то у каждой должен быть уникальный адрес. В качестве привязки мы указали `basicHttpBinding`, а в качестве имени контракта – имя класса, определенного в коде, – `EssentialWCF.StockService`.

Листинг 1.3. Конфигурационный файл для службы, реализованной в листинге 1.2

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>

      <services>
         <service name="EssentialWCF.StockService">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8080/EssentialWCF"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                      binding="basicHttpBinding"
                      contract="EssentialWCF.IStockService" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

Еще о конфигурационных файлах

Конфигурационный файл службы, `web.config` или `app.config` в зависимости от того, где служба размещена, должен содержать секцию `<system.serviceModel>`. Внутри нее определяются службы, привязки, поведения, клиенты, способы диагностики, расширения, параметры размещения и взаимодействия с COM+. Как минимум, должна существовать подсекция `<services>`, в которой описаны оконечные точки, а в ней хотя бы один узел `<endpoint>`, описывающий не инфраструктурную точку. АПК самой оконечной точки определяются внутри этого узла.

Атрибут `address` определяет URI, на который клиенты будут посылать сообщения оконечной точке. Например, если для службы определена привязка к протоколу HTTP `basicHttpBinding`, то URI может выглядеть как *http://www.myserver.com:8080/MyService/*. Если задан абсолютный адрес (то есть не пустой и содержащий не только путь), то он замещает базовый адрес, заданный владельцем службы в момент ее создания. Когда владелец запускает службу, WCF начинает прослушивать указанный адрес в ожидании входящих запросов. Если в качестве

владельца выступает IIS, то, скорее всего, прослушиватель уже запущен, поэтому WCF сообщает ему о себе – регистрируется, чтобы запросы на этот URI направлялись WCF-службе.

Атрибут `binding` описывает коммуникационные детали, необходимые для соединения со службой. В частности, определяется весь стек каналов, который должен как минимум включать канал сетевого адаптера. Помимо этого, можно включить в стек каналы шифрования, сжатия и другие. В комплекте с WCF поставляется ряд уже сконфигурированных привязок, например: `BasicHttpBinding`, совместимая с ASMX; `WSHttpBinding`, реализующая более развитые Web-службы, для которых требуется безопасность на уровне сообщений, поддержка транзакций и другие возможности, и `NetTcpBinding`, реализующая быстрый и безопасный формат передачи сообщений, аналогичный .NET Remoting и DCOM.

Атрибут `contract` ссылается на тип, определенный для оконечной точки службы. WCF инспектирует этот тип и раскрывает его метаданные в виде оконечной точки MEX, если таковая входит в состав службы. Указанный тип WCF ищет сначала в папке \bin, а затем в глобальном кэше сборок (GAC) на данной машине. Если найти тип не удается, то служба возвращает информацию об ошибке, когда вы щелкаете по узлу Add Service Reference в Visual Studio или запускаете утилиту `svcutil.exe` для генерации WSDL-документа. Если оконечная точка MEX не задана, то служба будет работать нормально, но клиенты не смогут опрашивать ее АПК.

Еще о размещении служб

WCF позволяет размещать службы в любом процессе операционной системы. В большинстве случаев наиболее подходящим владельцем является IIS, который обеспечивает высокую производительность, удобство управления и безопасность. Если в вашем окружении IIS работает, то инфраструктура обеспечения безопасности уже присутствует. IT-департаменты крупных компаний часто определяют явные политики и процедуры обеспечения безопасности и имеют автоматизированные инструменты для проверки на соответствие им. В небольших организациях обычно довольствуются параметрами безопасности, применяемыми в IIS и Windows 2003 по умолчанию. Так или иначе, к WCF-службам, размещенным в IIS, применяются уже определенные правила.

Но иногда использовать IIS в качестве владельца службы нежелательно. Возможно, вас не устраивает протокол HTTP. Или вы хотите явно управлять запуском и остановом службы. Или требуется предоставить специальный административный интерфейс, отличный от того, что имеется в IIS. Что ж, нет проблем. WCF позволяет очень легко и гибко осуществлять *авторазмещение* (self-hosting). Этим термином описывается метод размещения, при котором разработчик сам создает экземпляр владельца службы, не полагаясь ни на IIS, ни на Windows Process Activation Services (WAS).

Простейший способ разместить службу – это написать консольное приложение, как было показано в листинге 1.1. Для промышленной эксплуатации это не

очень полезно, так как открывать окно команд на сервере не рекомендуется. Но для первоначального тестирования службы вполне годится, так как при этом устраняются все зависимости от инфраструктуры IIS. Главная программа создает новый экземпляр класса `ServiceHost`, который, как следует из самого названия, призван стать владельцем службы. Затем программа вызывает метод `Open` этого объекта, и продолжает заниматься своим делом. В данном случае она просто ждет, пока кто-нибудь нажмет клавишу Enter, после чего вызывает метод `Close` объекта `ServiceHost`.

После вызова `Open` объект `ServiceHost` начинает прослушивать адреса, указанные в описаниях оконечных точек. Когда поступает сообщение, `ServiceHost` выполняет несколько действий. Во-первых, в соответствии с описанием стека каналов в привязке, он выполняет дешифрирование, распаковку и применяет правила безопасности. Во-вторых, обращается к контракту для десериализации сообщения, в результате чего создается новый объект. Затем вызывается затребованная операция этого объекта.

Включение оконечной точки обмена метаданными (MEX)

Метаданные в WCF содержат информацию, точно описывающую, как следует обращаться к службе. Запросив у работающей службы метаданные, клиент может узнать о ее оконечных точках и требуемых форматах сообщений. На этапе проектирования клиенты посылают такой запрос в виде сообщения, определенного в стандарте WS-MetadataExchange, и получают в ответ WSDL-документ. Этот документ клиент может использовать для генерации прокси-класса и конфигурационного файла, которые впоследствии будут использоваться для доступа к службе во время выполнения. Эта схема взаимодействия представлена на рис. 1.4.

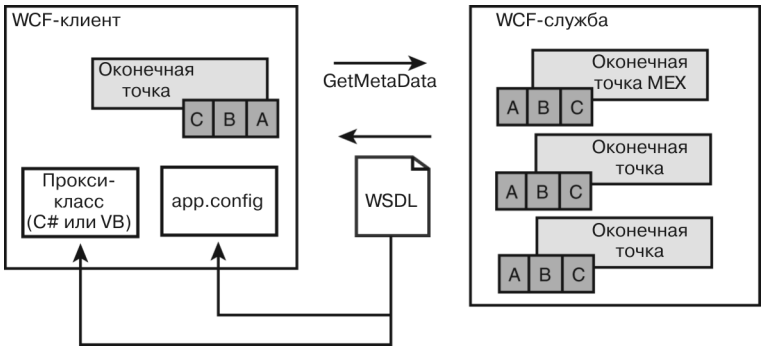

Рис. 1.4. Получение метаданных от оконечной точки MEX

По умолчанию WCF-службы не раскрывают оконечную точку MEX. Это означает, что никто не сможет узнать у службы, как с ней взаимодействовать. Не зная адреса, привязок и контракта, очень трудно обратиться к службе, которая не занесена в реестр. Но к счастью WCF позволяет очень просто раскрыть оконечную точку MEX, чтобы клиенты могли корректно общаться со службой. Это можно сделать как в коде, так и в конфигурационном файле.

В листинге 1.4 приведен код, необходимый для раскрытия оконечной точки MEX службы. Он является продолжением листинга 1.1 в нескольких отношениях. Во-первых, к службе добавлено поведение, которое заставляет WCF включить контракт для точки MEX – `IMetaDataExchange`. Во-вторых, в описание службы добавлена оконечная точка, для которой указан контракт `IMetaDataExchange`, протокол HTTP и адрес `"mex"`. Поскольку адрес относительный, то в его начало дописывается базовый адрес службы, так что полный адрес будет равен http://localhost:8000/EssentialWCF/mex. Отметим, что поведение также модифицировано с целью разрешить запросы HTTP GET. Это не обязательно, но позволяет пользователям напрямую обращаться к оконечной точке MEX из браузера.

Листинг 1.4. Служба, раскрывающая оконечную точку MEX в коде

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;

namespace EssentialWCF;
{
   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      double GetPrice(string ticker);
   }

   public class StockService : IStockService
   {
      public double GetPrice(string ticker)
      {
         return 94.85;
      }
   }

   public class Service
   {
      public static void Main()
      {
         ServiceHost serviceHost = new
                           ServiceHost(typeof(StockService)),
                           new Uri("http://localhost:8000/EssentialWCF"));

         serviceHost.AddServiceEndPoint(
                     typeof(IStockService),
                     new BasicHttpBinding(),
                     "");

         ServiceMetadataBehavior behavior = new
                  ServiceMetadataBehavior();
         behavior.HttpGetEnabled = true;
```

```
            serviceHost.Description.Behaviors.Add(behavior);

            serviceHost.AddServiceEndpoint(
                        typeof(IMetadataExchange),
                        MetadataExchangeBindings.CreateMexHttpBinding(),
                        "mex");

            serviceHost.Open();

            Console.WriteLine("Для завершения нажмите <ENTER>.\n\n");
            Console.ReadLine();

            serviceHost.Close();
        }
    }
}
```

Если вы хотите описывать оконечные точки в конфигурационном файле, а не в коде, то там же нужно описать и оконечную точку MEX. В листинге 1.5 показано, как следует модифицировать конфигурационный файл, приведенный в листинге 1.3. К службе добавлена оконечная точка MEX и поведение, позволяющее обращаться к ней по протоколу HTTP.

Листинг 1.5. Раскрытие оконечной точки MEX в конфигурационном файле

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>

      <services>
         <service name="EssentialWCF.StockService"
                  behaviorConfiguration="myServiceBehavior">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8080/EssentialWCF"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                      binding="basicHttpBinding"
                      contract="EssentialWCF.IStockService" />
            <endpoint address="mex"
                      binding="mexHttpBinding"
                      contract="IMetadataExchange" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="myServiceBehavior">
               <serviceMetadata htthGetEnabled="True"/>
            </behavior>
         </serviceBehaviors>
      </behaviors>

   </system.serviceModel>
</configuration>
```


Реализация клиента WCF-службы

WCF предоставляет развитый API для написания клиентов, желающих взаимодействовать со службой. Этот API, находящийся в пространстве имен `System.ServiceModel`, берет на себя заботу о сериализации типов в XML и отправку сообщений от клиента службе. Вы можете применять API в своих программах напрямую или воспользоваться инструментами для генерации прокси-класса и конфигурационного файла. В этом разделе мы сначала покажем, как обратиться к службе непосредственно, а потом – как сделать то же самое с помощью инструментов. При втором подходе приходится писать меньше кода, а все конфигурационные параметры выносятся во внешние файлы. При первом подходе количество внешних зависимостей меньше, и вы получаете более точный контроль над обращениями. Какой подход предпочесть, зависит от ситуации.

Реализация клиента целиком в коде

Если оконечная точка должна определить свои АПК, чтобы WCF могла раскрыть ее возможности при запросах из сети, то клиент должен знать АПК, если хочет этими возможностями воспользоваться. Поэтому при написании кода, обращающегося к оконечным точкам службы, АПК включаются в клиентское приложение.

С *адресом* оконечной точки все просто – это сетевой адрес, на который отправляются сообщения. Его формат определен транспортным протоколом, заданным в привязке. *Привязка* оконечной точки точно определяет механизм коммуникации, который использует данная точка. В комплект поставки WCF входит ряд заранее сконфигурированных привязок, например: `netTcpBinding`, `wsHttppBinding` и `basicHttppBinding`. *Контракт* определяет точный формат XML, распознаваемый службой. Обычно он задается с помощью атрибутов `[ServiceContract]` и `[DataContract]` в определении класса и/или интерфейса, а WCF сериализует структуру класса в виде XML для передачи по сети.

В листинге 1.6 приведен код для вызова операции службы. В него «зашиты» АПК оконечной точки, позволяющие воспользоваться ее возможностями.

Прежде всего, клиент определяет интерфейс, к которому собирается обратиться. Определение интерфейса является общим для клиента и службы. Синтаксически определение на языке C# сильно отличается от формата XML или WSDL, но семантически они эквивалентны. То и другое представляет собой точное описание того, как обращаться к службе, и включает имя операции и ее параметры. Далее клиент создает экземпляр класса `ChannelFactory`, передавая его конструктору АПК оконечной точки. В данном случае мы указываем адрес сервера IIS, в котором размещена служба, в качестве привязки задаем `BasicHttpBinding`, а в качестве контракта – интерфейс `IStockService`. Наконец, клиент получает от фабрики канал для установления связи со службой и «вызывает метод» службы.

Листинг 1.6. Реализация WCF-клиента целиком в коде

```
using System;
using System.ServiceModel;

namespace Client;
{
   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      double GetPrice(string ticker);
   }

   class Client
   {
      static void Main()
      {
         ChannelFactory<IStockService> myChannelFactory =
         new ChannelFactory<IStockService>(
                           new BasicHttpBinding(),
                           new EndpointAddress
                              ("http://localhost:8000/EssentialWCF"));

         IStockService wcfClient = myChannelFactory.CreateChannel();

         double p = wcfClient.GetPrice("msft");
         Console.WriteLine("Price:{0}",p);
      }
   }
}
```

Реализация клиента с помощью кода и конфигурационного файла

Еще в 2001 году в Visual Studio появилась операция Add Web Reference (Добавить Web-ссылку), которая свела распределенные вычисления к щелчку правой кнопкой мыши. Это было совсем неплохо, так как многие профессиональные разработчики получили в свое распоряжение простой способ создания масштабируемых, основанных на стандартах распределенных приложений. Но, предельно облегчив программирование распределенных систем, Microsoft одновременно скрыла многие важные детали. В Visual Studio 2008 операция Add Web Reference по-прежнему поддерживается ради совместимости с ASMX-файлами и другими Web-службами, но добавилась еще и операция Add Service Reference (ASR) (Добавить ссылку на службу) для поддержки WCF. Поскольку WCF не зависит от протокола и поддерживает различные механизмы сериализации, кодирования и обеспечения безопасности, то ASR оказывается более гибкой с точки зрения удобства управления, производительности и безопасности.

Операция ASR в Visual Studio применяется для получения метаданных от WCF-службы и генерации прокси-класса и конфигурационного файла, как показано на рис. 1.4. Невидимо для вас ASR вызывает программу `svcutil.exe`, которая запрашивает у оконечной точки MEX ее интерфейсы и генерирует прокси-класс и конфигурационный файл. Прокси-класс позволяет клиенту обращаться к операциям службы так, будто они являются методами локального класса. Прокси-класс пользуется классами WCF для конструирования и интерпретации SOAP-сообщений согласно контракту, определенному в оконечной точке службы. В конфигурационном файле хранятся АПК службы.

Для программирования клиента, обращающегося к службе, необходимо, во-первых, сгенерировать конфигурационный файл и прокси-класс, а, во-вторых, написать код, который будет с помощью прокси-класса обращаться к службе. Чтобы воспользоваться операцией ASR в Visual Studio 2008, щелкните правой кнопкой мыши по узлу Service References (Ссылки на службы) в окне Solution Explorer и выберите из контекстного меню пункт Add Service Reference. В результате появится диалоговое окно, изображенное на рис. 1.5.

Из этого окна производится обращение к утилите svcutil с целью создания исходного файла прокси-класса на языке проекта. Также генерируется файл `app.config` с секцией `<system.serviceModel>`, в которой хранится информация об адресе, привязке и контракте, необходимая для вызова оконечных точек.

Вместо ASR можно напрямую вызвать утилиту `svcutil.exe`, которая находится в папке C:\Program Files\Microsoft SDKs\ Windows\v6.0\Bin. У нее много флагов, полное описание которых можно получить, задав флаг `-h`. Эта утилита принимает на входе метаданные, а на выходе может создавать различные файлы. Метаданные могут читаться из DLL, в которой находится реализация класса, из WSDL-файла или из WSDL-документа, возвращаемого в результате обращения к работающей службе с запросом WS-Metadata. В листинге 1.7 показано, как вызвать `svcutil.exe` для генерации метаданных для службы, представленной в листингах 1.4 и 1.5.

Листинг 1.7. Генерация прокси-класса и конфигурационного файла для клиента с помощью svcutil.exe

```
svcutil http://localhost:8080/EssentialWCF/mex/
        -config:app.config
        -out:generatedProxy.cs
```

Какой бы способ ни выбрать, генерируются одинаковые прокси-класс и конфигурационный файл. В листинге 1.8 приведен конфигурационный файл. Обратите внимание, что конфигурационный файл для клиента гораздо подробнее, чем для соответствующей службы (см. листинг 1.3). Это позволяет клиенту гибко переопределять некоторые атрибуты, например, величины таймаутов, размеры буферов и предъявляемые верительные грамоты.

Листинг 1.8. Файл app.config, сгенерированный svcutil.exe

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
```

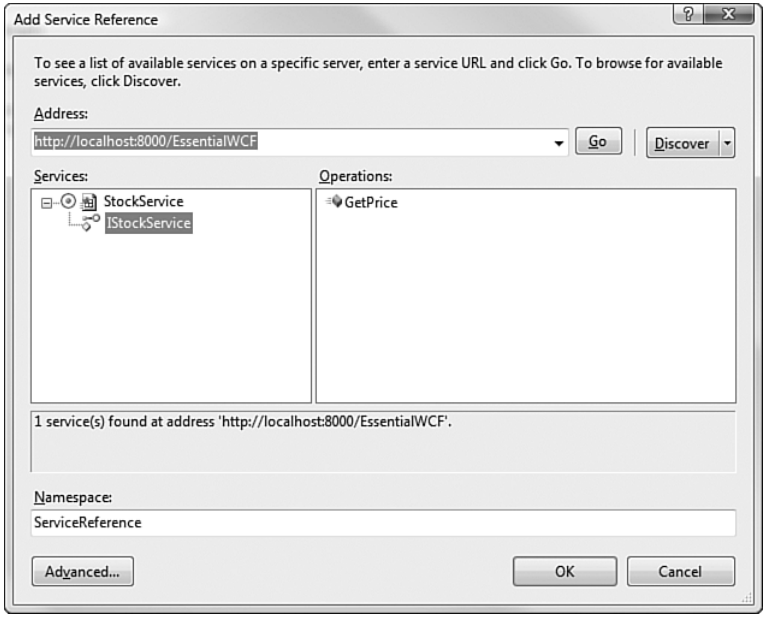

Рис. 1.5. Генерация прокси-класса и конфигурационного файла для клиента в Visual Studio

```
<bindings>
   <basicHttpBinding>
      <binding name="BasicHttpBinding_StockService">
         closeTimeout="00:01:00" openTimeout="00:01:00"
         receiveTimeout="00:10:00" sendTimeout="00:01:00"
         allowCookies="false" bypassProxyOnLocal="false"
         hostNameComparisonMode="StrongWildCard"
         maxBufferSize="65536" maxBufferPoolSize="524288"
         maxReceivedMessageSize="65536"
         messageEncoding="Text" textEncoding="utf-8"
         transferMode="Buffered" userDefaultWebProxy="true"
         <readerQuotas maxDepth="32">
                           maxStringContentLength="8192"
                           maxArrayLength="16384"
                           maxBytesPerRead="4096"
                           maxNameTableCharCount="16384" />
         <security mode="None">
            <transport clientCredentialType="None"
                       clientCredentialType="None"
                       realm="" />
            <message clientCredentialType="UserName"
                       algorithmSuite="Default" />
         </security>
      </binding>
   </basicHttpBinding>
</bindings>
```



```
      <client>
         <endpointAddress="http://localhost:8080/EssentialWCF"
               binding="basicHttpBinding"
               bindingConfiguration="BasicHttpBinding_StockService"
               contract="StockService"
               name="BasicHttpBinding_StockService" />
      </client>
   </system.serviceModel>
</configuration>
```

После того как конфигурационный файл и прокси-класс сгенерированы, обратиться к операции, подразумевающей диалог вида запрос-ответ, совсем просто. Имя прокси-класса образуется из имени контракта о службе, в конец которого дописано слово Client. Для службы, представленной в листингах 1.4 и 1.5, прокси-класс будет называться `StockServiceClient`. В клиентском коде создается экземпляр прокси-класса, а затем вызывается его метод. См. листинг 1.9.

Листинг 1.9. Код клиента, вызывающего операцию службы

```
using System;
using System.ServiceModel;

namespace EssentialWCF;
{
   class Client
   {
      static void Main()
      {
         StockServiceClient proxy = new StockServiceClient();
         double p = wcfClient.GetPrice("msft");
         Console.WriteLine("Price:{0}", p);
         proxy.Close();
      }
   }
}
```

Размещение службы в IIS

WCF-службу можно размещать в любом управляемом процессе операционной системы. Сама служба обычно не знает и не интересуется тем, где она размещена, хотя API позволяет это выяснить, и даже не одним способом. Можно разместить WCF-службу в обычной службе Windows, которая запускается при загрузке компьютера и останавливается при его выключении, а можно в клиентском приложении, свернутом в системном лотке. Но чаще всего WCF-службы размещаются в IIS.

Обсуждение

IIS прекрасно подходит на роль владельца служб. Он уже встроен в Windows, существует обширная литература о том, как им управлять, как обеспечивать безопасность и как разрабатывать для него приложения. IIS хорошо масштабируется,

надежен и может быть сконфигурирован для безопасной работы, так что представляет собой прекрасную платформу для размещения служб. ASMX-файлы, обслуживаемые IIS, до появления WCF были широко распространены в качестве механизма публикации Web-служб, и WCF продолжает эту традицию. В .NET 3.5 для публикации Web-служб через IIS рекомендуется применять WCF вместо ASMX.

Напомним еще раз, что в WCF огромную роль играет АПК: адрес, привязка и контракт. При размещении в IIS адрес службы определяется виртуальным каталогом, который содержит составляющие ее файлы. В качестве привязки всегда используется протокол HTTP/S, поскольку только его IIS и понимает; потому и существуют привязки `basicHttpBinding` и `wsHttpBinding`. Но это лишь две готовые системные привязки, а вообще для размещения в IIS годится любая привязка, связанная с протоколом HTTP. Контракт – определение оконечных точек службы в терминах SOAP – никак не лимитирован тем фактом, что служба размещена именно в IIS, никаких специальных правил для такого размещения нет.

Как и в случае ASMX, метаданные (в виде WSDL-документа) можно получить от размещенной в IIS службы, добавив к ее адресу параметр wsdl (http://localhost/myservice.svc?wsdl). Когда IIS получает такой запрос, он вызывает оконечную точку MEX и возвращает полученный результат в виде WSDL-документа. Но в отличие от ASMX по умолчанию точка MEX не раскрывается, поэтому IIS не отвечает на запросы о метаданных, посылаемые операцией Add Service Reference из Visual Studio 2008 или утилитой `svcutil.exe`. Вы должны явно активировать оконечную точку MEX в коде (см. листинг 1.4) или в конфигурационном файле (см. листинг 1.5).

Три шага размещения службы в IIS

Процедура размещения службы в IIS состоит из трех шагов:

- создать в IIS виртуальное приложение, в котором будут храниться файлы службы;
- создать SVC-файл, ссылающийся на реализацию службы;
- добавить в файл `web.config` секцию `<system.serviceModel>`.

Определение виртуального приложения IIS

В IIS с виртуальным приложением ассоциированы пул приложений и виртуальный каталог. Для WCF создается пул приложений ServiceHost, а в виртуальном каталоге хранятся файлы службы (SVC, config, .dll).

Создание SVC-файла

SVC-файл ссылается на реализацию службы. Создать такой файл можно в любом текстовом редакторе или в Visual Studio. В большинстве случаев класс реализации находится в какой-то DLL, которая и упоминается в SVC-файле. Эта DLL может располагаться в папке /bin виртуального каталога или в GAC. В листинге 1.10 показан SVC-файл, ссылающийся на откомпилированный класс .NET.

Листинг 1.10. SVC-файл, ссылающийся на откомпилированную службу

```
<%@ServiceHost Service="EssentialWCF.StockService" %>
```

Альтернативно SVC-файл может содержать собственно реализацию. В таком случае он будет длиннее, зато количество внешних зависимостей уменьшится. Поскольку исходный код находится там же, где сервер IIS, в котором размещена служба, сотрудники отдела эксплуатации или поддержки могут изменять его, не заходя для компиляции DLL в среду разработки. Плюсы и минусы такого подхода очевидны. К числу минусов следует отнести утрату контроля над интеллектуальной собственностью и невозможность управлять изменениями, так как код можно посмотреть и модифицировать на любом Web-сервере. Кроме того, снижается производительность. Плюс в том, что код прозрачен и ошибки можно быстро исправить, так как клиенты точно знают, что код делает и как его при необходимости изменить. В листинге 1.11 представлен SVC-файл, содержащий реализацию службы. Этот код будет компилироваться при первом обращении.

Листинг 1.11. SVC-файл, содержащий реализацию службы

```
<%@ServiceHost Language=c# Service="EssentialWCF.StockService" %>

using System;
using System.ServiceModel;

namespace EssentialWCF
{
   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      double GetPrice(string ticker);
   }

   public class StockService : IStockService
   {
      public double GetPrice(string ticker)
      {
         return 94.85;
      }
   }
}
```

Добавление секции <system.serviceModel> в файл web.config

Поскольку служба размещается в IIS, ее оконечные точки должны быть определены в конфигурационном файле, а не в коде. Конфигурационная информация хранится в секции `<system.serviceModel>` файла `web.config`. Как и при любом другом размещении, необходимо указать АПК оконечной точки: адрес, привязку и контракт. В листинге 1.12 приведен файл `web.config`, описывающий размещение службы в IIS. Отметим, что секция `<system.serviceModel>` ничем не отличается от приведенной в листинге 1.5.

Адрес службы определяется адресом виртуального каталога, в котором находится SVC-файл. Если у службы есть только одна оконечная точка, то адрес в ее привязке может быть пустым, тогда будет использоваться адрес самой службы. Если же оконечных точек несколько, то для каждой следует задать относительный адрес.

Стек каналов в привязке должен использовать HTTP в качестве транспортного протокола. В WCF уже встроены две таких привязки: `basicHttpBinding` и `wsHttpBinding`. Заказные привязки, в которых стек каналов составлен иначе, чем во встроенных, тоже поддерживаются при условии, что транспортом служит протокол HTTP. Заказные привязки рассматриваются в главах 3 и 4.

Контракт оконечной точки определяет класс, реализующий службу. Во время исполнения службе должен быть доступен его код, который может находиться в папке /bin, в GAC или непосредственно в SVC-файле.

Листинг 1.12. Определение службы в файле web.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockService"
                  behaviorConfiguration="MEXServiceTypeBehavior" >
            <endpoint address=""
                      binding="basicHttpBinding"
                      contract="EssentialWCF.IStockService " />
            <endpoint address="mex"
                      binding="mexHttpBinding"
                      contract="IMetadataExchange" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="MEXServiceTypeBehavior" >
               <serviceMetadata httpGetEnabled="true" />
            </behavior>
         </serviceBehaviors>
      </behaviors>
   </system.serviceModel>

</configuration>
```

Реализация WCF-клиента для ASMX-службы

WCF-клиенты могут обращаться к любым совместимым со стандартами службам вне зависимости от способа из размещения. Web-службы, которые создавались в версии .NET 1.1 Framework (ASMX), полностью отвечают стандартам. И, следовательно, согласно спецификации WS-I Basic Profile 1.1, к ним можно обращаться из WCF-клиентов.

Инструментальная поддержка

Как и при вызове WCF-службы, для создания прокси-класса и конфигурационного файла, необходимого для вызова ASMX-службы, можно воспользоваться операцией Add Service Reference (ASR) или утилитой `svcutil.exe`. После создания этих двух артефактов клиент может общаться с ASMX-службой – достаточно создать экземпляр прокси и вызывать его методы. Можно вместо этого сгенерировать прокси-класс и конфигурационный файл операцией Add Web Reference (AWR) или утилитой `wsdl.exe`, а затем действовать так же, как и выше.

При написании новых клиентских приложений, обращающихся к существующим ASMX-службам, лучше пользоваться ASR или `svcutil.exe`. Если уже имеется приложение, работающее с прокси-классами, созданными с помощью AWR или `wsdl.exe`, то лучше продолжать пользоваться этими инструментами. Тогда клиенту не потребуется два набора прокси и конфигурационных классов для работы с ASMX-службами. Если клиент доработан с расчетом на новые WCF-службы с привязкой `basicHttpBinding`, то все равно можно пользоваться AWR/wsdl.exe для генерации новых прокси-классов для WCF-служб.

Таблица 1.1. Варианты генерации прокси-классов и конфигурационных файлов

	ASMX-служба	WCF-служба
Модификация существующего клиента, который уже ссылается на ASMX-службу	Add Web Reference или `wsdl.exe`	Add Web Reference или `wsdl.exe`
Разработка новых клиентов для WCF-служб	Add Service Reference или `svcutil.exe`	Add Service Reference или `svcutil.exe`

Неважно, сгенерирован прокси-класс с помощью svcutil.exe или wsdl.exe, клиент сможет воспользоваться им для доступа к удаленной службе. Помимо этого, в файл app.config вносятся добавления, необходимые для того, чтобы клиентская программа могла поддержать прокси-класс.

Генерация прокси-класса и конфигурационного файла для клиента

Если вы модифицируете существующий клиент, для которого уже были созданы ASMX-прокси, то должны использовать Add Web Reference. В листинге 1.13 приведен код клиента, использующего для вызова операции службы прокси-класс, сгенерированный с помощью Add Web Reference.

Листинг 1.13. Код клиента, использующего для доступа к ASMX-службе прокси-класс, сгенерированный с помощью Add Web Reference

```
using System;
namespace Client
```

```
{
   class Program
   {
      static void Main(string[] args)
      {
         ASMXReference.StockService proxy =
                     new ASMXReference. StockService ())
         double p = proxy.GetPrice("msft");
         Console.WriteLine("Price:{0}", p);
         proxy.Close();
      }
   }
}
```

В листинге 1.14 показан конфигурационный файл, который генерирует операция Add Web Reference в Visual Studio. Обратите внимание, что в файле `app.config` хранится только адрес службы. Это составляет разительный контраст с уровнем детализации, который обеспечивает операция Add Service Reference (см. листинг 1.16). Дополнительные конфигурационные параметры позволяют разработчику или администратору менять, например, таймауты, не трогая исходный код.

Листинг 1.14. Конфигурационный файл, сгенерированный операцией Add Web Reference для ASMX-службы

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
      <sectionGroup name="applicationSettings"
         type="System.Configuration.ApplicationSettingsGroup,
                  System, Version=1.0.0.0, Culture=neutral,
                  PublicKeyToken=b77a5c561934e089" >
            <section name="Client.Properties.Settings"
               type="System.Configuration.ClientSettingsSection,
                        System, Version=1.0.0.0, Culture=neutral,
                        PublicKeyToken=b77a5c561934e089"
                        requirePermission="false" />
      </sectionGroup>
   </configSections>
   <applicationSettings>
      <Client.Properties.Settings>
         <setting name="Client_ASMXReference_StockService"
                              serializeAs="String">
            <value>http://localhost/asmx/service.asmx</value>
         </setting>
      </Client.Properties.Settings>
   </applicationSettings>
</configuration>
```

Если вы создаете новый клиент, для которого еще нет ASMX-прокси, то следует пользоваться операцией Add Service Reference, чтобы новый проект начал жизнь с новыми прокси-классами. В листинге 1.15 показан код клиента, в котором используется прокси, сгенерированный операцией Add Service Reference для ASMX-службы. Отметим, что при создании прокси необходимо задавать имя оконечной точки `StockServiceSoap`. Объясняется это тем, что Add Service Reference добавляет в файл `app.config` две оконечных точки: для одной указывается привязка `basicHttpBinding`, а для другой – специальная привязка, совместимая со спецификацией SOAP 1.1.

Листинг 1.15. Код клиента, использующего для доступа к ASMX-службе прокси-класс, сгенерированный с помощью Add Service Reference

```
using System;
namespace Client
{
   class Program
   {
      static void Main(string[] args)
      {
         using (WCFReference.StockServiceSoapClient proxy =
                  new client.WCFReference.StockServiceSoapClient
                     ("StockServiceSoap"))
         {
            double p = proxy.GetPrice("msft");
            Console.WriteLine("Price:{0}", p);
         }
      }
   }
}
```

В листинге 1.16 представлен конфигурационный файл, сгенерированный в Visual Studio с помощью операции Add Service Reference для ASMX-службы. Обратите внимание, насколько полная информация о привязке и оконечной точке получена от ASMX-службы и сохранена в файле `app.config`. Также заметьте, что определены две оконечные точки. Для первой – `StockServiceSoap` – задана привязка `basicHttpBinding`, совместимая со спецификацией WS-I Basic Profile 1.1. Для второй – `StockServiceSoap12` – указана специальная привязка, необходимая для взаимодействия по более поздней версии протокола SOAP. Поскольку ASMX-службы совместимы с WS-I Basic Profile 1.1, то реально используется первая оконечная точка.

Листинг 1.16. Файл app.config, сгенерированный операцией Add Service Reference для ASMX-службы

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <basicHttpBinding>
            <binding name="StockServiceSoap" closeTimeout="00:01:00"
                     openTimeout="00:01:00" receiveTimeout="00:10:00"
                     sendTimeout="00:01:00" allowCookies="false"
                     bypassProxyOnLocal="false"
                     hostNameComparisonMode="StrongWildcard"
                     maxBufferSize="65536" maxBufferPoolSize="524288"
                     maxReceivedMessageSize="65536"
```

```
                    messageEncoding="Text" textEncoding="utf-8"
                    transferMode="Buffered"
                    useDefaultWebProxy="true">
                       <readerQuotas maxDepth="32"
                          maxStringContentLength="8192" maxArrayLength="16384"
                          maxBytesPerRead="4096"
                          maxNameTableCharCount="16384" />
                       <security mode="None">
                          <transport clientCredentialType="None"
                                proxyCredentialType="None"
                                realm="" />
                          <message clientCredentialType="UserName"
                                algorithmSuite="Default" />
                       </security>
                 </binding>
              </basicHttpBinding>
              <customBinding>
                 <binding name="StockServiceSoap12">
                    <textMessageEncoding maxReadPoolSize="64"
                          maxWritePoolSize="16"
                          messageVersion="Soap12" writeEncoding="utf-8">
                          <readerQuotas maxDepth="32"
                                maxStringContentLength="8192"
                                maxArrayLength="16384"
                                maxBytesPerRead="4096"
                                maxNameTableCharCount="16384" />
                    </textMessageEncoding>
                    <httpTransport manualAddressing="false"
                          maxBufferPoolSize="524288"
                          maxReceivedMessageSize="65536"
                          allowCookies="false"
                          authenticationScheme="Anonymous"
                          bypassProxyOnLocal="false"
                          hostNameComparisonMode="StrongWildcard"
                          keepAliveEnabled="true" maxBufferSize="65536"
                          proxyAuthenticationScheme="Anonymous"
                          realm="" transferMode="Buffered"
                          unsafeConnectionNtlmAuthentication="false"
                          useDefaultWebProxy="true" />
                 </binding>
              </customBinding>
           </bindings>
           <client>
              <endpoint address="http://localhost/asmx/service.asmx"
                          binding="basicHttpBinding"
                          bindingConfiguration="StockServiceSoap"
                          contract="Client.WCFReference.StockServiceSoap"
                          name="StockServiceSoap" />
              <endpoint address="http://localhost/asmx/service.asmx"
                          binding="customBinding"
                          bindingConfiguration="StockServiceSoap12"
                          contract="Client.WCFReference.StockServiceSoap"
                          name="StockServiceSoap12" />
           </client>
        </system.serviceModel>
</configuration>
```


Резюме

В этой главе мы рассмотрели основы WCF, обозначаемые аббревиатурой АПК. Служба состоит из оконечных точек, у каждой из которых есть АПК: адрес, привязка и контракт. Со службой может быть ассоциировано поведение, описывающее ее семантику, например, многопоточность и параллелизм, но об этом мы поговорим в последующих главах.

Службу можно размещать в любом процессе операционной системы от консольного приложения, находящегося на рабочем столе Windows, до сервера IIS, работающего в составе Web-фермы. Мы привели различные примеры размещения. В качестве владельца WCF-служб чаще всего выбирается IIS. Если на компьютере, где работает IIS, установлена версия .NET 3.5, то все запросы к SVC-ресурсам направляются WCF. SVC-файл содержит ссылку на реализацию службы. Реализация может находиться либо в DLL в папке /bin виртуального каталога IIS, либо в DLL, загружаемой из глобального кэша сборок (GAC), либо непосредственно в SVC-файле.

Клиенты общаются со службами только с помощью сообщений. Для удобства разработчиков в Visual Studio имеются инструменты для автоматического создания клиентских прокси-классов, представляющих операции сервера. Клиентское приложение пользуется этими прокси-классами для взаимодействия со службой. Прокси-класс WCF сериализует параметры в виде XML и отправляет XML-сообщение на адрес оконечной точке службы. Конфигурационные данные, необходимые прокси-классу, хранятся в файле `app.config` на стороне клиента. И прокси-класс, и конфигурационный файл генерируются утилитой `svcutil.exe` или операцией Add Service Reference в Visual Studio. Хотя эти инструменты могут резко повысить производительность труда, бывают случаи, когда приходится программировать на уровне WCF API. Это тоже возможно.

ASMX-службы совместимы со спецификацией WS-I Basic Profile 1.1. С ней же совместима поставляемая в комплекте с WCF привязка `basicHttpBinding`, следовательно, с ее помощью WCF-клиенты могут обращаться к ASMX-службам.

Располагая информацией, почерпнутой в этой главе, вы уже можете определять, размещать и потреблять WCF-службы.

Глава 2. Контракты

В материальном мире *контракт* – это обязывающее соглашение между двумя или более сторонами о поставке товаров или услуг по определенной цене. В мире битов и служб функция контракта аналогична: это соглашение между двумя или более сторонами о формате сообщений, которыми они обмениваются.

Контракт представляет собой описание сообщений, передаваемых оконечным точкам службы и возвращаемых ей. Каждая оконечная точка определяется своими АПК: адресуемым местом в сети, куда посылаются сообщения; привязкой, описывающей способ передачи сообщений, и контрактом, в котором оговорены форматы сообщений. Напомним, что служба – это на самом деле набор оконечных точек, которые реализуют те или иные программно закодированные алгоритмы. Это может быть бизнес-функция высокого уровня, например ввод заказа в систему, или более специализированная функция, как, скажем, поиск адреса клиента. Для высокоуровневых функций обычно требуются сложные структуры данных, тогда как для более простых часто хватает базовых типов. В любом случае оконечная точка должна специфицировать, какие операции она может выполнять, и формат ожидаемых данных. В совокупности эти спецификации и составляют контракт.

В WCF есть контракты трех видов:

- **Контракт о службе** описывает функциональные операции, реализуемые службой. Он отображает методы класса .NET на описания служб, типов портов и операций на языке WSDL. Внутри контракта о службе имеются контракты об операциях, которые описывают отдельные операции службы, то есть методы, реализующие ее функции.
- **Контракт о данных** описывает структуры данных, используемые службой для взаимодействия с клиентами. Контракт о данных отображает типы CLR на определения на языке XML Schema Definitions (XSD) и определяет, как их следует сериализовывать и десериализовывать. Он описывает все данные, получаемые и отправляемые операциями службы.
- **Контракт о сообщениях** отображает типы CLR на сообщения протокола SOAP и описывает формат последних, что находит отражение в определениях сообщений на языках WSDL и XSD. Контракт о сообщениях позволяет точно контролировать состав заголовков и тел SOAP-сообщений.

Чтобы контракты были интероперабельны с максимально широким диапазоном систем, они выражаются на языке WSDL (Web Service Description Language – язык описания Web-служб). Поэтому, прежде чем углубляться в обсуждение контрактов, будет уместно дать краткий обзор WSDL. Консорциум W3C – организация, устанавливающая стандарты для веб (в нее входят в частности компании Microsoft, IBM и другие), – специфицирует WSDL следующим образом:

> *WSDL – это формат XML для описания сетевых служб, представленных в виде набора оконечных точек, выполняющих операции над сообщениями, которые содержат документо-ориентированную или процедурно-ориентированную информацию. Для того чтобы определить оконечную точку, операции и сообщения описываются абстрактно, а затем привязываются к конкретному сетевому протоколу и формату сообщений. Взаимосвязанные конкретные оконечные точки объединяются в абстрактные оконечные точки (службы). Язык WSDL расширяем, что позволяет описывать оконечные точки и ассоциированные с ними сообщения независимо от того, какие используются форматы сообщений и сетевые протоколы; однако в этом документе описывается только использование WSDL в сочетании с протоколами SOAP 1.1, HTTP GET/POST и MIME.*

В полной спецификации, размещенной по адресу www.w3.org/TR/wsdl, описываются базовые понятия и дополнительные детали, на основе которых поставщики и, в частности, корпорация Microsoft могут создавать инструменты для создания и потребления WSDL-документов. Основные элементы WSDL описаны в таблице 2.1, где суть официальной спецификации изложена более понятным языком.

Таблица 2.1. Элементы WSDL

Элемент WSDL	Описание
Тип	Определения типов данных, применяемые для описания передаваемых сообщений. Обычно выражаются на языке XML Schema Definition (XSD)
Сообщение	Представляет собой абстрактное определение передаваемых данных. Сообщение состоит из логических частей, каждая из которых ассоциирована с определением в некоторой системе типов. Сообщение аналогично формальному параметру функции или метода интерфейса и используется для определения сигнатур операций
Операция	Имя и описание действия, поддерживаемого службой. С помощью операций раскрываются функциональные возможности оконечной точки службы
Тип порта (PortType)	Именованный набор абстрактных операций и абстрактных сообщений. Оконечная точка службы реализует некий тип порта, группирующий взаимосвязанные операции
Привязка	Определяет детали формата сообщений и протокола для операций и сообщений, определенных конкретным типом порта
Порт	Определяет отдельную оконечную точку путем задания одиночного адреса для привязки
Служба	Определяет набор взаимосвязанных портов

Поскольку контракты описываются на языках WSDL и XSD, а программа обычно работает с типами CLR, возникает необходимость отобразить одну систе-

му типов на другую. В WCF эта задача решается в три этапа. Сначала при написании кода службы вы снабжаете класс определенными в WCF атрибутами `[ServiceContract]`, `[OperationContract]`, `[FaultContract]`, `[MessageContract]` и `[DataContract]`. Затем при написании клиентского кода вы запрашиваете у службы детали контракта. Это делается с помощью Visual Studio или утилиты `svcutil.exe`, которая вызывает инфраструктурную оконечную точку службы, возвращающую метаданные, необходимые для генерации WSDL-документа, получая их от атрибутов. Наконец, на этапе исполнения, когда клиент вызывает какой-то метод, определенный в интерфейсе службы, WCF сериализует типы CLR и вызов метода в формат XML и посылает сообщение в сеть в соответствии с привязкой и схемой кодирования, согласованным посредством WSDL.

В этом процессе участвуют четыре конструкции: две со стороны .NET и две со стороны XML. Со стороны .NET имеется *mun CLR*, который определяет структуры данных и функциональные возможности, но что-то сделать он может лишь после того, как будет создан *объект* этого типа. Со стороны XML мы имеем *XSD-описание* структуры данных, но сообщение начинает существовать лишь после того, как будет создан *экземпляр XML* (XML Instance).

Чтобы разобраться в том, как работает WCF, вы должны понимать обе ипостаси: код и WSDL-описание. К счастью, в комплект поставки WCF входят два инструмента для отображения одного на другое. Первый – это утилита `SvcUtil.exe`, которую можно вызывать явно из командной строки или неявно при выполнении операции Add Service Reference в Visual Studio. Эта утилита, для которой можно задавать много разных флагов, порождает WSDL-документ и прокси-классы, облегчающие отображение между типами .NET и XSD, а также между методами классов .NET и операциями WSDL. Второй – Service Trace Viewer, или SvcTraceViewer.exe – это графический инструмент, который читает и интерпретирует диагностические файлы протоколов, формируемых WCF. С его помощью вы можете видеть форматы сообщений, получаемых и отправляемых оконечными точками, и трассировать весь поток сообщений. Подробно эта программа описана в главе 9 «Диагностика».

В этой главе мы опишем, как пользоваться четырьмя из пяти видов контрактов. Начнем с контрактов о службах, в которых описаны оконечные точки, и контрактов об операциях, которые определяют методы. Затем рассмотрим контракты о данных, с помощью которых описываются данные, получаемые и отправляемые оконечными точками. И наконец мы изучим контракты о сообщениях, позволяющие более точно контролировать структуру SOAP-сообщений. Позже, в главе 10 «Обработка исключений» мы обсудим контракты об отказах.

Контракт о службе определяет интерфейс операций, реализованных оконечной точкой службы. Он ссылается на форматы сообщений и описывает порядок обмена сообщениями. Сами форматы сообщений описываются контрактами о данных и сообщениях. В этом разделе мы рассмотрим различные способы обмена сообщениями, которые могут быть реализованы в контракте о службе.

Контракты о службах встречаются в WCF как на этапе проектирования, так и на этапе выполнения. На этапе проектирования они идентифицируют те классы

программы, которые необходимо раскрыть в виде оконечных точек в WSDL-документе. Класс, помеченный атрибутом `[ServiceContract]`, и его методы, помеченные атрибутом `[OperationContract]`, включаются в WSDL-документ и, стало быть, доступны клиентам. Такой класс обозначается как `wsdl:service`, а соответствующие операции – как `wsdl:operation`. На этапе выполнения, когда диспетчер WCF получает сообщение, он ищет в нем имя `wsdl:operation`, чтобы определить, какой из методов класса, помеченных атрибутом `[OperationContract]`, должен получить десериализованное сообщение. На рис. 2.1 приведена высокоуровневая диаграмма трансляции кода на язык WSDL.

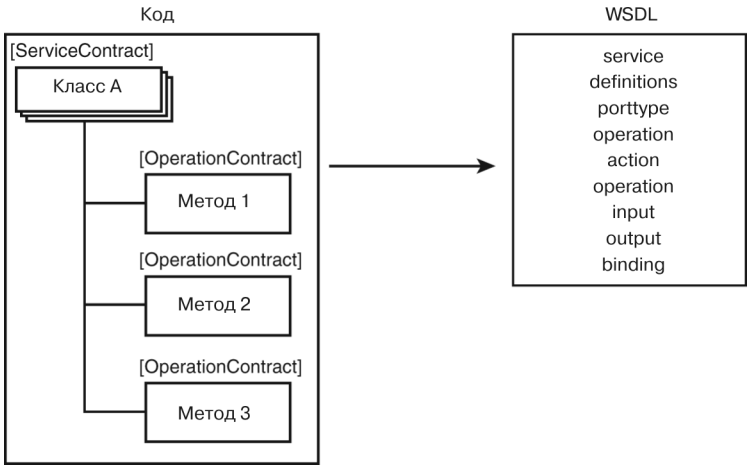

Рис. 2.1. Высокоуровневая диаграмма трансляции артефактов кода на язык WSDL

На рис. 2.2 изображена та же диаграмма, что на рис. 2.1, но для ясности показаны также конструкции на языке C# и соответствующие им элементы WSDL.

Синхронные операции запрос-ответ

Обмен сообщениями в виде синхронных операций запрос-ответ – это наиболее распространенный способ функционирования службы. Этот паттерн знаком всем, кто программировал на процедурных или объектно-ориентированных языках. Его прототипом является вызов локальной процедуры, но употребляется он и при вызовах удаленных процедур. На рис. 2.3 показана диаграмма взаимодействия типа запрос-ответ, где прокси-класс, работающий на стороне клиента, посылает запрос службе, а та синхронно отвечает.

WCF позволяет очень просто организовать такой обмен сообщениями между клиентом и службой. На этапе проектирования вы с помощью операции Add Service Reference или утилиты `svcutil.exe` обращаетесь к оконечной точке MEX (Metadata Exchange) службы и генерируете клиентский прокси-класс с такой же

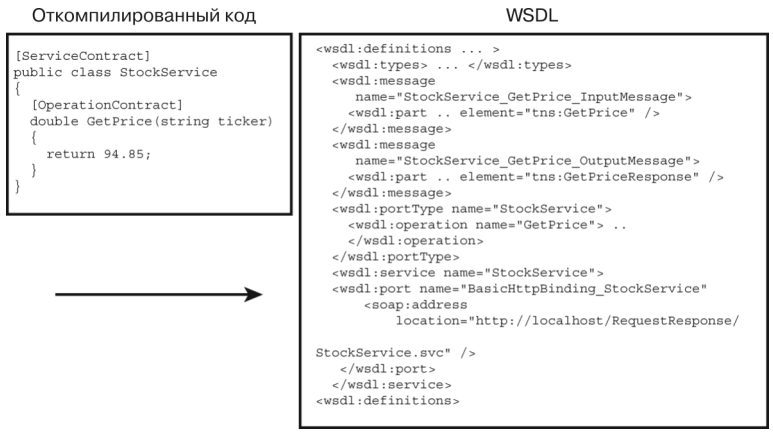

Рис. 2.2. Высокоуровневая диаграмма трансляции синтаксических конструкций на язык WSDL

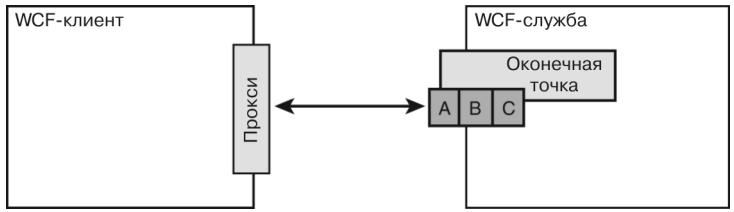

Рис. 2.3. Синхронный обмен сообщениями типа запрос-ответ

сигнатурой, как у операций службы. Это позволяет клиенту вызывать методы прокси, как локальные функции. Прокси сериализует имя и параметры метода, представляя их в виде SOAP-сообщения, отправляет это сообщение службе, дожидается ответного сообщения и создает объект типа .NET, представляющий полученный ответ.

В листинге 2.1 показано определение контракта о службе. В данном случае определен один контракт о службе и один контракт об операции. Последний описывает метод, который может вызывать клиент, или, если быть более точным, сообщение, которое может послать клиент, чтобы оно было понято службой. Отметим, что контракт определен в применении к интерфейсу, а не к определению класса.

Листинг 2.1. Служба типа запрос-ответ

```
using System;
using System.ServiceModel;

namespace EssentialWCF
{
```



```
   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      double GetPrice(string ticker);
   }

   public class StockService : IStockService
   {
      public double GetPrice(string ticker)
      {
         return 94.85;
      }
   }
}
```

В листинге 2.2 показан код клиента, в котором используется прокси-класс, сгенерированный операцией Add Service Reference для обращения к службе из листинга 2.1. Он аналогичен коду, представленному в листинге 1.2.

Листинг 2.2. Клиент службы типа запрос-ответ

```
using System;
using System.ServiceModel;

namespace Client
{
   class client
   {
      static void Main(string[] args)
      {
         localhost.StockServiceClient proxy =
                        new localhost.StockServiceClient();
         double price = proxy.GetPrice("msft");
         Console.WriteLine("msft:{0}", price);
         proxy.Close();
      }
   }
}
```

В листинге 2.3 показано SOAP-сообщение, которое клиент посылает оконечной точке службы. Стоит отметить следующие моменты:

- в SOAP-сообщении указано пространство имен http://tempuri.org/, оно задается по умолчанию, если не переопределено с помощью атрибута `[ServiceContract]`. Если служба должна быть доступна вне приложения или сравнительно небольшой организации, то имя пространства имен следует изменить, поскольку смысл его в том, чтобы уникально идентифицировать вашу службу и исключить неоднозначность при комбинировании нескольких служб;
- имя `GetPrice` метода в определении класса в листинге 1.1 используется в заголовке SOAP для формирования значения тега Action. Полное значение действия образуется путем объединения пространства имен контрак-

та, имени контракта (имени интерфейса или класса, если явно не указан интерфейс), имени операции и дополнительной строки (Response), если сообщение – это ответ на запрос;

- тело SOAP-сообщения определяется сигнатурой метода и модификаторами, заданными в атрибутах `[OperationContract]` и `[DataContract]`;
- заголовок SOAP-сообщения включает адрес, на который посылается сообщение. В данном случае это SVC-файл, размещенный на машине, где работает IIS.

Листинг 2.3. SOAP-сообщение, посылаемое в диалоге типа запрос-ответ

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Header>
      <To s:mustUnderstand="1"
         xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">
         http://localhost/RequestResponseService/StockService.svc
      </To>
      <Action s:mustUnderstand="1"
            xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">
            http://tempuri.org/StockService/GetPrice
      </Action>
   </s:Header>
   <s:Body>
      <GetPrice xmlns="http://tempuri.org/">
            <ticker>msft</ticker>
      </GetPrice>
   </s:Body>
</s:Envelope>
```

Асинхронные операции запрос-ответ

В хорошей программе стараются избегать ситуаций, когда пользователь должен дождаться завершения одного задания, прежде чем начать выполнение следующего. Например, когда почтовый клиент загружает с сервера новые сообщения, вы можете читать и удалять те сообщения, которые уже получены. Другой пример: пока Web-браузер загружает изображения, встречающиеся на странице, вы можете ее прокручивать или даже уйти на другую страницу. Такая форма многозадачности достигается за счет использования асинхронной работы.

В WCF операции службы типа запрос-ответ приводят к блокировке клиента в течение всего времени выполнения операции. Если спуститься на уровень ниже, то окажется, что в прокси-классе, сгенерированном `svcutil.exe`, применяются блокирующие обращения к стеку каналов, отвечающему за взаимодействие со службой. В результате клиентское приложение блокируется на все время вызова службы. Если для получения ответа службе требуется десять секунд, то все это время клиент будет ждать, ничего не делая.

К счастью, в каркасе .NET Framework еще со времен .NET 1.0 имеются средства для реализации асинхронного поведения клиента, которые позволяют асинхронно вызывать любой синхронный метод. Достигается это с помощью класса `IAsyncResult` и создания двух методов `Begin`*`OperationName`* и `End`*`OperationName`*. Сначала клиент вызывает метод `Begin`*`OperationName`*, а затем продолжает выполнение в текущем потоке, пока асинхронно вызванная операция выполняется в другом потоке. Позже для каждого вызова `Begin`*`OperationName`* клиент должен вызвать метод `End`*`OperationName`*, чтобы получить результат работы операции. Методу `Begin`*`OperationName`* клиент передает делегат, который будет вызван при завершении асинхронной операции и может сохранять информацию, полученную при вызове `Begin`*`OperationName`*.

Вы можете потребовать, чтобы Add Service Reference генерировала асинхронные методы. Для этого следует нажать кнопку Advanced (Дополнительно) в диалоговом окне Add Service Reference и отметить флажок Generate Asynchronous Operations (Генерировать асинхронные операции). На рис. 2.4 показано диалоговое окно Service Reference Settings (Параметры ссылки на службу). Утилита `SvcUtil` также может для каждой операции службы создавать методы `Begin<operation>` и `End<operation>` в дополнение к синхронным, если задать при ее вызове флаг `/async`.

На рис. 2.5 показана диаграмма асинхронного взаимодействия в .NET Framework, когда используется прокси-класс, сгенерированный `SvcUtil`. Отметим, что

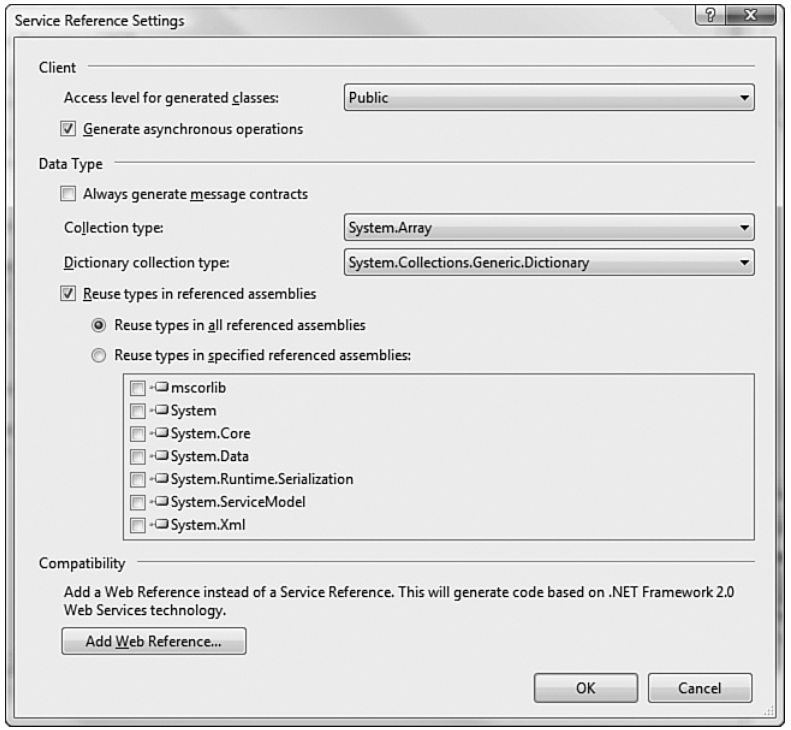

Рис. 2.4. Задание режима генерации асинхронных методов в диалоговом окне Service Reference Settings

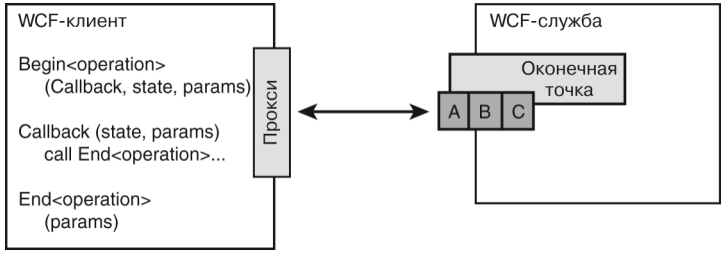

Рис. 2.5. Асинхронное взаимодействие типа запрос-ответ

служба ничего не знает о том, что клиент вызывает ее асинхронно; в контракте о службе говорится лишь о взаимодействии типа запрос-ответ, а клиент реализует асинхронный паттерн без какой бы то ни было помощи со стороны службы.

В листинге 2.4 демонстрируется использование методов `BeginGetPrice` и `EndGetPrice` в сочетании с `IAsyncResult` для сохранения информации о состоянии операции службы. Метод `BeginGetPrice` принимает два параметра, помимо строки, определенной в качестве аргумента операции службы. Первый параметр, делегат типа `AsyncCallback`, – это локальный метод с одним аргументом типа `AsyncResult`. Второй параметр может быть любым объектом и служит для передачи состояния от инициирующей программы делегату `AsyncCallback`. Он передается `AsyncCallback` в свойстве `AsyncResult.AsyncState` в момент завершения операции. Полезно передавать объект прокси-класса, начавший взаимодействие со службой, чтобы `AsyncCallback` мог вызвать метод `EndGetPrice` и получить ответ от операции службы. Статическая переменная `c` нужна для того, чтобы предотвратить завершение клиента до получения ответа от службы, а класс `Interlocked` гарантирует безопасность относительно потоков при выполнении программы на многопроцессорных машинах.

Листинг 2.4. Клиент, участвующий в диалоге типа запрос-ответ с применением паттерна асинхронного программирования в .NET

```
using System;
using System.Threading;

namespace AW.EssentialWCF.Samples
{
   class Program
   {
      static int c = 0;
      static void Main(string[] args)
      {
         StockServiceClient proxy = new StockServiceClient();
         IAsyncResult arGetPrice;
         for (int i = 0; i < 10; i++)
         {
            arGetPrice = proxy.BeginGetPrice("msft",
                                 GetPriceCallback, proxy);
            Interlocked.Increment(ref c);
         }

         while (c > 0)
         {
            Thread.Sleep(1000);
            Console.WriteLine("Ожидание... Незавершенных вызовов:{0}", c);
         }
         proxy.Close();
         Console.WriteLine("Готово!");
      }

      // Асинхронный обратный вызов для вывода результата.
      static void GetPriceCallback(IAsyncResult ar)
      {
         double d = ((StockServiceClient)ar.AsyncState).EndGetPrice(ar);
         Interlocked.Decrement(ref c);
      }
   }
}
```


Односторонние операции

Односторонний обмен сообщениями полезен, когда клиенту нужно просто послать информацию службе, а ответ его не интересует, необходимо лишь получить подтверждение успешной доставки. Иногда односторонний обмен ошибочно называют «послал и забыл». На самом деле следует говорить «послал и получил подтверждение», поскольку отправляющая сторона все же получает подтверждение о том, что сообщение было успешно помещено в коммуникационный канал.

WCF поддерживает односторонний обмен сообщениями на уровне операции службы. Иначе говоря, операцию можно пометить как одностороннюю, и инфраструктура позаботится об оптимизации для этого случая. Если клиент вызывает односторонний метод службы или, точнее, отправляет сообщение оконечной точке, чья операция помечена как односторонняя, то управление возвращается вызывающей программе еще до завершения операции службы. Односторонние операции помечаются модификатором `IsOneWay=true` в атрибуте `[OperationContract]`. В листинге 2.5 показан контракт о службе, у которой есть две операции. Когда клиент обращается к операции `DoBigAnalysisFast`, прокси возвращает управление немедленно, а не ждет десять секунд, которые служба проводит в методе `Thread.Sleep`. Если же клиент обращается к операции `DoBigAnalysisSlow`, то прокси блокируется на эти десять секунд.

Отметим, что, как и при других способах обмена сообщениями, программа ничего не знает о том, какая привязка или коммуникационный протокол применены для доставки сообщения. Поскольку привязка `netTcpBinding` поддерживает двустороннюю связь, а `basicHttpBinding` – диалог запрос-ответ, любую из них можно использовать для одностороннего обмена.

Листинг 2.5. Контракт об односторонней операции

```
[ServiceContract]
public interface IStockService
```

```
{
   [OperationContract(IsOneWay = true)]
   void DoBigAnalysisFast(string ticker);

   [OperationContract]
   void DoBigAnalysisSlow(string ticker);
}

public class StockService : IStockService
{
   public void DoBigAnalysisFast(string ticker)
   {
      Thread.Sleep(10000);
   }
   public void DoBigAnalysisSlow(string ticker)
   {
      Thread.Sleep(10000);
   }
}
```

Дуплексные операции

Чаще всего диалог между клиентом и службой строится по принципу запрос-ответ. Клиент начинает сеанс связи, посылает службе сообщение-запрос, после чего служба посылает ответ клиенту. Если ожидается, что ответ придет быстро, можно организовать синхронный диалог, тогда клиент будет блокирован, пока ответ не придет. Если же ожидается задержка, то диалог запрос-ответ можно реализовать асинхронно, применяя на стороне клиента стандартные механизмы .NET. В таком случае WCF возвращает управление клиентскому приложению сразу после отправки запроса службе. Когда от службы будет получен ответ, .NET вызовет процедуру обратного вызова, чтобы завершить обмен сообщениями.

Однако, что если службе самой необходимо отправить сообщение, например, оповещение или предупреждение? Что если клиент и служба должны обмениваться информацией на более высоком уровне, чем индивидуальные сообщения, то есть на несколько запросов со стороны клиента поступает лишь один ответ от службы? Что если ожидаемое время обработки запроса составляет порядка десяти минут? WCF поддерживает двусторонние коммуникации с помощью *дуплексных* контрактов о службах. Дуплексный контракт реализует паттерн дуплексного обмена сообщениями, когда любая сторона может инициировать отправку сообщения, не дожидаясь запроса, коль скоро установлен канал связи. По дуплексному каналу можно как вести диалог запрос-ответ, так и посылать односторонние сообщения.

Поскольку сообщения могут передаваться в любом направлении – от клиента службе или от службы клиенту, – обеим сторонам необходимы адрес, привязка и контракт, определяющие куда, как и какие сообщения следует посылать. Чтобы упростить передачу сообщений от службы клиенту, WCF может создать дополнительный канал. Если исходный канал не способен поддержать двустороннюю

связь, то WCF создаст второй канал, применяя тот же протокол, что задан для оконечной точки службы. Это изображено на рис. 2.6.

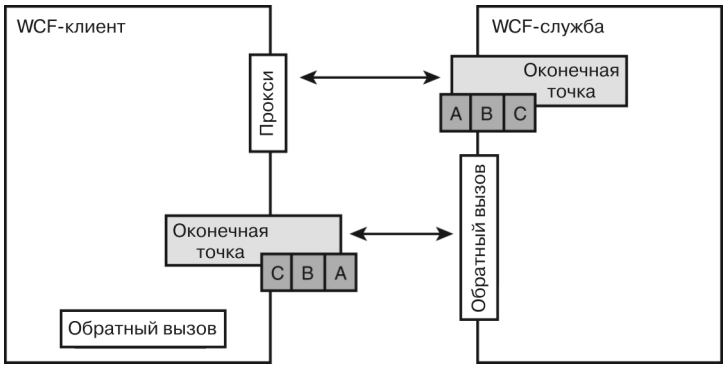

Рис. 2.6. Дуплексная коммуникация

В зависимости от того, какая привязка использовалась при установлении сеанса в направлении от клиента к службе, WCF создает один или два канала для реализации дуплексного обмена. Если протокол поддерживает двустороннюю связь, как в случае именованных каналов (named pipe) или TCP, то достаточно одного канала. Если же двусторонняя связь не поддерживается протоколом (например, http), то WCF создает дополнительный канал в направлении от службы к клиенту. Те из готовых привязок WCF, в названии которых есть слово *dual* (например, `wsDualHttpBinding`), реализуют два канала. Заказные привязки, в которых для достижения конкретной цели комбинируются различные каналы, тоже могут реализовать сдвоенный канал, включив в стек каналов элемент `compositeDuplex`. Подробно заказные привязки рассматриваются в главе 4.

При отправке сообщения службе клиент пользуется адресом, который указан для оконечной точки службы. При отправке же сообщения в обратном направлении по составному дуплексному каналу служба должна знать адрес оконечной точки клиента. Этот адрес автоматически генерируется каналом WCF, но может быть переопределен путем задания атрибута `BaseAddress` элемента привязки `compositeDuplex`.

Сравнение парного одностороннего и дуплексного обмена

Решить задачу о двустороннем обмене сообщениями можно двумя способами: определить либо пару односторонних контрактов, либо один дуплексный. В первом случае клиент и служба – независимые владельцы WCF. Каждый из них объявляет свои оконечные точки, в которые противоположная сторона может посылать сообщения. Поскольку обе стороны являются полноценными службами,

то они могут объявить несколько оконечных точек, пользоваться несколькими привязками и вести независимый учет версий своих контрактов. В случае дуплексного контракта клиент не становится явной WCF-службой и не может свободно выбирать привязки или объявлять свои оконечные точки. Адрес, привязка и контракт, определяющие клиентскую оконечную точку, назначаются фабрикой каналов в момент инициирования дуплексного сеанса связи клиентом.

В таблице 2.2 парные односторонние контрактов сравниваются с дуплексными.

Таблица 2.2. Сравнение парных односторонних и дуплексных контрактов для организации двусторонней коммуникации

Парные односторонние контракты	Дуплексный контракт
Версии контрактов можно изменять независимо. Поскольку клиент является полноценной службой, он может объявлять оконечные точки и менять версию контракта независимо от службы	Обратный контракт клиента определяется службой. Если служба изменяет свой контракт, то может оказаться необходимо изменить клиента. Тем самым единственным потребителем сервисных возможностей клиента становится служба, определяющая обратный контракт
Каждый односторонний контракт определяет свою привязку, поэтому в каждом направлении можно использовать различные протоколы, схемы кодирования и шифрования	Коммуникационный протокол одинаков в обоих направлениях, поскольку определяется привязкой службы

Реализация серверной части дуплексного контракта о службе

Дуплексный контракт содержит спецификации интерфейсов для оконечных точек службы и клиента. В этом случае часть контракта службы реализуется на стороне клиента.

В листинге 2.6 определен контракт о службе, предоставляющей сведения об изменении котировок акций. В нем используется дуплексная коммуникация, позволяющая клиенту зарегистрироваться для получения обновлений, которые служба будет периодически ему посылать. Клиент инициирует сеанс связи, вызывая операцию службы `RegisterForUpdates`. Затем служба создает поток, который периодически отправляет обновления клиенту, вызывая его операцию `PriceUpdate`.

Листинг 2.6. Дуплексный контракт о службе: реализация серверной части

```
[ServiceContract(CallbackContract = typeof(IClientCallback))]
public interface IServerStock
{
   [OperationContract (IsOneWay=true)]
   void RegisterForUpdates(string ticker);
}

public interface IClientCallback
{
   [OperationContract(IsOneWay = true)]
   void PriceUpdate(string ticker, double price);
}

public class ServerStock : IServerStock
{
   // Это ПЛОХОЙ алгоритм оповещения, поскольку для каждого клиента
   // создается отдельный поток. Следовало бы вместо этого создавать
   // по одному потоку на каждый тикер.
   public void RegisterForUpdates(string ticker)
   {
      Update bgWorker = new Update();
      bgWorker.callback =
               OperationContext.Current.
                                    GetCallbackChannel<IClientCallback>();
      Thread t = new
            Thread(new ThreadStart(bgWorker.SendUpdateToClient));
      t.IsBackground = true;
      t.Start();
   }
}

public class Update
{
   public IClientCallback callback = null;
   public void SendUpdateToClient()
   {
      Random p = new Random();
      for (int i=0;i<10;i++)
      {
         Thread.Sleep(5000); // откуда-то получаем обновления
         try
         {
            callback.PriceUpdate("msft", 100.00+p.NextDouble());
         }
         catch (Exception ex)
         {
            Console.WriteLine("Ошибка при отправке кэшированного значения
клиенту: {0}",
                                    ex.Message);
         }
      }
   }
}
```

И для полноты картины в листинге 2.7 приведен соответствующий конфигурационный файл. Обратите внимание на использование спаренной (dual) привязки.

Листинг 2.7. Дуплексный контракт о службе: конфигурация на стороне сервера

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
```

```
      <services>
         <service behaviorConfiguration="MEXServiceTypeBehavior"
                  name="EssentialWCF.StockService">
            <endpoint address="" binding="wsDualHttpBinding"
                  contract="EssentialWCF.IStockService" />
            <endpoint address="mex" binding="mexHttpBinding"
                  contract="IMetadataExchange" />
         </service>
      </services>
      <behaviors>
         <serviceBehaviors>
            <behavior name="MEXServiceTypeBehavior" >
               <serviceMetadata httpGetEnabled="true" />
            </behavior>
         </serviceBehaviors>
      </behaviors>
   </system.serviceModel>
</configuration>
```

Программа в листинге 2.6 имеет недостаток: она создает по одному потоку для каждого клиента. В данном случае оценить заранее количество клиентов невозможно (их могут быть миллионы), зато поддается оценке количество тикеров (тысячи). Следовательно, лучше создавать отдельный поток для каждого тикера, а не клиента.

В листинге 2.8 приведен альтернативный алгоритм. Здесь таблица `hashtable` используется для отслеживания тикеров ценных бумаг, для которых клиенты затребовали обновления. В таблице хранятся экземпляры класса `Update`, причем каждый экземпляр работает в собственном потоке. Список обратных вызовов клиентов хранится в локальной памяти потока в классе `Update`, что позволяет ему уведомить всех клиентов об изменении своего тикера. Отметим, что при доступе к списку клиентов выставляется блокировка `lock` как в методе `RegisterForUpates` главного класса `StockService`, так и в самом классе `Update`. Это необходимо для того, чтобы класс `StockService` не обновлял набор, когда класс `Update` его обходит.

Листинг 2.8. Дуплексный контракт о службе: реализация серверной части (улучшенное использование потоков)

```
public class StockService : IStockService
{
   public class Worker
   {
      public string ticker;
      public Update workerProcess;
   }
   public static Hashtable workers = new Hashtable();

   public void RegisterForUpdates(string ticker)
   {
      Worker w = null;

      // при необходимости создаем новый рабочий объект, добавляем его
```



```
      // в хэш-таблицу и запускаем в отдельном потоке
      if (!workers.ContainsKey(ticker))
      {
         w = new Worker();
         w.ticker = ticker;
         w.workerProcess = new Update();
         w.workerProcess.ticker = ticker;
         workers[ticker] = w;

         Thread t = new Thread(new
                           ThreadStart(w.workerProcess.SendUpdateToClient));
         t.IsBackground = true;
         t.Start();
      }

      // получить рабочий объект для данного тикера и добавить
      // прокси клиента в список обратных вызовов
      w = (Worker)workers[ticker];
      IClientCallback c =
                        OperationContext.Current.
                                    GetCallbackChannel<IClientCallback>();
      lock (w.workerProcess.callbacks)
      w.workerProcess.callbacks.Add(c);
   }
}

public class Update
{
   public string ticker;
   public List<IClientCallback> callbacks =
                                    new List<IClientCallback>();
   public void SendUpdateToClient()
   {
      Random w = new Random();
      Random p = new Random();
      while(true)
      {
         Thread.Sleep(w.Next(5000)); // откуда-то получаем обновления
         lock (callbacks)
         foreach (IClientCallback c in callbacks)
         try
         {
            c.PriceUpdate(ticker, 100.00+p.NextDouble()*10);
         }
         catch (Exception ex)
         {
            Console.WriteLine("Ошибка при отправке кэшированного значения
клиенту: {0}",
                                    ex.Message);
         }
      }
   }
}
```

При любой реализации – один поток на клиента (листинг 2.7) или один поток на тикер (листинг 2.8) – остается открытым вопрос о надежности. Например, если операция обратного вызова клиента завершается с ошибкой, служба просто выво-

дит сообщение на консоль, но не пробует повторить попытку. Надо ли пытаться вызвать клиента еще раз и, если да, с каким интервалом и когда прекращать попытки? Или, быть может, существует некий промежуток времени, в течение которого клиент заведомо не сможет получать обновления; где тогда сохранять накопившиеся обновления, чтобы отправить их позже? Все это важные вопросы, которые можно решить с помощью брокера сообщений, например Microsoft BizTalk Server, или аналогичного продукта. Брокеры сообщений обычно располагают внешним хранилищем (база данных, файловая система или очередь сообщений) и средствами конфигурирования, позволяющими описать транспортные протоколы и механизмы повтора. Но они обходятся далеко не даром с точки зрения производительности, сложности и стоимости, поэтому решение зависит от конкретных требований.

Реализация клиентской части дуплексного контракта

Чтобы принять участие в дуплексном обмене сообщениями, клиент должен реализовать АПК в смысле WCF, то есть определить адрес, куда служба сможет посылать сообщения, привязку, описывающую, как доставлять сообщения клиенту, и контракт, определяющий структуру сообщений. К счастью, все это делается автоматически на этапе генерации клиентского прокси-класса и позже на этапе выполнения с привлечением инфраструктуры каналов.

Для генерации прокси-класса служит утилита `svcutil.exe` или операция Add Service Reference в Visual Studio. Прокси-класс определяет интерфейс с таким же именем, как у службы, но с добавлением в конец слова `Callback`. Так, если интерфейс контракта о службе называется `IStockService`, то у клиентского интерфейса будет имя `IStockServiceCallback`. Клиент должен реализовать класс, производный от этого интерфейса.

На этапе выполнения все обращения к клиенту осуществляются строго в соответствии с определением оконечной точки, которой посылаются сообщения. Основное отличие клиентской оконечной точки от серверной состоит в том, что для клиента оконечную точку динамически создает WCF. В клиентской программе не существует ни конфигурационного файла, ни явных обращений к методам класса `ServiceHost`. Обо всем этом заботится WCF, на долю клиента выпадает только реализация класса, производного от сгенерированного интерфейса.

В листинге 2.9 показан код клиента, который вызывает метод `RegisterForUpdates` службы StockService, регистрируясь для периодического получения обновлений. Клиент также реализует метод обратного вызова `PriceUpdate`, которому служба будет передавать котировки акций. Обратите внимание, как для создания прокси создается и используется объект `InstanceContext`. В нем хранится контекстная информация для службы, в частности ссылки на входной и выходной каналы, созданные от имени клиента.

Листинг 2.9. Реализация дуплексного контракта на стороне клиента

```
using System;
using System.ServiceModel;

namespace Client
{
   public class CallbackHandler : IServerStockCallback
   {
      static InstanceContext site =
                        new InstanceContext(new CallbackHandler());
      static ServerStockClient proxy = new ServerStockClient (site);

      public void PriceUpdate(string ticker, double price)
      {
         Console.WriteLine("Получено извещение в : {0}. {1}:{2}",
                                    System.DateTime.Now, ticker, price);
      }

      class Program
      {
         static void Main(string[] args)
         {
            proxy.RegisterForUpdates("MSFT");

            Console.WriteLine("Для завершения нажмите любую клавишу");
            Console.ReadLine();
         }
      }
   }
}
```


Службы с несколькими контрактами и оконечными точками

По определению, служба – это набор оконечных точек. У каждой оконечной точки имеется адрес, привязка и контракт. В контракте объявляются функциональные возможности оконечной точки. Адрес – это место в сети, в которое можно обратиться за получением услуг от приложения (или службы), а привязка описывает, как именно следует обращаться.

Между оконечными точками и контрактами существует отношение один-комногим. У оконечной точки может быть только один контракт, однако на один и тот же контракт могут ссылаться несколько оконечных точек. И, хотя в описании оконечной точки разрешается указать лишь один контракт, с помощью агрегирования интерфейсов единственный контракт может раскрывать несколько интерфейсов. Кроме того, по одному и тому же адресу могут располагаться несколько оконечных точек с одинаковой привязкой, но разными контрактами; тем самым создается иллюзия, будто единственная оконечная точка реализует более одного контракта.

Объявляя для нескольких оконечных точек службы один и тот же контракт, вы можете сделать его доступным через разные привязки. Можно определить одну оконечную точку, которая объявляет свой контракт через привязку WS-I Basic Profile, обеспечивая доступность максимальному числу клиентов, и тот же контракт объявить в другой оконечной точке с привязкой к протоколу TCP и двоичным кодированием для достижения максимальной производительности. Агре-

гируя несколько интерфейсов в один, вы можете предоставить консолидированный доступ к функциональности, которая первоначально была разнесена по нескольким интерфейсам одной службы.

В листинге 2.10 показаны два контракта о службе: `IGoodStockService` и `IGreatStockService`, которые агрегированы в третий контракт `IStockServices`. В агрегате реализованы методы, определенные в обоих интерфейсах. Хотя интерфейсы службы можно наследовать, атрибут `[ServiceContract]` необходимо задать в каждом интерфейсе, который оконечная точка должна раскрывать.

Листинг 2.10. Объявление нескольких контрактов для одной оконечной точки

```
namespace EssentialWCF
{
   [ServiceContract]
   public interface IGoodStockService
   {
      [OperationContract]
      double GetStockPrice(string ticker);
   }
   [ServiceContract]
   public interface IGreatStockService
   {
      [OperationContract]
      double GetStockPriceFast(string ticker);
   }

   [ServiceContract]
   public interface IAllStockServices :
                           IGoodStockService, IGreatStockService { };

   public class AllStockServices : IAllStockServices
   {
      public double GetStockPrice(string ticker)
      {
         Thread.Sleep(5000);
         return 94.85;
      }
      public double GetStockPriceFast(string ticker)
      {
         return 94.85;
      }
   }
}
```

В листинге 2.11 приведен конфигурационный файл, в котором описано несколько оконечных точек для всех трех контрактов. Для контракта `IGoodStockService` есть одна оконечная точка, для контракта `IGreatStockService` – две, и для контракта `IAllStockServices` – одна.

Поскольку существует несколько оконечных точек, пользующихся привязкой с общей схемой адресации, для каждой необходимо задать уникальный адрес. Мы использовали относительные адреса, поэтому полный адрес оконечной точки образуется из базового адреса службы, в конец которого дописана относительная часть.

Листинг 2.11. Объявление нескольких оконечных точек службы

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>

      <services>
         <service name="EssentialWCF.StockServices"
                        behaviorConfiguration="mexServiceBehavior">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EssentialWCF/"/>
               </baseAddresses>
            </host>
            <endpoint name="GoodStockService"
                           binding="basicHttpBinding"
                           contract="EssentialWCF.IGoodStockService" />
            <endpoint name="BetterStockService"
                           address="better"
                           binding="basicHttpBinding"
                           contract="EssentialWCF.IGreatStockService" />
            <endpoint name="BestStockService"
                           address="best"
                           binding="wsHttpBinding"
                           contract="EssentialWCF.IGreatStockService" />
            <endpoint name="AllStockServices"
                           address="all"
                           binding="wsHttpBinding"
                           contract="EssentialWCF.IAllStockServices" />
            <endpoint address="mex"
                           binding="mexHttpBinding"
                           contract="IMetadataExchange" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="mexServiceBehavior">
               <serviceMetadata httpGetEnabled="True"/>
            </behavior>
         </serviceBehaviors>
      </behaviors>

   </system.serviceModel>
</configuration>
```

Так как контракт `IGreatStockService` объявлен несколькими оконечными точками, клиентские приложения при создании экземпляра прокси-класса для такого контракта должны ссылаться на оконечную точку по имени. Если не задать имя оконечной точки, WCF возбудит исключение, поскольку не будет знать, к какой точке обратиться. В листинге 2.12 демонстрируется два способа использования прокси-класса `GreatStockServiceClient`: сначала для доступа к оконечной точке `BetterStockService` с помощью привязки `basicHttpBinding`, а потом для доступа к `BestStockService` с помощью привязки `wsHttpBinding`.

Листинг 2.12. Задание одной из нескольких оконечных точек по имени

```
using (localhost.GreatStockServiceClient proxy = new
              Client.localhost.GreatStockServiceClient
                                    ("BetterStockService"))
{
   Console.WriteLine(proxy.GetStockPriceFast("MSFT"));
}
using (localhost.GreatStockServiceClient proxy = new
            Client.localhost.GreatStockServiceClient
                                    ("BestStockService"))
{
   Console.WriteLine(proxy.GetStockPriceFast("MSFT"));
}
```

Имена операций, типов, действий и пространств имен в WSDL

WCF генерирует различные раскрываемые внешнему миру артефакты службы, опираясь на имена классов и атрибутов в исходном тексте службы. Эти артефакты раскрываются через оконечную точку MEX и обычно потребляются клиентом на этапе проектирования в виде WSDL-документа. Затем на стороне клиента WSDL-описание используется для генерации кода, который строит сообщение нужного формата, отправляемое службе. Поэтому выбранные вами имена классов, методов и параметров могут быть видны далеко за пределами исходного кода службы.

Однако в общем случае не рекомендуется раскрывать внешнему миру внутренние имена и детали реализации. Например, в программе может присутствовать алгоритм распределения с именем BurgerMaster, который вы хотели бы представить внешнему миру в виде операции Resources. Или, возможно, в организации существуют стандарты именования служб. К счастью, все раскрываемые службой имена можно контролировать с помощью атрибутов `[ServiceContract]`, `[OperationContract]` и `[ServiceBehavior]`. В таблице 2.3 описано, как WCF-атрибуты при элементах программы влияют на имена элементов в WSDL-описании.

Таблица 2.3. WCF-атрибуты, позволяющие переопределить подразумеваемые по умолчанию имена WSDL

Элемент WSDL	Атрибут WCF
`targetNamespace`	По умолчанию http://tempuri.org. Можно изменить с помощью атрибута `[ServiceBehavior]` в коде.
`wsdl:service` и `wsdl:definitions`	`[ServiceBehavior(Name="myServiceName")]`
`wsdl:porttype`	`[ServiceContract(Name="myContractName")]`
`wsdl:operation` и `soap:operation`	`[OperationContract(Name="myOperationName")]`
`xsl:element`	`[MessageParameter(Name = "myParamName")]`

Таблица 2.3. WCF-атрибуты, позволяющие переопределить подразумеваемые по умолчанию имена WSDL (окончание)

Элемент WSDL	Атрибут WCF
`wsdl:input` и `wsdl:output` `wsdl:Binding`	`[OperationContract(Action="myOperationAction", ReplyAction="myOperationReplyAction")]` Пользуйтесь атрибутами `[DataContract]` и `[ServiceContract]`

В листинге 2.13 приведена служба, которая с помощью атрибутов WCF переопределяет подразумеваемые по умолчанию имена.

Листинг 2.13. Управление именами WSDL в определении службы

```
[ServiceBehavior (Namespace="http://MyService/")]
[ServiceContract
                 (Name="MyServiceName",
                 Namespace="http://ServiceNamespace")]
public class BurgerMaster
{
      [return: MessageParameter(Name = "myOutput")]
      [OperationContract
                 (Name="OperationName",
                 Action="OperationAction",
                 ReplyAction="ReplyActionName")]

      public double GetStockPrice(string ticker)
      {
         return 100.00;
      }
}
```

Для генерации WSDL-описания службы можно запустить утилиту `svcutil.exe` с флагом `-t:metadata`. Или, если служба раскрывает оконечную точку MEX с привязкой к протоколу http, то можно посмотреть WSDL-описание, обратившись к базовому адресу службы из браузера Internet Explorer. Формат WSDL в обоих случаях слегка отличается, но эти различия несущественны и относятся исключительно к оформлению. В листинге 2.14 показано WSDL-описание, ассоциированное с кодом из листинга 2.13. Имена `wsdl:portType`, `wsdl:operation` и `wsdl:action` были заданы в коде. Отметим, что `wsdl:portType` носит имя `MyServiceName`, а не `BurgerMaster`, как класс в листинге 2.13.

Листинг 2.14. Как задание имен отразилось на WSDL-описании

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://ServiceNamespace"
   targetNamespace="http://ServiceNamespace"
   .
   .
```

```
   .
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <wsdl:types>
      <xsd:schema targetNamespace="http://ServiceNamespace/Imports">
         <xsd:import
                  schemaLocation=
                              "http://localhost:8000/EssentialWCF/?xsd=xsd0"
                  namespace="http://ServiceNamespace" />
         <xsd:import
                  schemaLocation=
                              "http://localhost:8000/EssentialWCF/?xsd=xsd1"
                  namespace="http://schemas.microsoft.com/
                                    2003/10/Serialization/"/>
      </xsd:schema>
   </wsdl:types>

   <wsdl:message name="MyServiceName_OperationName_InputMessage">
      <wsdl:part name="parameters" element="tns:OperationName" />
   </wsdl:message>
   <wsdl:message name="MyServiceName_OperationName_OutputMessage">
      <wsdl:part name="parameters" element="tns:OperationNameResponse" />
   </wsdl:message>
   <wsdl:portType name="MyServiceName">
      <wsdl:operation name="OperationName">
         <wsdl:input wsaw:Action="OperationAction"
                     message="tns:MyServiceName_OperationName_InputMessage" />
         <wsdl:output wsaw:Action="ReplyActionName"
                     message="tns:MyServiceName_OperationName_OutputMessage" />
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

Контракты о данных

Внутри службы функциональные возможности реализованы программно. А вне нее они описываются WSDL-документом. Данные внутри службы представлены простыми и составными типами, тогда как вне нее они описаны на языке XML Schema Definitions (XSD). Контракты о данных в WCF осуществляют отображение между типами .NET CLR, определенными в коде, и XSD-определениями, постулированными консорциумом W3C (www.w3c.org/), которые используются для обращения к службе извне.

При работе с WCF разработчики тратят больше времени на написание кода и определение семантики интерфейса, чем на копания в синтаксисе XSD и WSDL. Мы не хотим сказать, что синтаксис XSD и WSDL вообще не важен; для построения интероперабельных систем на гетерогенных платформах, эти элементы имеют первостепенную важность. Но компиляторы прекрасно справляются с трансляцией структур данных в .NET-совместимых языках в эквивалентные представления на XSD и WSDL, необходимые для кросс-платформенной интероперабельности.

На этапе проектирования мы используем атрибут `[DataContract]` для обозначения того, какие классы следует представить на языке XSD и включить

в WSDL-описание, раскрываемое службой. Затем мы уточняем XSD-представление, снабжая атрибутом `[DataMember]` те члены класса, которые должны быть видны извне. На этапе выполнения класс `DataContractSerializer`, входящий в состав WCF, сериализует объекты в виде XML, применяя правила, описанные атрибутами `[DataContract]` и `[DataMember]`. На рис. 2.7 показаны классы на исходном .NET-языке и их представление в виде XSD-схемы, понятное системе на любой другой платформе.

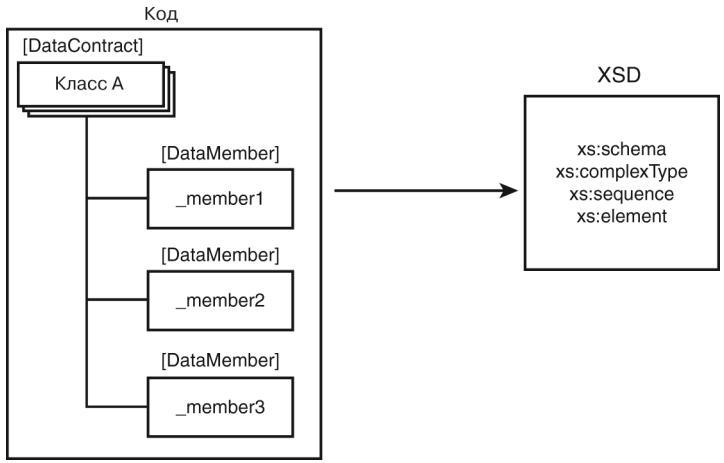

Рис. 2.7. Высокоуровневая диаграмма трансляции артефактов кода на язык XSD

На рис. 2.8 изображена та же трансляция, но для ясности включены конкретные синтаксические конструкции на языке C# и соответствующие им элементы XSD.

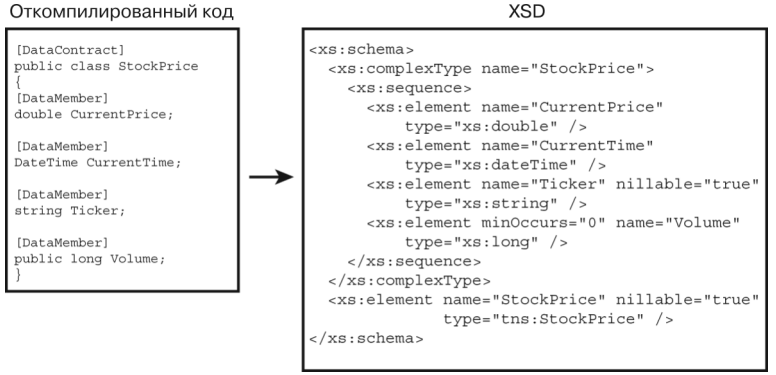

Рис. 2.8. Высокоуровневая диаграмма трансляции языковых конструкций на язык XSD

Класс `DataContractSerializer` сериализует тип и раскрывает его в контракте на языке WSDL, если удовлетворяется хотя бы одно из следующих условий:

- типы, помеченные атрибутами `[DataContract]` и `[DataMember]`;
- типы, помеченные атрибутом `[CollectionDataContract]`;
- типы, производные от `IXmlSerializable`;
- типы, помеченные атрибутом `[Serializable]`, за исключением членов, помеченных атрибутом `[NonSerialized]`;
- типы, помеченные атрибутом `[Serializable]` и реализующие интерфейс `ISerializable`;
- встроенные в CLR примитивные типы, например Int32 и String;
- массив байтов, DateTime, TimeSpan, Guid, Uri, XmlQualifiedName, XmlElement, и XmlNode;
- массивы и наборы, например: `List<T>`, `Dictionary<K,V>` и `Hashtable`;
- перечисления.

Определение XSD-схемы для класса .NET

Атрибут `[DataContract]`, определенный в пространстве имен `System.Runtime.Serialization`, обозначает, что класс должен быть представлен в формате XSD в WSDL-документе, описывающем службу. Если класс не снабжен атрибутом `[DataContract]`, то он и не будет включен в WSDL-описание. По умолчанию имя XSD-схемы совпадает с именем класса, а имя пространства имен схемы образуется путем конкатенации строки `http://schemas.datacontract.org/2004/07/` с пространством имен .NET, в котором определен класс. То и другое можно переопределить, если вы хотите самостоятельно управлять именами, видимыми извне. Например, внутреннее имя класса `reqOrderIn` можно раскрыть в XSD как `Order`. В листинге 2.16 показано, как переопределяются имена и пространство имен в XSD.

Атрибутом `[DataMember]`, который также определен в пространстве имен `System.Runtime.Serialization`, помечаются те члены .NET-класса, снабженного атрибутом `[DataContract]`, которые нужно включить в XSD-схему. Если член не помечен атрибутом `[DataMember]`, то он не войдет в XSD-схему, несмотря на то, что является членом экспортируемого класса. По умолчанию никакие члены класса не включаются в XSD-схему, вы должны определять это явно. Видимость членов класса (public или private) не влияет на включение в XSD-схему; решение основывается исключительно на наличии или отсутствии атрибута `[DataMember]`.

В листинге 2.15 приведено определение класса `StockPrice` с пятью открытыми членами-данными. Три из них – `ticker`, `theCurrentPrice` и `theCurrentTime` – обязательны, поскольку помечены признаком `isRequired=true`. Продемонстрированы также несколько дополнительных возможностей атрибута `[DataMember]`:

- имена всех членов класса в коде начинаются с префикса `m_`. Но при экспорте в XSD имена переопределяются, так чтобы префикс `m_` не был виден в интерфейсе службы;
- в атрибуте `[DataMember]` задается порядок следования членов класса. Если порядок не задан, то в XSD члены класса идут в алфавитном порядке. Обычно порядок не имеет значения, но для целей интероперабельности важно уметь им управлять. Если вы посылаете сообщения службе, которая ожидает, что элементы упорядочены определенным образом, то этот атрибут позволяет добиться желаемого результата;
- члены `m_CurrentPrice`, `m_CurentType` и `m_ticker` помечены как обязательные, а члены `m_dailyVolume` и `m_dailyChange` – нет. Необязательные члены класса можно опускать при трансляции объекта в XML-экземпляр, и он тем не менее будет считаться достоверным (valid) при сверке с XSD.

Листинг 2.15. Определение контракта о данных

```
using System;
using System.ServiceModel;
using System.Runtime.Serialization;

namespace EssentialWCF
{
   [DataContract (Namespace="http://EssentialWCF",Name="StockPrice")]
   public class clsStockPrice
   {
      [DataMember (Name="CurrentPrice",Order=0,IsRequired=true)]
      public double theCurrentPriceNow;

      [DataMember(Name = "CurrentTime", Order=1, IsRequired = true)]
      public DateTime theCurrentTimeNow;

      [DataMember(Name = "Ticker", Order=2, IsRequired = true)]
      public string theTickerSymbol;

      [DataMember(Name = "DailyVolume", Order=3, IsRequired = false)]
      public long theDailyVolumeSoFar;

      [DataMember(Name = "DailyChange", Order=4, IsRequired = false)]
      public double theDailyChangeSoFar;
   }

   [ServiceContract]
   public class StockService
   {
      [OperationContract]
      private clsStockPrice GetPrice(string ticker)
      {
         clsStockPrice s = new clsStockPrice();
         s.theTickerSymbol = ticker;
         s.theCurrentPriceNow = 100.00;
         s.theCurrentTimeNow = System.DateTime.Now;
         s.theDailyVolumeSoFar = 450000;
         s.theDailyChangeSoFar = .012345;
         return s;
      }
   }
}
```

Команда `svcutil.exe /t:metadata` генерирует XSD-код, анализируя члены класса, помеченные атрибутом `[DataMember]`. В листинге 2.16 представлен XSD-код, сгенерированный для программы из листинга 2.15. Обратите внимание, что имена и порядок элементов соответствуют тем, что были заданы с помощью WCF-атрибутов. Еще отметим, что для необязательных членов класса в XSD-схеме указан атрибут `minOccurs=0`.

Листинг 2.16. Сгенерированный XSD-файл, представляющий контракт о данных

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://EssentialWCF" elementFormDefault="qualified"
targetNamespace="http://EssentialWCF" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
   <xs:complexType name="StockPrice">
      <xs:sequence>
         <xs:element name="CurrentPrice" type="xs:double" />
         <xs:element name="CurrentTime" type="xs:dateTime" />
         <xs:element name="Ticker" nillable="true" type="xs:string" />
         <xs:element minOccurs="0" name="DailyVolume" type="xs:long" />
         <xs:element minOccurs="0" name="DailyChange" type="xs:double" />
      </xs:sequence>
   </xs:complexType>
   <xs:element name="StockPrice" nillable="true" type="tns:StockPrice"/>
</xs:schema>
```

Определение иерархий классов

Составные типы обычно представляются в программе классами. Кроме того, при их определении используется наследование как способ постепенного наращивания специфичности конструкции. Таким образом, от общего типа, например «цена», можно породить подкласс более специфичного типа, например «цена акции» или «цена дома». WCF поддерживает иерархии классов, позволяя корректно представлять их на языке WSDL, сериализовывать и десериализовать в формат XML и включать в агрегат атрибуты каждого класса.

В листинге 2.17 определен класс `Price` с тремя элементами и его подкласс `StockPrice`. Для каждого класса задано пространство имен, чтобы в представлении на XML у них были полностью квалифицированные имена. Каждый элемент сохраняет свое пространство имен.

Листинг 2.17. Включение контракта о данных в иерархию классов

```
[DataContract(Namespace = "http://EssentialWCF/Price/")]
public class Price
{
   [DataMember] public double CurrentPrice;
   [DataMember] public DateTime CurrentTime;
   [DataMember] public string Currency;
}

[DataContract(Namespace = "http://EssentialWCF/StockPrice")]
public class StockPrice : Price
{
   [DataMember] public string Ticker;
   [DataMember] public long DailyVolume;
   [DataMember] public double DailyChange;
}
```


В листинге 2.18 приведены две XSD-схемы, описывающие эту иерархию. Первой показана XSD-схема `Price`, за ней `StockPrice`. Отметим, что схема `StockPrice` импортирует схему `Price`. Обратите также внимание, что в описаниях всех элементов задан атрибут `minOccurs=0`, поскольку в коде ни один член не помечен признаком `[isRequired=true]`.

Листинг 2.18. Иерархия классов, представленная в виде XSD-схем

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://EssentialWCF/Price/"
                        elementFormDefault="qualified"
                        targetNamespace="http://EssentialWCF/Price/"
                        xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:complexType name="Price">
      <xs:sequence>
         <xs:element minOccurs="0" name="Currency"
                           nillable="true" type="xs:string" />
         <xs:element minOccurs="0" name="CurrentPrice" type="xs:double" />
         <xs:element minOccurs="0" name="CurrentTime" type="xs:dateTime"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="Price" nillable="true" type="tns:Price" />
</xs:schema>

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://EssentialWCF/StockPrice"
                        elementFormDefault="qualified"
                        targetNamespace="http://EssentialWCF/StockPrice"
                        xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:import namespace="http://EssentialWCF/Price/" />
   <xs:complexType name="StockPrice">
      <xs:complexContent mixed="false">
         <xs:extension xmlns:q1="http://EssentialWCF/Price/"
                              base="q1:Price">
            <xs:sequence>
               <xs:element minOccurs="0" name="DailyChange"
                                 type="xs:double" />
               <xs:element minOccurs="0" name="DailyVolume"
                                 type="xs:long" />
               <xs:element minOccurs="0" name="Ticker"
                                 nillable="true" type="xs:string" />
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:element name="StockPrice" nillable="true" type="tns:StockPrice"/>
</xs:schema>
```

Тело SOAP-сообщения, содержащего сериализованный тип `StockPrice`, показано в листинге 2.19. Отметим, что пространства имен, заданные в атрибутах классов `Price` и `StockPrice`, мигрируют из листинга 2.17 в XSD-схему в листинге 2.18 и далее в тело SOAP-сообщения.

Листинг 2.19. Представление сериализованной иерархии классов в теле SOAP-сообщения

```
<s:Body>
   <GetPriceResponse xmlns="http://EssentialWCF/FinanceService/">
      <GetPriceResult
                  xmlns:a="http://EssentialWCF/StockPrice"
                  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
         <Currency xmlns="http://EssentialWCF/Price/">Dollars</Currency>
         <CurrentPrice xmlns="http://EssentialWCF/Price/">
                           100</CurrentPrice>
         <CurrentTime xmlns="http://EssentialWCF/Price/">
                           2006-12-13T21:18:51.313-05:00</CurrentTime>
         <a:DailyChange>0.012345</a:DailyChange>
         <a:DailyVolume>450000</a:DailyVolume>
         <a:Ticker>msft</a:Ticker>
      </GetPriceResult>
   </GetPriceResponse>
</s:Body>
```

Включение дополнительных типов в WSDL с помощью атрибута KnownType

Типы данных включаются в WSDL-описание, если удовлетворяют сформулированным выше условиям. Но бывают особые случаи, когда необходимо принудительно включить объявление типа в WSDL-контракт.

Один такой пример – иерархии классов. Если в оконечную точку поступает объект сериализованного производного класса, а та ожидает сериализованное представление базового класса, то WCF не будет знать, как десериализовать класс, поскольку производный класс не является частью контракта. Другой пример – класс `hashtable`, в котором хранятся объекты других классов. В WSDL-описание класс `hashtable` будет включен, а описания хранящихся в нем классов – нет.

В таких случаях необходимо явно сообщить WCF о классах, которые требуется включить в WSDL-контракт. Делается это с помощью атрибута `KnownType` одним из четырех способов: добавлением его к атрибуту `[DataContract]`, `[ServiceContract]` или `[OperationContract]`, или путем ссылки на сборку этого атрибута в конфигурационном файле либо при генерации WSDL.

В листинге 2.20 показан контракт о данных, в котором определен базовый класс `Price` и два производных от него класса `StockPrice` и `MetalPrice`. Обратите внимание на атрибут `[KnownType]` в контракте о данных. Он извещает WCF о необходимости включить XSD-представление классов `StockPrice` и `MetalPrice` в WSDL-документ, в котором объявляется контракт. В листинге приведена также

реализация службы. Операция `GetPrice` полиморфна и возвращает один из типов `StockPrice` или `MetalPrice` в зависимости от того, что запрашивалось. Клиент, вызывающий метод `GetPrice` через прокси, должен будет привести результат к ожидаемому типу.

Листинг 2.20. Употребление атрибута KnownType в контракте о данных

```
using System;
using System.ServiceModel;
using System.Runtime.Serialization;

namespace EssentialWCF
{
   [DataContract(Namespace = "http://EssentialWCF/")]
   [KnownType(typeof(StockPrice))]
   [KnownType(typeof(MetalPrice))]
   public class Price
   {
      [DataMember] public double CurrentPrice;
      [DataMember] public DateTime CurrentTime;
      [DataMember] public string Currency;
   }

   [DataContract(Namespace = "http://EssentialWCF/")]
   public class StockPrice : Price
   {
      [DataMember] public string Ticker;
      [DataMember] public long DailyVolume;
   }
   [DataContract(Namespace = "http://EssentialWCF/")]
   public class MetalPrice : Price
   {
      [DataMember] public string Metal;
      [DataMember] public string Quality;
   }

   [ServiceBehavior (Namespace="http://EssentialWCF/FinanceService/")]
   [ServiceContract (Namespace="http://EssentialWCF/FinanceService/")]
   public class StockService
   {
      [OperationContract]
      private Price GetPrice(string id, string type)
      {
         if (type.Contains("Stock"))
         {
            StockPrice s = new StockPrice();
            s.Ticker = id;
            s.DailyVolume = 45000000;
            s.CurrentPrice = 94.15;
            s.CurrentTime = System.DateTime.Now;
            s.Currency = "USD";
            return s;
         }
         if (type.Contains("Metal"))
         {
```

```
            MetalPrice g = new MetalPrice();
            g.Metal = id;
            g.Quality = "0.999";
            g.CurrentPrice = 785.00;
            g.CurrentTime = System.DateTime.Now;
            g.Currency = "USD";
            return g;
         }
         return new Price();
      }
   }
}
```

Можно было бы также задать известный тип на уровне `OperationContract` с помощью атрибута `[ServiceKnownType]`. Если известные типы определяются на уровне операции, то производные типы можно использовать только в операции, для которой известные типы заданы. Другими словами, не все операции службы могут работать с производными типами. В листинге 2.21 приведен фрагмент кода, где используется атрибут `[ServiceKnownType]`. Здесь клиент может обратиться к операции `GetPrice`, и при получении ответного сообщения десериализатор будет знать, как создать объект типа `StockPrice` или `MetalPrice`. Но при вызове операции `SetPrice` клиент может передать только объект типа `Price`, а не `StockPrice` или `MetalPrice`, поскольку сериализатор не знает, как преобразовать производные типы в XML.

Листинг 2.21. Определение известных типов в контракте об операции

```
.
.
.
   [ServiceBehavior (Namespace="http://EssentialWCF/FinanceService/")]
   [ServiceContract (Namespace="http://EssentialWCF/FinanceService/")]
   public class StockService
   {
      [ServiceKnownType(typeof(StockPrice))]
      [ServiceKnownType(typeof(MetalPrice))]
      [OperationContract]
      private Price GetPrice(string id, string type)
      {
         .
         .
         .
      }
      [OperationContract]
      void SetPrice(Price p)
      {
         .
         .
         .
      }
```

Недостаток задания известных типов в коде – неважно, на уровне контракта о данных или контракта о службе, – заключается в том, что вы на этапе компиляции должны знать обо всем множестве производных типов. Если добавится новый тип, программу придется перекомпилировать. Решить эту проблему можно двумя способами.

Во-первых, можно перенести ссылки на известные типы из кода в конфигурационный файл службы, включив информацию о типе в секцию `system.runtime.serialization`. Чтобы сохранить сведения об иерархии классов, вы должны сначала добавить ссылку на базовый класс, а затем элементы `knownType`, содержащие ссылки на производные классы. Это продемонстрировано в листинге 2.22, где `EssentialWCF.Price` – базовый класс, а `EssentialWCF.StockPrice` и `EssentialWCF.MetalPrice` – производные. Все эти типы находятся в DLL StockService.

Листинг 2.22. Определение известных типов в конфигурационном файле

```
<system.runtime.serialization>
   <dataContractSerializer>
      <declaredTypes>
         <add type="EssentialWCF.Price, StockService,
                    Version=1.0.0.0, Culture=neutral,
                    PublicKeyToken=null">
            <knownType type="EssentialWCF.StockPrice, StockService,
                    Version=1.0.0.0, Culture=neutral,
                    PublicKeyToken=null"/>
            <knownType type="EssentialWCF.MetalPrice, StockService,
                    Version=1.0.0.0, Culture=neutral,
                    PublicKeyToken=null"/>
         </add>
      </declaredTypes>
   </dataContractSerializer>
</system.runtime.serialization>
```

Самое общее решение задачи о задании производных типов в контракте – сгенерировать их во время выполнения. Это можно сделать, благодаря наличию в WCF некоторых точек подключения. Конструкторы атрибутов `[KnownType]` и `[ServiceKnownType]` принимают строковый параметр. Строка интерпретируется как имя метода, который вызывается в момент сериализации или десериализации и возвращает список известных типов. Если имеется репозиторий метаданных, то можно получить из него информацию о типах и предоставить ее на этапе выполнения. В листинге 2.23 показана простая реализация, когда имена типов «зашиты» в код метода `GetKnownTypes`, а не извлекаются из внешнего репозитория.

Листинг 2.23. Программное определение известных типов во время выполнения

```
[DataContract(Namespace = "http://EssentialWCF/")]
[KnownType("GetKnownTypes")]
public class Price
{
   [DataMember] public double CurrentPrice;
   [DataMember] public DateTime CurrentTime;
```

```
   [DataMember] public string Currency;
   static Type[] GetKnownTypes()
   {
      return new Type[] { typeof(StockPrice),
                          typeof(MetalPrice) };
   }
}

[DataContract(Namespace = "http://EssentialWCF/")]
public class StockPrice : Price
{
   [DataMember] public string Ticker;
   [DataMember] public long DailyVolume;
}
[DataContract(Namespace = "http://EssentialWCF/")]
public class MetalPrice : Price
{
   [DataMember] public string Metal;
   [DataMember] public string Quality;
}
```

Контроль версий контрактов о данных

Изменения неизбежны. Меняется бизнес, меняются технологии, меняется законодательство, а вместе с ними меняются и программные контракты. И потому необходима надежная стратегия контроля версий. Нужно заранее подумать о неизбежности изменений и решить, как обеспечить обратную совместимость с существующими клиентами.

Чаще всего потребность в контроле версий возникает, когда в существующий контракт о данных добавляются новые члены. Следуя стратегии, описанной в этом разделе, вы сможете реализовать такие изменения, не нарушая работу существующих клиентов. Если же необходимо отказаться от обратной совместимости, то следует создать абсолютно новую версию всего контракта о данных, изменив его имя или пространство имен.

Говоря о том, что изменения не нарушают работу, мы не вполне честны. То, что не нарушает работу с точки зрения WCF, вполне может приводить к несовместимости с другими системами. Например, система, которая требует строгого соответствия схеме, может отвергнуть сообщение, если в его XML-представлении есть неожиданные элементы. Термин «ненарушающее» в этой главе относится к изменениям, которые не сказываются на взаимодействии двух систем на базе WCF.

Ненарушающие изменения

Есть два вида изменений, которые не нарушают совместимость с существующими клиентами:

- добавление новых необязательных членов;
- удаление старых необязательных членов.

В обоих случаях можно по новому сообщению создать старый тип, просто проигнорировав новые или отсутствующие необязательные члены. И наоборот – из старого типа можно создать новое сообщение. Класс `DataContractSerializer` делает это автоматически во время выполнения.

Нарушающие изменения

Хотя некоторые атрибуты в контракте о данных можно изменять, не нарушая обратную совместимость, большинство изменений приводят к неработоспособности старых клиентов. Это относится к любому из следующих изменений:

- изменение имени или пространства имен контракта о данных;
- переименование существующего обязательного члена данных;
- добавление нового члена данных с именем, которое раньше уже было задействовано;
- изменение типа существующего члена данных;
- добавление новых членов с признаком `IsRequired=true` в атрибуте `DataMemberAttribute`;
- удаление существующих членов с признаком `IsRequired=true` в атрибуте `DataMemberAttribute`.

В листинге 2.24 приведены определения двух контрактов о данных. Первый определен в версии V1 службы, второй – в версии V2. Отметим, что при переходе от V1 к V2 был удален член `Currency` и добавлен член `DailyVolume`. Это ненарушающее изменение.

Листинг 2.24. Ненарушающее изменение контракта о данных – добавление и удаление членов

```
[DataContract (Namespace="http://EssentialWCF")]
public class StockPrice //V1
{
   [DataMember] public double CurrentPrice;
   [DataMember] public DateTime CurrentTime;
   [DataMember] public string Ticker;
   [DataMember] public string Currency;
}

[DataContract (Namespace="http://EssentialWCF")]
public class StockPrice //V2
{
   [DataMember] public double CurrentPrice;
   [DataMember] public DateTime CurrentTime;
   [DataMember] public string Ticker;
   [DataMember] public int DailyVolume;
}
```

Чтобы существующие клиенты пропускали данные после добавления новых членов, исходный контракт о данных должен поддерживать расширяемость. Иначе говоря, исходный контракт должен уметь сериализовывать неизвестные будущие данные. Это обеспечит *неизменный оборот* (round tripping), когда клиент мо-

жет передать данные версии V2 службе версии V1, а та вернет клиенту данные, в которых элементы, относящиеся к версии V2, не изменились. WCF реализует расширяемость по умолчанию в коде прокси-класса, который генерирует `svcutil.exe`. Если эта возможность вам не нужна, ее можно отключить, добавив строку `<dataContractSerializer ignoreExtensionDataObject="true"/>` в секцию `ServiceBehavior` конфигурационного файла службы.

В листинге 2.25 приведен код клиента, который вызывает метод `GetPrice`, чтобы получить объект `StockPrice`, а затем передает его методу `StoreStockPrice`. Предположим, что прокси-класс для `StockService` был сгенерирован утилитой `svcutil.exe` для версии службы V1, а затем служба была модернизирована до версии V2, показанной в листинге 2.24. Если тот же клиент обратится к версии V2, то `GetPrice` вернет XML-сообщение с добавленным членом `DailyVolume` и без члена `Currency`. Десериализатор контракта о данных, который знает об объекте `StockPrice` из версии V1, поместит член `DailyVolume` в поле `ExtensionData` объекта и не станет жаловаться на отсутствие члена `Currency`. Клиент получит ожидаемый объект `StockPrice`, в котором член `Currency` инициализирован значением по умолчанию, при этом размер объекта немного увеличится из-за дополнительных данных (`DailyVolume`). Таким образом, служба вернула корректные с точки зрения версии V2 данные, а клиент получил их корректное представление в версии V1.

Листинг 2.25. Вызов службы версии V2 по контракту для версии V1

```
localhost.StockServiceClient proxy=new localhost.StockServiceClient())
localhost.StockPrice s = proxy.GetPrice("msft");
proxy.StoreStockPrice(s);
```

Эквивалентность контрактов о данных

Если вы применяете WCF для организации некоторой службы, а прокси-класс, необходимый для доступа к этой службе, строите с помощью `svcutil.exe`, то обычно вообще не задумываетесь о том, в каком виде сообщения между клиентом и службой передаются по сети. Контракт о данных говорит WCF, как сериализовать тип .NET в информационный набор XML Infoset и десериализовать последний в исходную форму. Набор XML Infoset можно кодировать в двоичном или текстовом виде в зависимости от используемой привязки, но код, который пишете вы, ничего о кодировке не знает. Таким образом, вы работаете с типами .NET, а не с кодированным представлением удовлетворяющего всем стандартам набора XML Infoset, который передается по проводам.

Но бывают случаи, когда клиент и служба должны работать с разными типами данных. Например, так происходит, если клиент и служба разрабатывались в разных организациях или если только одна сторона написана с использованием WCF. На самом деле, когда вы применяете для генерации клиентского прокси-класса утилиту `svcutil.exe` или операцию Add Service Reference, есть все шансы, что имена членов на стороне клиента будут отличаться от имен членов на стороне

службы. Однако, управляя именами с помощью атрибута `[DataMember]`, вы можете привести их к единому виду в XML-представлении. При условии, что клиент и служба работают с эквивалентным XML-представлением, WCF может десериализовать набор XML Infoset в различные типы .NET. Если два класса сериализуются в одну и ту же XSD-схему, то говорят, что представляющие их контракты о данных эквивалентны. Чтобы контракты о данных были эквивалентны, у них должны быть одинаковые пространства имен, имена и члены. Данные-члены должны иметь одинаковые типы и следовать в XML в одном и том же порядке. Короче говоря, при передаче по сети они должны быть неразличимы.

В листинге 2.26 приведены два эквивалентных контракта о данных. Первый объявлен службой, второй описан клиентом. Оба генерируют одинаковые XSD-схемы. По умолчанию на стороне службы WCF упорядочивает элементы XML по алфавиту, поэтому во второй схеме явно задан алфавитный порядок элементов. Поскольку в атрибутах `DataContract` и `DataMember` заданы соответственно поля `Name="StockPriceSvc"` и `Name="Currency"`, то XSD-описание второго контракта в точности идентично первому.

Листинг 2.26. Эквивалентные контракты о данных

```
[DataContract(Namespace = http://EssentialWCF)]
public partial class StockPriceSvc
{
   [DataMember] public double CurrentPrice;
   [DataMember] public DateTime CurrentTime;
   [DataMember] public string Ticker;
   [DataMember] public string Currency;
}

[DataContract(Namespace = http://EssentialWCF, Name="StockPriceSvc")]
public partial class StockPrice
{
   [DataMember(Order=4)] public string Ticker;
   [DataMember(Order=2)] public double CurrentPrice;
   [DataMember(Order=3)] public DateTime CurrentTime;
   [DataMember(Order=1, Name="Currency")] public string Money;
}
```

Работа с наборами

Наборы – это очень удобные структуры данных в .NET, сочетающие различные достоинства: динамическое выделение памяти, возможность обхода и навигации по списку. Однако ни в XSD, ни в WSDL нет эквивалентов наборам. Поэтому при сериализации WCF трактует наборы как массивы. На самом деле, в сериализованном представлении набор ничем не отличается от массива. Помимо наборов (типов, которые реализуют интерфейс `ICollection<T>`), то же самое справедливо для типов, реализующих интерфейсы `IEnumerable<T>` или `IList<T>`.

В листинге 2.27 приведен контракт о службе и операции, в котором используется набор. Этот набор снабжен атрибутом `[CollectionDataContract]`, специ-

ально предназначенным в WCF для этой цели. Этот атрибут заставляет WCF сериализовать любой тип, который поддерживает интерфейс `IEnumerable` и реализует метод `Add` для вставки в массив. Чтобы поддержать сериализацию, класс `StockPriceCollection` наследует обобщенному классу `List`, который реализует интерфейс `ICollection`.

Листинг 2.27. Поддержка набора в службе

```
using System;
using System.ServiceModel;
using System.Runtime.Serialization;
using System.Collections.Generic;

namespace EssentialWCF
{
   [DataContract(Namespace = "http://EssentialWCF")]
   public class StockPrice
   {
      [DataMember] public double CurrentPrice;
      [DataMember] public DateTime CurrentTime;
      [DataMember] public string Ticker;
   }

   [CollectionDataContract]
   public class StockPriceCollection : List<StockPrice>
   {
   }

   [ServiceContract]
   public class StockService
   {
      [OperationContract]
      private StockPriceCollection
                        GetPricesAsCollection(string[] tickers)
      {
         StockPriceCollection list = new StockPriceCollection();
         for (int i = 0; i < tickers.GetUpperBound(0) + 1; i++)
         {
            StockPrice p = new StockPrice();
            p.Ticker = tickers[i];
            p.CurrentPrice = 94.85;
            p.CurrentTime = System.DateTime.Now;
            list.Add(p);
         }
         return list;
      }
   }
}
```

Контракты о сообщениях

Контракты о сообщениях описывают структуру SOAP-сообщений, получаемых и отправляемых службой, и позволяет просматривать и управлять большинством деталей заголовка и тела SOAP. Если контракты о данных обеспечивают

возможность интероперабельности по стандарту XML Schema Definition (XSD), то контракты о сообщениях позволяют общаться с любой системой, которая поддерживает спецификацию SOAP.

Применение контрактов о сообщениях дает вам полный контроль над SOAP-сообщениями, поскольку вы получаете прямой доступ к заголовку и телу. Это позволяет использовать простые или составные типы для точного определения содержимого всех частей SOAP-сообщения. Если для сериализации данных вы можете пользоваться любым из классов `DataContractSerializer` или `XmlSerializer`, то для управления SOAP-сообщением в вашем распоряжении классы `DataContracts` и `MessageContracts`.

Передавать информацию в заголовках SOAP полезно, когда вы хотите сообщить что-то, не укладывающееся в сигнатуры операций. Например, информацию о сеансе или корреляции сообщений можно передавать в заголовках, а не заводить для этой цели дополнительные параметры операций или поля в самих данных. Другой пример – безопасность, когда вы реализуете протокол защиты самостоятельно (в обход спецификации WS-Security) и передаете верительные грамоты или маркеры в заголовках SOAP. Третий пример тоже связан с безопасностью и касается подписания или шифрования всей или некоторой информации в заголовках SOAP. Все эти задачи решаются с помощью контрактов о сообщениях. Впрочем, у этой техники есть недостаток – клиент и служба должны вручную помещать в заголовок SOAP и извлекать из него информацию, не полагаясь на сериализацию, которую обеспечивают контракты о данных и операциях применительно к классам.

Структуру SOAP-сообщений определяет атрибут `[MessageContract]`. У него не очень много модификаторов, поскольку цель – определить границы сообщения, а не его содержимое. Модификаторы определяют лишь, сколько тел упаковывается в одно SOAP-сообщение, надо ли выполнять обертывание и, если да, то какое выбрать имя и пространство имен для обертки.

Для типизированных сообщений применяются атрибуты `[MessageHeader]` и `[MessageBodyMember]`, которые описывают структуру заголовка и тела SOAP. Клиент и служба могут затем ссылаться на данные, пользуясь сериализованными типами. С заголовками можно ассоциировать дополнительную информацию, например имя и пространство имен, надо ли переправлять сообщение и кто является конечным исполнителем или получателем сообщения. С телом также можно ассоциировать дополнительную информацию, например имя и пространство имен. Если используется несколько тел, то в атрибуте `MessageContract` можно задать порядок частей. И для заголовка, и для тела можно задавать простые или составные определения типов.

В случае нетипизированных сообщений для описания содержимого не используются никакие атрибуты. Интерпретация содержимого оставляется целиком на усмотрение исполняющего кода. Это очень полезно, когда нужно напрямую работать с информационным набором XML-сообщения, так чтобы WCF не мешала манипулировать объектной моделью документа непосредственно. Операции службы, работающие с нетипизированными сообщениями, принимают и возвращают данные типа `message`, реализующие XML Infoset.

Типизированные сообщения

В листинге 2.28 приведен контракт о типизированном сообщении `StockPrice`. Заголовок содержит простой тип `DateTime`, а тело – составной тип `PriceDetails`. Класс `PriceDetails` должен быть сериализуемым – за счет наличия атрибута `[DataContract]`, как в данном случае, либо атрибута `[Serializable]`. В этом примере есть только один заголовок и одно тело, но, вообще говоря, количество тех и других не ограничено.

Желание завести несколько заголовков или тел может возникнуть, если они потребляются разными слоями программного обеспечения на стороне клиента. Например, одному слою может потребоваться корреляционная информация из заголовка SOAP, чтобы соотнести запрос с ответом, а другому – идентификационная информация для правильной маршрутизации сообщения. В таком случае у двух заголовков будет разное назначение, поэтому объединять их в одну структуру не имеет смысла. Отметим, что операция службы принимает и отправляет сообщения определенных типов. При использовании контракта о сообщениях входные и выходные параметры должны быть сообщениями, помеченными атрибутом `[MessageContract]`. Точнее, операция должна иметь ровно один входной параметр и возвращать ровно один результат, причем тот и другой должны быть сообщениями. Объясняется это тем, что сообщения, получаемые и отправляемые операцией, непосредственно отображаются на SOAP-представления. Кроме того, нельзя смешивать программирование на основе сообщений и на основе параметров, поэтому невозможно передать операции в качестве входного аргумента DataContract и ожидать, что она вернет MessageContract, или наоборот. Можно смешивать типизированные сообщения с нетипизированными, но не контракты о данных с контрактами о сообщениях. Если вы попытаетесь это сделать, то получите исключение при генерации WSDL-документа службой.

Для генерации кода клиентского прокси-класса, который представляет типизированное сообщение в `[MessageContract]`, нужно отметить флажок Always Generate Message Contracts (Всегда генерировать контракты о сообщениях) в диалоговом окне, которое открывается при нажатии кнопки Advanced (Дополнительно) в окне Add Service Reference (см. рис. 2.9).

Альтернативно можно задать флаг `/messageContract` или `/mc` при вызове `svcutil.exe`. Он заставляет `svcutil.exe` генерировать прокси-класс с открытыми методами, принимающими типизированное сообщение, так что клиенты могут вызывать ориентированные на сообщения методы. Если вы запускаете `svcutil.exe` без флага `/mc` или выполняете операцию Add Service Reference, не отмечая флажок Always Generate Message Contracts, то генерируется прокси-класс, открытые методы которого принимают параметры, а внутри себя вызывают операции, основанные на сообщениях. В любом случае в сеть передается одно и то же XML-представление сообщения.

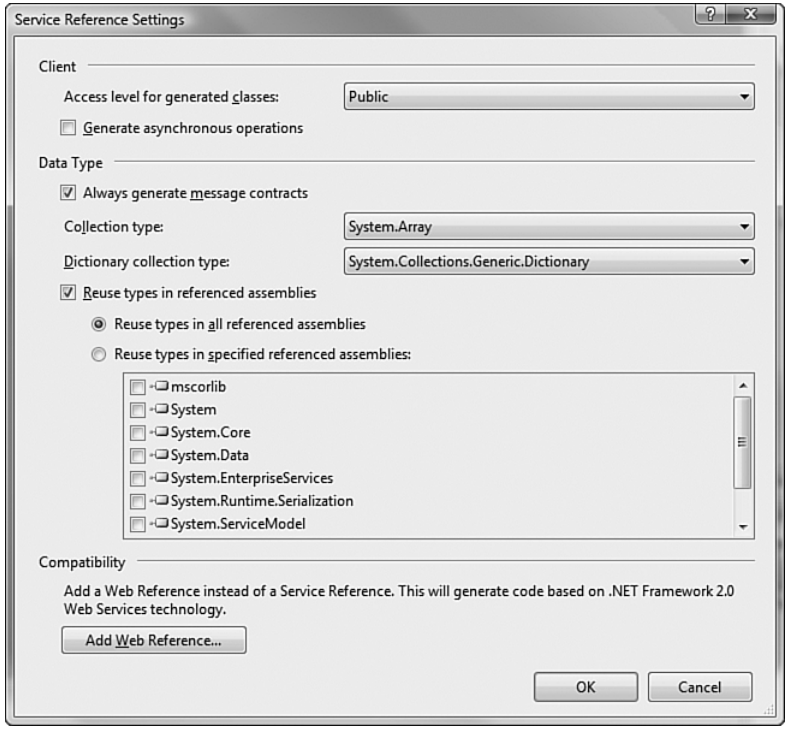

Рис. 2.9. Задание контрактов о сообщениях в окне Service Reference Setup

Листинг 2.28. Определение контракта о типизированном сообщении

```
namespace EssentialWCF
{
   [Serializable]
   public class PriceDetails
   {
      public string Ticker;
      public double Amount;
   }
   [MessageContract]
   public class StockPrice
   {
      [MessageHeader]
      public DateTime CurrentTime;
      [MessageBodyMember]
      public PriceDetails Price;
   }
```

```
[MessageContract]
public class StockPriceReq
{
   [MessageBodyMember] public string Ticker;
}

[ServiceContract]
public interface IStockService
{
   [OperationContract]
   StockPrice GetPrice(StockPriceReq req);
}

public class StockService : IStockService
{
   public StockPrice GetPrice(StockPriceReq req)
   {
      StockPrice resp = new StockPrice();
      resp.Price = new PriceDetails();
      resp.Price.Ticker = req.Ticker;
      resp.Price.Amount = 94.85;
      return resp;
   }
}
}
```

В листинге 2.29 приведен XML-документ, который передается в сеть, когда служба возвращает SOAP-сообщение клиенту. Обратите внимание, что элемент `CurrentTime`, помеченный атрибутом `[MessageHeader]`, находится в заголовке SOAP, а элемент `Price`, помеченный атрибутом `[MessageBodyMember]`, – в теле SOAP.

Листинг 2.29. SOAP-ответ, сгенерированный по контракту о типизированном сообщении

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Header>
      <h:CurrentTime xmlns:h="http://tempuri.org/">
                  2006-12-18T10:31:55.0584-05:00
      </h:CurrentTime>
   </s:Header>
   <s:Body>
      <StockPrice xmlns="http://tempuri.org/">
         <Price
               xmlns:a="http://schemas.datacontract.org/2004/07/EssentialWCF"
               xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
            <a:Amount>94.85</a:Amount>
            <a:Ticker>MSFT</a:Ticker>
         </Price>
      </StockPrice>
   </s:Body>
</s:Envelope>
```


Нетипизированные сообщения

В некоторых ситуациях структура сообщений, которыми обмениваются клиент и служба, неизвестна на этапе проектирования. Например, в сами сообщения может быть встроена какая-то логика; не исключено, что информация о маршруте и о применяемой операции определяется во время выполнения. Или между клиентом и службой находится некий слой программного (или аппаратного) обеспечения, который манипулирует SOAP-сообщениями и требует специальных форматов данных. В таких случаях бывают очень полезны нетипизированные контракты об операциях.

Нетипизированный контракт об операции позволяет клиенту и службе передавать в теле SOAP практически любую информацию при условии, что коммуникационный стек, указанный в привязке, способен закодировать содержимое. Содержимое сообщения непрозрачно для WSDL, поскольку не существует XSD-схемы, определяющей данные. При создании, чтении и записи сообщений клиент и служба работают с классом `Message`, который определен в пространстве имен `System.ServiceModel.Channels`.

В листинге 2.30 показан контракт об операции, в котором для описания входных и выходных данных используется тип `Message`. Отметим, что метод этого класса `GetBody` реализует обобщенную десериализацию тела сообщения в некоторый тип. Этот метод обращается к классу XMLReader для чтения элемента `<body>` в SOAP-сообщении. Класс XMLReader способен прочитать `<body>` только один раз; если вам необходимо читать его многократно, воспользуйтесь методом `CreateBufferedCopy` класс `Message`. Применяемое SOAP-действие описывается в запросе, его имя заканчивается словом «Response». Имя действия можно переопределить с помощью модификатора `(ReplyAction=)` в атрибуте `[OperationContract]`.

В классе `Message` имеются многочисленные методы для создания, чтения и записи содержимого сообщений. Клиент отвечает за создание сообщения перед отправкой его службе, а служба – за создание ответного сообщения. Прежде чем посылать сообщение, его содержимое нужно поместить в тело SOAP. Это позволяют сделать методы `CreateMessage`, `WriteMessage` или `WriteBody`.

Листинг 2.30. Определение и реализация контракта о нетипизированном сообщении

```
[ServiceContract (Namespace="http://EssentialWCF")]
public class StockService
{
   [OperationContract]
   private Message GetPrice(Message req)
   {
      string ticker = req.GetBody<String>();
      Message resp = Message.CreateMessage(
                        req.Version,
                        req.Headers.Action + "Response",
```

```
                        ticker + "|" + "94.85");
      return resp;
   }
}
```

Код клиента аналогичен коду службы; в нем тоже вызывается метод `CreateMessage` для создания сообщения подходящей версии, которая согласована с привязкой, а затем с помощью `GetBody` читается ответ, пришедший от службы. Отметим, что использованный вариант метода `CreateMessage` принимает три параметра: версию, действие и строковое сообщение. При создании сообщения проверяется, что его версия совместима со свойством `MessageVersion` канала в привязке, определяющей порядок коммуникации со службой. Действие, в данном случае `MessageVersion`, используется SOAP и инфраструктурой WCF для доставки сообщения подходящей операции службы. В листинге 2.31 приведен клиентский код, в котором инициируется обмен данными со службой, представленной в листинге 2.30.

Листинг 2.31. Инициирование клиентом обмена данными по контракту о нетипизированном сообщении

```
using (localhost.StockServiceClient proxy =
                  new localhost.StockServiceClient())
{
   new OperationContextScope(proxy.InnerChannel);
   Message msgReq = Message.CreateMessage(
         OperationContext.Current.OutgoingMessageHeaders.
                                       MessageVersion,
         "http:// EssentialWCF /StockService/GetPrice",
         "msft");
   Message msgResp = proxy.GetPrice(msgReq);
   Console.WriteLine("Returned {0} ", msgResp.GetBody<string>());
}
```

В листинге 2.32 показано SOAP-сообщение, переданное клиенту службой в ответ на запрос, представленный в листинге 2.31. Отметим, что имя действия в заголовке SOAP оканчивается словом «Response», а тело SOAP-сообщения – строка, не размеченная в соответствии с XML.

Листинг 2.32. SOAP-ответ, сгенерированный по контракту о нетипизированном сообщении

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Header>
      <Action s:mustUnderstand="1"
            xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">
               http://EssentialWCF/StockService/GetPriceResponse
      </Action>
   </s:Header>
   <s:Body>
      <string xmlns=
                  "http://schemas.microsoft.com/2003/10/Serialization/">
         msft|94.85
      </string>
   </s:Body>
</s:Envelope>
```


Использование заголовков SOAP в сочетании с нетипизированными сообщениями

Вне зависимости от того, работаете вы с типизированными или нетипизированными сообщениями, иногда хочется передать какую-то информацию не только в теле, но и в заголовке SOAP. Например, часто возникает необходимость передавать вместе с сообщением информацию о сеансе или контексте. Вместо того чтобы создавать дополнительные сообщения-обертки, проще и удобнее воспользоваться для этой цели заголовками SOAP.

Для типизированных сообщений WCF поддерживает этот механизм с помощью атрибута `[MessageHeader]`; это было продемонстрировано в листинге 2.28. А в случае нетипизированных сообщений вам придется добавить заголовок явно.

В листинге 2.33 приведен контракт о службе, который реализует операцию с нетипизированным сообщением и читает данные из заголовка сообщения. Отметим, что для получения значения `timeZone` из заголовка нужна всего одна строчка кода.

Листинг 2.33. Явное чтение данных из заголовка нетипизированного сообщения

```
[ServiceContract]
public class StockService
{
   [OperationContract]
   private Message GetPrice(Message req)
   {
      string timeZone =
                  OperationContext.Current.IncomingMessageHeaders.
                     GetHeader<String>
                              ("TimeZone", "http://EssentialWCF/");

      string ticker = req.GetBody<String>();
      Message resp = Message.CreateMessage(
                        req.Version,
                        req.Headers.Action + "Response",
                        timeZone + "|" + ticker + "|" + "94.85");
      return resp;
   }
}
```

В листинге 2.34 показано, как клиент может добавить заголовок SOAP к нетипизированному сообщению, отправляемому службе. Сначала с помощью метода `CreateMessage` создается объект `Message`, и конструктор помещает в него данные. Затем создается типизированный заголовок `MessageHeader` – в нашем случае это просто строка, и данные тоже заполняются конструктором. Далее из типи-

зированного заголовка создается нетипизированный заголовок `MessageHeader`, и последний добавляется в сообщение, отправляемое службе.

Листинг 2.34. Клиент вставляет заголовки в нетипизированное сообщение

```
static void Main(string[] args)
{
   using (localhost.StockServiceClient proxy
                  = new localhost.StockServiceClient())
   {
      new OperationContextScope(proxy.InnerChannel);
      Message msgReq =
                     Message.CreateMessage
                           (OperationContext.Current.
                           OutgoingMessageHeaders.MessageVersion,
                           "http://tempuri.org/StockService/GetPrice",
                           "msft");
      MessageHeader<String> msgHeader = new
                           MessageHeader<string>("GMT-05:00");
      MessageHeader untypedHeader =
                           msgHeader.GetUntypedHeader("TimeZone",
                                    "http://EssentialWCF/");
      msgReq.Headers.Add(untypedHeader);
      Message msgResp = proxy.GetPrice(msgReq);
   }
}
```

В листинге 2.35 приведено SOAP-сообщение, сгенерированное этим кодом. Обратите внимание, что в заголовок сообщения помещен элемент TimeZone в подходящем пространстве имен.

Листинг 2.35. Клиент вставляет заголовки в нетипизированное сообщение

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Header>
      <TimeZone xmlns="http://EssentialWCF/">GMT-05:00</TimeZone>
      <To s:mustUnderstand="1"
            xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">
            http://localhost/UntypedMessageHeader/StockService.svc
      </To>
      <Action s:mustUnderstand="1"
            xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/
none">
            http://tempuri.org/StockService/GetPrice
      </Action>
   </s:Header>
   <s:Body>
      <string
            xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
         msft
      </string>
   </s:Body>
</s:Envelope>
```


Резюме

В этой главе мы рассмотрели контракты, лежащие в основе интероперабельности. Контракт точно описывает сообщения, которые понимает служба.

В WCF определения контрактов используются повсеместно. Точнее, для описания оконечных точек службы применяется WSDL-документ, а для описания данных – XSD-схема. Находящиеся в WSDL определения операций службы используются для доставки входящих сообщений тому или иному классу .NET во время исполнения. Аналогично, данные, описанные в XSD-контрактах, десериализуются в типы .NET и во время выполнения передаются нужной операции службы. Совместно WSDL и XSD-описания составляют стандартизованное представление типов .NET, используемых внутри реализации службы.

Мы детально обсудили три вида контрактов:

- **Контракты о службе** описывают функциональные операции, реализуемые службой.
- **Контракты о данных** описывают структуры данных, необходимых для общения со службой. Согласно контракту о данных производится сериализация типов CLR в формат XML. Программист должен явно пометить атрибутами те члены, которые подлежат сериализации.
- **Контракты о сообщениях** позволяют работать с типизированными и нетипизированными данными и осуществлять точный контроль над заголовками и телами SOAP.

Контракты о службе

Контракт о службе определяет операции службы – раскрываемые внешнему миру методы из интерфейса реализующего ее класса. Контракт формально описывает интерфейс со службой на языке WSDL и формулируется с помощью атрибутов `[ServiceContract]` и `[OperationContract]`. Имена операций образуются из имен классов и методов, но могут быть переопределены в атрибутах. Контракт о службе поддерживает три способа обмена сообщениями: запрос-ответ, односторонний и дуплексный.

В случае обмена вида запрос-ответ клиент после отправки сообщения блокируется в ожидании ответа от службы. Поэтому такой способ следует применять только тогда, когда служба отвечает достаточно быстро, и пользователь готов немного подождать. На стороне клиента можно применить имеющийся в .NET механизм асинхронных вызовов, чтобы избежать блокировки в случае, когда запрос обрабатывается долго.

Односторонние операции не возвращают клиенту никакого результата. В коде службы они должны иметь тип возвращаемого значения `void` и помечаться модификатором `IsOneWay=true` в атрибуте `[OperationContract]`. Односторонний контракт можно реализовать поверх любого транспортного протокола, в том числе и MSMQ.

Дуплексные операции обеспечивают максимальную гибкость и производительность, поскольку разрывают связь между запросом и его обработкой. После

того как между клиентом и службой установлен дуплексный канал, инициировать обмен сообщениями может как клиент, так и служба. Этот способ идеально подходит для оповещения клиентов.

Контракты о данных

Контракт о данных описывает прикладные данные для службы. Классы, помеченные атрибутами `[DataContract]` и `[DataMember]`, включаются в виде XSD-схемы в WSDL-документ, описывающий контракт о службе. В том же документе можно объявить и другие типы данных, например, базовые типы и типы, помеченные атрибутом `[Serializable]`. Поскольку в атрибут `[DataContract]` уже встроены правила сериализации, спроектированные с учетом интероперабельности, то этот механизм является при работе с WCF предпочтительным.

В контракт о данных включаются только члены класса, помеченные атрибутом `[DataMember]`. И это делает `[DataContract]` более подходящим механизмом сериализации по сравнению с `[Serializable]`, так как последний может раскрывать внутренние структуры данных службы.

Поддерживаются иерархии классов, причем информация о пространствах имен сохраняется. Для поддержки полиморфизма и наборов, содержащих объекты, WCF разрешает службе публиковать список известных типов. Контракты о данных спроектированы с учетом возможного изменения версий. Когда в контракт добавляются новые члены, существующие клиенты продолжают работать при условии, что соблюдены определенные правила.

Контракты о сообщениях

Контракт о сообщениях – это контракт об операциях, который разрешает доступ к заголовкам и телам SOAP. Сообщения могут быть типизированы с помощью атрибутов `[DataContract]` или `[Serializable]` или просто принадлежать типу `Message`. Типизированные сообщения описываются атрибутами `[MessageHeader]` и `[MessageBody]`. Данные сообщения могут не задаваться на этапе проектирования, что обеспечивает высокую гибкость. Нетипизированные сообщения также имеют доступ к заголовкам и телам SOAP.

Глава 3. Каналы

Канал – это труба, по которой курсируют все сообщения, получаемые и отправляемые WCF-приложением. Он отвечает за подготовку и доставку сообщений единообразным способом. Определены каналы для транспорта, протоколов и перехвата сообщений. Каналы можно комбинировать, формируя *стек каналов*. Каждое сообщение обрабатывается всеми входящими в стек каналами. Например, можно сконструировать стек, содержащий транспортный канал TCP и канал транзакционного протокола. Такой стек будет передавать и принимать сообщения по протоколу TCP, сохраняя информацию о транзакциях.

Назначение стека каналов состоит в том, чтобы преобразовать передаваемое по сети сообщение в формат, совместимый с получателем и отправителем, и организовать его транспортировку. Для этого используются каналы двух видов: *транспортные* и *протокольные*. Транспортный канал всегда находится внизу стека и отвечает за транспортировку сообщений в соответствии с транспортным протоколом. WCF предоставляет широкое разнообразие транспортных каналов, включая HTTP, TCP, MSMQ, пиринговые (peer-to-peer) и именованные каналы (named pipe). Протокольные каналы располагаются поверх транспортных и прочих каналов; они трансформируют и модифицируют сообщения. WCF поддерживает много разных протокольных каналов, например, для обеспечения безопасности, транзакционной целостности и надежной передачи.

Транспортные каналы. WCF изначально предоставляет несколько транспортных каналов, включая HTTP, TCP, MSMQ, пиринговые и именованные каналы. В примерах и продуктах сторонних фирм реализованы и другие транспорты, например: SMTP, FTP, UDP, WebSphere MQ и SQL Service Broker. Многие из них можно найти на сайте http://wcf.netfx3.com. Реализация транспортного канала для протокола UDP имеется в Windows SDK. Транспортный канал для WebSphere MQ доступен на сайте alphaWorks компании IBM.

Чтобы начать обмен данными, клиент и сервер должны сформировать совместимые стеки каналов. В случае, когда на обоих концах работают приложения .NET, это обычно делается путем создания идентичных стеков на стороне клиента и сервера. А в общем случае требуется, чтобы функциональные возможности стеков были согласованы. Чтобы упростить создание стеков каналов, мы пользуемся *привязками*. Привязка описывает конфигурацию стека и знает о том, как создать его на этапе выполнения. Каждая привязка составляется из набора *элементов привязки*, которые обычно представляют отдельные каналы в стеке. Привязки и их элементы подробно рассматриваются в главе 4.

Архитектура каналов в WCF обеспечивает высочайшую гибкость за счет того, что способ коммуникации абстрагируется от самого приложения. Это дает разработчикам возможность конструировать службы, доступные с помощью различных механизмов коммуникации, и тем самым видоизменять их вместе с изменением требований. Например, WCF-служба, первоначально работавшая в контексте приложений .NET, легко может быть раскрыта приложению, написанному на Java, без какой бы то ни было модификации. По мере изменения требований WCF-службу можно без труда расширить, добавив такие функции, как интероперабельность, надежная доставка сообщений и транзакционность. В предыдущих технологиях Microsoft (ASP.NET Web Services, .NET Remoting, Enterprise Services или MSMQ) требовалось переписывать части приложения для каждой новой формы коммуникации. WCF позволяет выбирать нужные возможности без существенной модификации приложения.

Такой гибкостью WCF обязана механизму послойной сборки стека каналов. На рис. 3.1 показано, как сообщение, отправленное WCF-приложением, проходит по стеку каналов на стороне клиента и доставляется выбранным транспортом серверу. Серверный стек каналов обнаруживает поступление сообщений и передает его серверному приложению.

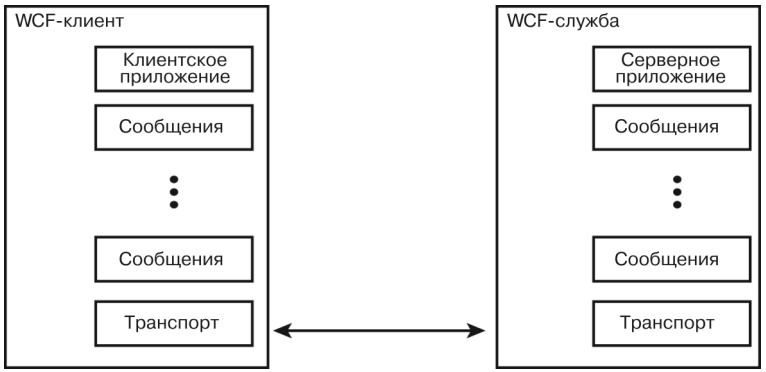

Рис. 3.1. Стек каналов

Стек каналов представляет собой последовательность каналов, сконфигурированных с помощью элементов привязки. Заранее сконфигурированный стек также называют привязкой. Привязка состоит из нескольких элементов привязки, так же как стек состоит из каналов. На вершине стека располагаются протокольные каналы. Протокольные каналы взаимодействуют с сообщением и обеспечивают безопасность, надежную доставку, транзакционность и протоколирование. В стеке может быть столько протокольных каналов, сколько необходимо для конкретной задачи.

Транспортные каналы отвечают за передачу байтов в соответствии с конкретным протоколом передачи, например TCP или HTTP. Кроме того, на них ложится ответственность за выбор *кодировщика*, который преобразует сообщение в массив байтов. Именно кодировщик восстанавливает сообщение из XML-представления в массив байтов. О том, какой кодировщик использовать, транспортному каналу сообщает элемент привязки. Транспортный канал ищет в контексте привязки реализацию класса `MessageEncoder`. Если ее нет, канал может выбрать кодировщик сообщения по умолчанию.

В стеке каналов есть транспортный канал и кодировщик. В любом стеке каналов должен быть по меньшей мере один транспортный канал и один кодировщик. Обычно транспортный канал знает, какую кодировку применять по умолчанию. Примером может служить транспортный канал `tcpTransport`, для которого по умолчанию используется кодировщик `binaryMessageEncoding`. Это тот минимум, который необходим для реализации стека каналов в WCF. Протокольные каналы необязательны.

Канальные формы

WCF поддерживает три разных способа обмена сообщениями: односторонний, дуплексный и запрос-ответ. Для их поддержки WCF предоставляет десять интерфейсов, которые называют канальными формами (channel shape). Пять из них таковы: `IOutputChannel`, `IInputChannel`, `IDuplexChannel`, `IRequestChannel` и `IReplyChannel`. У каждой из этих форм есть аналог для поддержки сеансов: `IOutputSessionChannel`, `IInputSessionChannel`, `IDuplexSessionChannel`, `IRequestSessionChannel` и `IReplySessionChannel`. Набор этих интерфейсов позволяет реализовать в стеке каналов различные способы коммуникации. В настоящем разделе мы рассмотрим все способы и ассоциированные с ними интерфейсы.

Односторонняя коммуникация

При таком способе сообщения посылаются только в одном направлении – от клиента к серверу. Односторонняя коммуникация обычно применяется, когда клиенту не нужна никакая ответная информация, достаточно подтверждения того, что сообщение было отправлено. Как только сообщение отправлено, коммуникация прекращается. Для реализации односторонней коммуникации предназначены интерфейсы `IOutputChannel` и `IInputChannel`. На рис. 3.2 показан поток сообщений между клиентом и сервером в случае односторонней коммуникации.

Интерфейс `IOutputChannel` отвечает за отправку сообщений, а интерфейс `IInputChannel` – за прием. В листинге 3.1 показано клиентское приложение, в котором для отправки сообщения применяется канальная форма `IOutputChannel`.

Листинг 3.1. Пример использования IOutputChannel

```
using System;
using System.Collections.Generic;
```

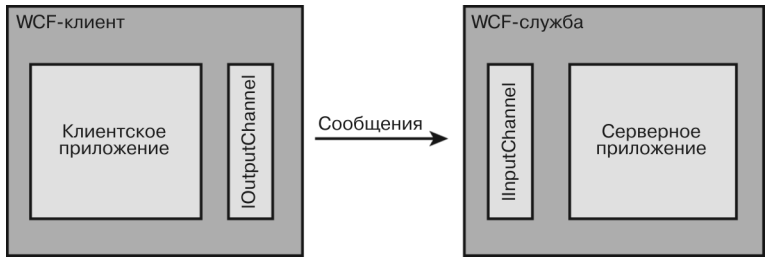

Рис. 3.2. Односторонняя коммуникация

```
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Text;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         BasicHttpBinding binding = new BasicHttpBinding();
         BindingParameterCollection parameters =
               new BindingParameterCollection();

         Message m =
               Message.CreateMessage(MessageVersion.Soap11, "urn:sendmessage");
         IChannelFactory<IOutputChannel> factory =
               binding.BuildChannelFactory<IOutputChannel>(parameters);
         IOutputChannel channel = factory.CreateChannel(
               new EndpointAddress("http://localhost/sendmessage/"));
         channel.Send(m);
         channel.Close();
         factory.Close();
      }
   }
}
```

Дуплексная коммуникация

При дуплексной коммуникации используются две односторонних канальных формы, объединенных в третий интерфейс, который называется `IDuplexChannel` (рис. 3.3). Преимущество дуплексной коммуникации по сравнению с односторонней и запрос-ответ состоит в том, что обмен сообщениями может инициироваться как клиентом, так и сервером.

Примером дуплексной коммуникации может служить система извещения о событиях. Сервер посылает информацию о событиях, а клиент получает ее. Клиент предоставляет оконечную точку, в которую сервер может посылать сообщения. В листинге 3.2 приведен пример клиента, который пользуется канальной формой `IDuplexChannel`.

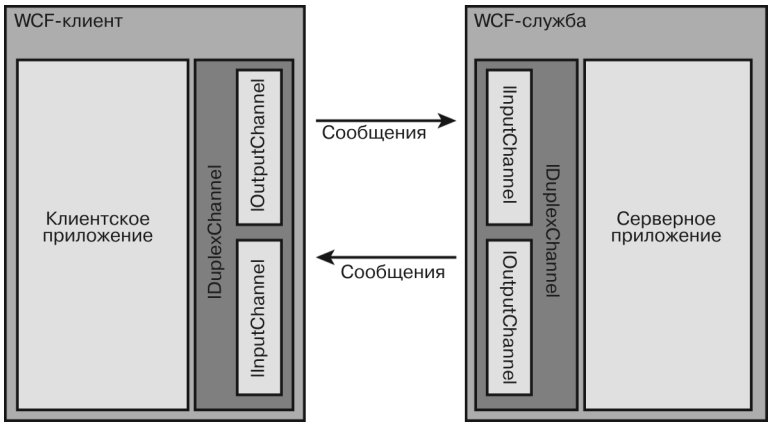

Рис. 3.3. Дуплексная коммуникация

Листинг 3.2. Пример использования IDuplexChannel

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Text;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         NetTcpBinding binding = new NetTcpBinding();
         BindingParameterCollection parameters =
            new BindingParameterCollection();

         Message m =
         Message.CreateMessage(MessageVersion.Soap12WSAddressing10,
            "urn:sendmessage");
         IChannelFactory<IDuplexChannel> factory =
            binding.BuildChannelFactory<IDuplexChannel>(parameters);
         IDuplexChannel channel = factory.CreateChannel(
            new EndpointAddress("net.tcp://localhost/sendmessage/"));
         channel.Send(m);
         channel.Close();
         factory.Close();
      }
   }
}
```

Коммуникация запрос-ответ

Коммуникация запрос-ответ – это частный случай двусторонней коммуникации, когда на каждый запрос поступает ровно один ответ. Она всегда инициирует-

ся клиентом. Послав запрос, клиент должен дождаться ответа, прежде чем посылать следующий запрос.

Типичный случай коммуникации запрос-ответ – это диалог между браузером и Web-сервером по протоколу HTTP. Браузер посылает серверу GET или POST-запрос, сервер его обрабатывает и посылает ответ. В WCF для коммуникации запрос-ответ используются интерфейсы `IRequestChannel` и `IReplyChannel`, как показано на рис. 3.4.

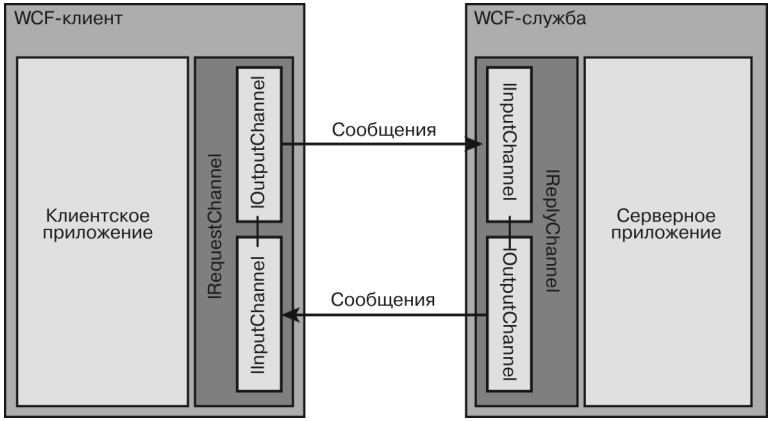

Рис. 3.4. Коммуникация запрос-ответ

В листинге 3.3 приведен пример клиентского приложения, которое пользуется интерфейсом `IRequestChannel` для отправки сообщения. Отметим, что метод `Request` возвращает полученное ответное сообщение.

Листинг 3.3. Пример использования IRequestChannel

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Text;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         BasicHttpBinding binding = new BasicHttpBinding();
         BindingParameterCollection parameters =
               new BindingParameterCollection();

         Message request =
               Message.CreateMessage(MessageVersion.Soap11, "urn:sendmessage");
```



```
         IChannelFactory<IRequestChannel> factory =
               binding.BuildChannelFactory<IRequestChannel>(parameters);
         IRequestChannel channel = factory.CreateChannel(
                  new EndpointAddress("http://localhost/sendmessage/"));
         Message response = channel.Request(request);
         channel.Close();
         factory.Close();
      }
   }
}
```

Изменение формы

Сама природа протокола HTTP предполагает диалог вида запрос-ответ, поэтому в транспортном канале HTTP применяется именно эта канальная форма. При других способах коммуникации по протоколу HTTP, скажем односторонней или дуплексной, форму следует изменить. Для этого в стек поверх транспортного канала помещается соответствующий протокольный канал. В листинге 3.4 показана заказная привязка, в которой элемент `OneWayBindingElement`, изменяющий канальную форму на одностороннюю, расположен над транспортом HTTP. В главе 12 мы познакомимся с более сложными примерами изменения формы с помощью элемента привязки `CompositeDuplexBindingElement`.

Листинг 3.4. Пример использования элемента OneWayBindingElement

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Text;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         CustomBinding binding = new CustomBinding(
               new OneWayBindingElement(),
               new TextMessageEncodingBindingElement(),
               new HttpTransportBindingElement());
         }
   }
}
```

Контракт об операциях и канальные формы

Канальные формы используются каналами для реализации различных способов обмена сообщениями. Например, транспортный канал на базе TCP мог бы реализовать интерфейсы `IInputChannel` и `IOutputChannel`, потому что эти транспорты принципиально односторонние. Другие транспортные каналы, основанные

на протоколах, подобных TCP, могут реализовывать несколько канальных форм. Разработчики не работают с каналами напрямую. WCF выбирает канальную форму, ориентируясь на атрибут `OperationContract` службы. В таблице 3.1 перечислены различные модификаторы этого атрибута и соответствующие им канальные формы. Отметим, что у большинства канальных форм есть безсеансовый (по умолчанию) и сеансовый варианты. Сеансовый канал передает некий идентификатор от клиента серверу. Этим можно воспользоваться для отслеживания состояния сеанса. Именно так организовано управление состоянием в ASP.NET. В WCF не встроена функция управления состоянием, но вы можете реализовать ее самостоятельно, пользуясь сеансами в сочетании с порождением экземпляров. Управление экземплярами описывается в главе 5 «Поведения».

Таблица 3.1. Выбор канальной формы в зависимости от модификаторов атрибута OperationContract

OneWay	Request/Reply	Session	Callback	Канальная форма
Любой	Любой	No	Yes	IDuplexChannel
Любой	Любой	No	Yes	IDuplexSessionChannel
Любой	Любой	Yes	Yes	IDuplexSessionChannel
Yes	Yes	No	No	IDuplexChannel
Yes	Yes	No	No	IRequestChannel
Yes	Yes	No	No	IDuplexSessionChannel
Yes	Yes	Yes	No	IDuplexSessionChannel
Yes	Yes	Yes	No	IRequestSessionChannel
Yes	No	No	No	IOutputChannel
Yes	No	No	No	IDuplexChannel
Yes	No	No	No	IDuplexSessionChannel
Yes	No	No	No	IRequestChannel
Yes	No	Yes	No	IOutputSessionChannel
Yes	No	Yes	No	IDuplexSessionChannel
Yes	No	Yes	No	IRequestSessionChannel
No	Yes	No	No	IRequestChannel
No	Yes	No	No	IDuplexChannel
No	Yes	No	No	IDuplexSessionChannel
No	Yes	Yes	No	IRequestSessionChannel
No	Yes	Yes	No	IDuplexSessionChannel

Не каждый канал реализует все эти интерфейсы. Если канал не поддерживает конкретную форму, то WCF попытается адаптировать существующую канальную форму под свои нужды. Например, если канал не реализует интерфейсы `IInputChannel` и `IOutputChannel` для односторонней коммуникации, то WCF попробует использовать либо `IDuplexChannel`, либо `IRequestChannel/IReplyChannel`.

Прослушиватели каналов

Прослушиватели каналов – это основа серверных коммуникаций в WCF. На них возлагается ожидание входящих сообщений, создание стека каналов и предоставление приложениям ссылки на вершину стека. Прослушиватель получает со-

общения от транспортного канала или канала, расположенного ниже него в стеке. Разработчики как правило не работают с прослушивателями каналов напрямую, а пользуются для размещения служб классом `ServiceHost`, который уже прибегает к помощи прослушивателя для ожидания сообщений. Подробнее о классе `ServiceHost` см. главу 7 «Размещение». Метод привязки `BuildChannelListener` конструирует прослушиватель канала, исходя из заданной канальной формы. В примере ниже мы используем привязку `BasicHttpBinding` и канальную форму `IReplyChannel`.

Листинг 3.5. Использование прослушивателя канала

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         BasicHttpBinding binding = new
               BasicHttpBinding(BasicHttpSecurityMode.None);

         Uri address = new Uri("http://localhost/request");
         BindingParameterCollection bpc = new
               BindingParameterCollection();

         Console.WriteLine("Запускается служба...");
         IChannelListener<IReplyChannel> listener =
               binding.BuildChannelListener<IReplyChannel>(address, bpc);
         listener.Open();
         IReplyChannel channel = listener.AcceptChannel();
         channel.Open();
         Console.WriteLine("Служба запущена!");

         Console.WriteLine("Ожидание запроса...");
         RequestContext request = channel.ReceiveRequest();
         Message message = request.RequestMessage;
         string data = message.GetBody<string>();
         Message replymessage =
         Message.CreateMessage(message.Version,
                               "http://localhost/reply",
                               data);
         request.Reply(replymessage);
         Console.WriteLine("Служба остановлена!");

         message.Close();
         request.Close();
         channel.Close();
```

```
            listener.Close();

            Console.ReadLine();
        }
    }
}
```

Фабрики каналов

Фабрика каналов создает канал для отправки сообщений и становится владельцем созданных каналов. Разработчики обычно не работают с фабрикой каналов напрямую, а пользуются классом, производным от `ClientBase<>`, который как правило генерируется утилитой `svcutil.exe` или операцией Add Service Reference в Visual Studio. Однако понимать, что такое фабрики каналов, важно, так как в WCF они лежат в основе коммуникаций на стороне клиента.

Фабрика каналов владеет созданными каналами. Между прослушивателями и фабриками каналов имеется важное различие. Фабрика отвечает за закрытие всех созданных ей каналов, у прослушивателя такой обязанности нет. Так сделано для того, чтобы прослушиватель канала можно было остановить независимо от ассоциированных с ним каналов.

В листинге 3.6 демонстрируется использование фабрики каналов для вызова службы. Это клиент сервера, представленного в листинге 3.5. Для создания канала мы обращаемся к методу `CreateChannel` привязки.

Листинг 3.6. Использование фабрики каналов

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         BasicHttpBinding binding = new
               BasicHttpBinding(BasicHttpSecurityMode.None);

         IChannelFactory<IRequestChannel> factory =
               binding.BuildChannelFactory<IRequestChannel>(
                     new BindingParameterCollection());
         factory.Open();
         IRequestChannel channel = factory.CreateChannel(
                     new EndpointAddress("http://localhost/request"));
         channel.Open();

         Message requestmessage = Message.CreateMessage(
```



```
               MessageVersion.Soap11,
                           "http://contoso.com/reply",
                           "Это данные в теле сообщения");

         Console.WriteLine("Посылается сообщение...");
         Message replymessage = channel.Request(requestmessage);
         string data = replymessage.GetBody<string>();
         Console.WriteLine("Получен ответ!");
         requestmessage.Close();
         replymessage.Close();
         channel.Close();
         factory.Close();

         Console.ReadLine();
      }
   }
}
```

Класс ChannelFactory<>

В WCF к фабрикам каналов имеют отношение два класса: `ChannelFactory` и `ChannelFactory<>`. Хотя они похожи, но все же это два разных класса, предназначенных для решения различных задач. Класс `ChannelFactory<>` применяется в нетривиальных ситуациях, когда нужно создать несколько клиентов. По существу он работает с заданной фабрикой `ChannelFactory`, но не отвечает за создание стека каналов. Для использования `ChannelFactory<>` необходимо определить класс типа `ServiceContract`. В листинге 3.7 приведен пример использования класса `ChannelFactory<>` для обращения к службе, которая реализует интерфейс `IStockQuoteService`.

Предложение using и класс `ChannelFactory<>`. Будьте осторожны, применяя предложение using для закрытия `ChannelFactory`. В листинге 3.7 показан рекомендуемый способ, когда обращение к службе заключается в скобки `try..catch`, чтобы точно знать, какую ошибку вернула служба. Если опустить операторные скобки, то все исключения просочатся из предложения using наружу. В этот момент фабрика каналов возбудит исключение, потому что она уже закрыта. Тем самым будет замаскировано предыдущее исключение, возбужденное службой. Мы включили два блока `try..catch`, чтобы перехватывать все ошибки, возникшие в результате вызова службы.

Листинг 3.7. Использование класса ChannelFactory<>

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{
   class Program
   {
```

```
      static void Main(string[] args)
      {
         try
         {
            using (ChannelFactory<IStockQuoteService> cf =
                  new ChannelFactory<IStockQuoteService>())
            {
               IStockQuoteService service = cf.CreateChannel();

               try
               {
                  double value = service.GetQuote("MSFT");
               }
               catch (Exception ex)
               {
                  // Перехватываем исключение, возникшее внутри GetQuote
                  Console.WriteLine(ex.ToString());
               }
            }
         }
         catch (Exception ex)
         {
            // Перехватываем исключение, возникшее при создании канала
            Console.WriteLine(ex.ToString());
         }
         Console.ReadLine();
      }
   }
}
```

Интерфейс ICommunicationObject

Интерфейс `ICommunicationObject` (см. листинг 3.8) – основа всех коммуникационных объектов в WCF (каналов, фабрик каналов, прослушивателей каналов и т.д.). Разработчики, планирующие создавать нестандартные каналы или работать с каналами напрямую, должны хорошо понимать его назначение. Коммуникационные объекты в WCF должны реализовывать специальный конечный автомат, описывающий все состояния, через которые проходит коммуникационный объект. Тут имеется прямая аналогия с подходом к другим коммуникационным объектам (например, сокетам). Интерфейс `ICommunicationObject` (его методы, свойства и события) предназначен для реализации этого конечного автомата. Он позволяет WCF трактовать все коммуникационные объекты единообразно, абстрагируясь от деталей реализации.

Листинг 3.8. Интерфейс ICommunicationObject

```
public interface ICommunicationObject
{
   // События
   event EventHandler Closed;
   event EventHandler Closing;
   event EventHandler Faulted;
   event EventHandler Opened;
   event EventHandler Opening;
```



```
   // Методы
   void Abort();
   IAsyncResult BeginClose(AsyncCallback callback, object state);
   IAsyncResult BeginClose(TimeSpan timeout,
            AsyncCallback callback, object state);
   IAsyncResult BeginOpen(AsyncCallback callback, object state);
   IAsyncResult BeginOpen(TimeSpan timeout,
            AsyncCallback callback, object state);
   void Close();
   void Close(TimeSpan timeout);
   void EndClose(IAsyncResult result);
   void EndOpen(IAsyncResult result);
   void Open();
   void Open(TimeSpan timeout);

   // Свойства
   CommunicationState State { get; }
}
```

В листинге 3.9 показаны состояния, определенные в перечислении `CommunicationState`.

Листинг 3.9. Перечисление CommunicationState

```
public enum CommunicationState
{
   Created,
   Opening,
   Opened,
   Closing,
   Closed,
   Faulted
}
```

В перечислении `CommunicationState` определено шесть состояний, через которые проходят коммуникационные объекты. Любой такой объект начинает существование в состоянии `Created`. Конечным состоянием для любого коммуникационного объекта является `Closed`. В течение времени жизни объекта для него вызываются методы интерфейса `ICommunicationObject`, которые переводят объект из одного состояния в другое. Например, метод `Open()` переводит коммуникационный объект из состояния `Created` в состояние `Opened`. На рис. 3.5 изображена диаграмма состояний, на которой видны состояния коммуникационного объекта и переходы между ними.

Примером коммуникационного объекта может служить класс `ClientBase<>`, который является базовым для всех клиентов, сгенерированных с помощью операции Add Service Reference в Visual Studio или утилиты `svcutil.exe`.

Повторно использовать клиентов нельзя. После того как коммуникационный объект перешел из состояния `Opened` в состояние `Closing` или `Faulted`, обратного пути нет. Иначе говоря, вернуться в состояние `Opened` невозможно без повторного создания объекта. Поэтому, коль скоро клиент закрыт (находится в состоянии `Closed`), его надо создавать заново.

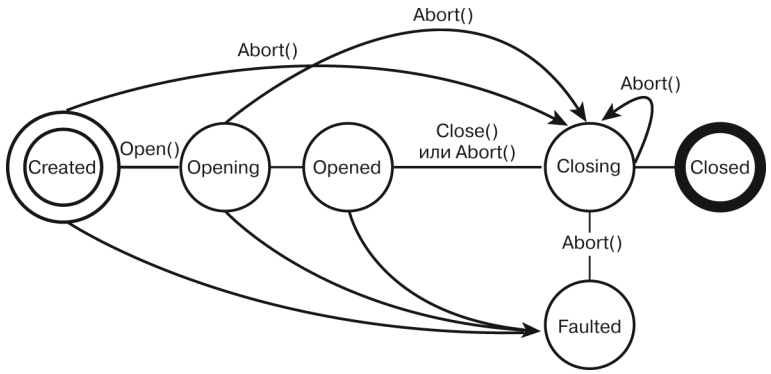

Рис. 3.5. Диаграмма состояний ICommunicationObject

В интерфейсе `ICommunicationObject` определены пять событий (`Opening`, `Opened`, `Closing`, `Closed` и `Faulted`). Они служат для извещения программы о переходах состояний.

Извещения клиентов. Приложения часто хранят ссылку на клиентский прокси-класс. В таких случаях важно использовать события перехода состояний, чтобы знать, когда прокси входит в состояние `Faulted` (и в конечном итоге в состояние `Closed`), чтобы поддержать дальнейшее взаимодействие между клиентом и сервером.

Обычно коммуникационный объект приводится к интерфейсу `ICommunicationObject`, чтобы получить доступ к его методам и событиям. Но иногда необходимо создать новый коммуникационный объект, расширяющий возможности, встроенные в WCF. Для таких случаев WCF предоставляет абстрактный класс `CommunicationObject`, который реализует интерфейс `ICommunicationObject` и ассоциированный с ним конечный автомат. В листинге 3.10 демонстрируется работа с классом `StockQuoteServiceClient`, сгенерированным с помощью `svcutil.exe`. Он наследует классу `ClientBase<>`. Показано, что класс клиента приводится к интерфейсу `ICommunicationObject`, чтобы можно было получать коммуникационные события.

Листинг 3.10. Пример использования интерфейса ICommunicationObject

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         string symbol = "MSFT";
         double value;

         StockQuoteServiceClient client =
               new StockQuoteServiceClient());
         ICommunicationObject commObj =
            (ICommunicationObject)client;

         commObj.Closed += new EventHandler(Closed);
         commObj.Faulted += new EventHandler(Faulted);
         value = client.GetQuote(symbol);

         Console.WriteLine("{0} @ ${1}", symbol, value);
         Console.ReadLine();
      }

      static void Closed(object sender, EventArgs e)
      {
         // Обработать событие Closed
      }

      static void Faulted(object sender, EventArgs e)
      {
         // Обработать событие Faulted
      }
   }
}
```


Резюме

Стек каналов – это многоуровневый коммуникационный стек, составленный из одного или нескольких каналов, обрабатывающих сообщения. Каналы бывают транспортными и протокольными. Транспортные каналы располагаются внизу стека и отвечают за передачу сообщений с помощью некоторого транспортного механизма (например, HTTP, TCP, MSMQ). Протокольные каналы реализуют различные протоколы (безопасности, надежной доставки, транзакционности и т.д.) посредством трансформации и модификации сообщений.

Фабрики и прослушиватели каналов лежат в основе приема и передачи сообщений. Они отвечают за создание стека каналов и предоставление его приложениям.

WCF умело скрывает детали модели каналов от разработчиков. Большинству разработчиков достаточно класса, производного от `ClientBase<>`, для отправки сообщений и класса `ServiceHost` для размещения служб. Эти классы надстроены над архитектурой каналов.

Архитектура каналов лежит в основе всех коммуникаций в WCF. Разобравшись с ее составными частями – стеками каналов, каналами, фабриками и прослушивателями каналов, – разработчик может расширить модель коммуникаций или подстроить ее под свои нужды.

Глава 4. Привязки

В главе 3 было сказано, что стек каналов – это многоуровневый коммуникационный стек, составленный из одного или более каналов, обрабатывающих сообщения. *Привязкой* называется заранее сконфигурированный стек каналов. Привязки описывают соглашения между клиентом и сервером о порядке передачи данных по сети. В привязке задается способ транспортировки, кодирование и протоколы, участвующие в коммуникации. WCF с помощью привязок инкапсулирует конфигурацию в различных сценариях коммуникации. Для наиболее распространенных сценариев – Web-служб, служб REST/POX и приложений на основе очередей – имеются уже готовые привязки. Например, привязка `basicHttpBinding` презназначена для работы с Web-службами, созданными в ASP.NET или совместимыми со спецификацией WS-I Basic Profile 1.1. Привязки `ws2007HttpBinding` и `wsHttpBinding` похожи на `basicHttpBinding`, но поддерживают больше возможностей, в частности надежную доставку и транзакции, а также основаны на более современных стандартах, таких, как WS-Addressing. В таблице 4.1 перечислены 12 привязок, применяемых в разных сценариях коммуникации.

Таблица 4.1. Коммуникационные привязки WCF в версии .NET Framework 3.5

Имя привязки	Описание	.NET Framework
`basicHttpBinding`	Привязка для Web-служб, совместимых с WS-I Basic Profile 1.1, в частности для ASMX-служб	3.0/3.5
`wsHttpBinding`	Привязка для Web-служб с дополнительными возможностями, в частности WS-Security, WS-Transactions и т.п.	3.0/3.5
`wsDualHttpBinding`	Привязка для поддержки двусторонней коммуникации с использованием дуплексных контрактов	3.0/3.5
`webHttpBinding`	Привязка для поддержки REST/POX-служб с использованием сериализации в форматах XML и JSON	3.0/3.5
`netTcpBinding`	Привязка для коммуникаций между двумя .NET-системами	3.0/3.5
`netNamedPipeBinding`	Привязка для коммуникаций в рамках одной машины или между несколькими .NET-системами	3.0/3.5

Таблица 4.1. Коммуникационные привязки WCF в версии .NET Framework 3.5 (окончание)

Имя привязки	Описание	.NET Framework
`netMsmqBinding`	Привязка для асинхронных коммуникаций с использованием Microsoft Message Queue (MSMQ)	3.0/3.5
`netPeerTcpBinding`	Привязка для построения приложений в пиринговых сетях	3.0/3.5
`msmqIntegration Binding`	Привязка для отправки и получения сообщений приложениям с помощью очередей MSMQ	3.0/3.5
`wsFederationHttp Binding`	Привязка для продвинутых Web-служб, совместимых со стандартами WS-*, с помощью интегрированной идентификации (federated identity)	3.0/3.5
`ws2007HttpBinding`	Привязка, производная от `wsHttpBinding`, с дополнительной поддержкой самых последних спецификаций WS-*, вышедших в 2007 году	3.5
`ws2007Federation HttpBinding`	Привязка, производная от `wsFederationHttpBinding`, с дополнительной поддержкой самых последних спецификаций WS-*, вышедших в 2007 году	3.0/3.5

Привязки, перечисленные в таблице 4.1, можно задавать как в коде, так и в конфигурационном файле. В листинге 4.1 показано, как привязка `basicHttpBinding` задается в конфигурационном файле. Задавая привязку таким образом, разработчик может затем заменить ее на другую или модифицировать параметры, не перекомпилируя приложение.

Листинг 4.1. Задание привязки в конфигурационном файле

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="http://localhost/helloworld"
                   binding="basicHttpBinding"
                   contract="EssentialWCF.HelloWorld">
         </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```

В листинге 4.2 показано использование класса `BasicHttpBinding`. Задавая конкретную привязку в коде, разработчик не дает изменить ее впоследствии.

Листинг 4.2. Задание привязки в коде

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         BasicHttpBinding binding = new BasicHttpBinding();

         using (HelloWorldClient client =
               new HelloWorldClient(Binding,
                     "http://localhost/helloworld"))
            client.SayHello("Rich");

            Console.ReadLine();
         }
   }
}
```

Для составления стека каналов с помощью привязок используется набор *элементов привязки*. Элемент привязки представляет канальный объект в стеке каналов. Каждая привязка, например `basicHttpBinding`, состоит из нескольких элементов. Убедиться в этом можно, написав программу, в которой создается объект привязки, а затем перебирается набор входящих в нее элементов привязки. Такая программа показана в листинге 4.3.

Листинг 4.3. Обход набора BindingElementCollection

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         OutputBindingElements(new WSHttpBinding());
         OutputBindingElements(new NetTcpBinding());
         OutputBindingElements(new NetNamedPipeBinding());
         OutputBindingElements(new BasicHttpBinding());

         Console.ReadLine();
      }

      static void OutputBindingElements(Binding binding)
      {
         Console.WriteLine(" Привязка : {0}", binding.GetType().Name);

         BindingElementCollection elements =
                  binding.CreateBindingElements();

         foreach (BindingElement element in elements)
            Console.WriteLine(" {0}", element.GetType().FullName);

         Console.WriteLine();
      }
   }
}
```


На рис. 4.1 показано, что выводит эта программа для четырех готовых привязок: `WSHttpBinding`, `NetTcpBinding`, `NetNamedPipeBinding` и `BasicHttpBinding`. Мы подробно рассмотрим привязку `WSHttpBinding`, чтобы понять, как она конструируется из элементов.

По умолчанию привязка `WSHttpBinding` состоит из четырех элементов: `HttpTransportBindingElement`, `TextMessageEncodingBindingElement`, `SymmetricSecurityBindingElement` и `TransactionFlowBindingElement`. Это означает, что она допускает коммуникации по протоколу HTTP, сообщения кодируются в текстовом виде, поддерживается безопасность и транзакционность. В списке содержатся все элементы, сконфигурированные по умолчанию. Но в привязку можно добавлять новые элементы или удалять из нее имеющиеся.

Обратите внимание, что каждая привязка состоит из одного или нескольких элементов, причем некоторые элементы входят в различные привязки. Например, и `WSHttpBinding`, и `BasicHttpBinding` включают элемент `HttpTransportBindingElement`. Обе эти привязки используют один и тот же транспортный ме-

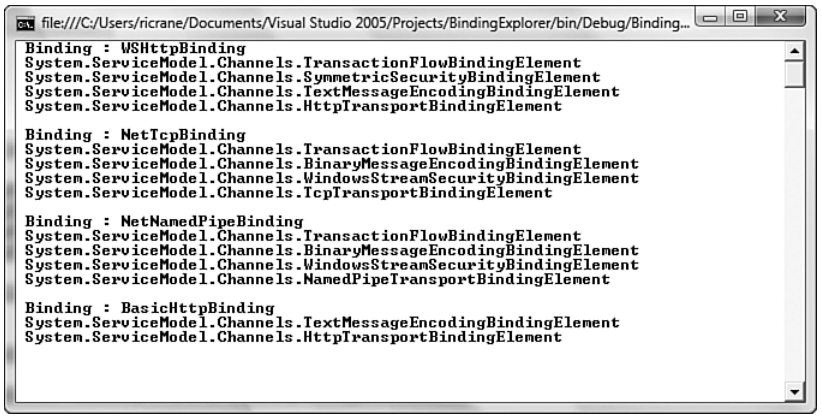

Рис. 4.1. Исследование состава привязок

ханизм, но различаются поддерживаемыми функциональными возможностями. Ниже в этой главе мы обсудим эти различия.

Далее в этой главе мы сосредоточим внимание на коммуникациях на основе Web-служб, межмашинных, в пределах одной машины и на основе очередей. Именно с этими видами коммуникаций должен быть знаком разработчик, приступающий к изучению WCF. Существуют и другие виды коммуникаций, в частности на базе спецификации REST/POX, в пиринговых сетях и с интегрированной безопасностью, но их мы будем подробно обсуждать в главе 13 «Средства программирования Web», главе 12 «Пиринговые сети» и в главе 8 «Безопасность» соответственно.

Выбор подходящей привязки

В WCF существует девять готовых привязок. Каждая из них отвечает потребностям одного конкретного способа распределенных вычислений. На выбор привязки оказывают влияние несколько факторов, в том числе безопасность, интероперабельность, надежность, производительность и транзакционность. В таблице 4.2 эти девять привязок сравниваются с точки зрения предоставляемых возможностей. Вы можете пользоваться этой таблицей для выбора привязки, оптимальной для ваших потребностей.

Чтобы выбрать привязку, вы должны изучить потребности приложения и решить, какая привязка наилучшим образом удовлетворяет требованиям. Так, если приложение должно обмениваться данными по ненадежной (к примеру, беспроводной) сети, то стоит подумать о привязке, поддерживающей надежные сеансы (reliable sessions – RS). На рис. 4.2 изображена блок-схема принятия решения о выборе привязки.

При выборе привязки приходится рассматривать много разных факторов. Перечислить их все в таблице 4.2 невозможно, поэтому будьте готовы к проведению дополнительного анализа.

Каждая привязка поддерживает какой-то один сценарий коммуникаций, например: межмашинная, локальная или интероперабельная с использованием Web-служб. Эти сценарии и ассоциированные с ними привязки мы рассмотрим в этой главе. Но существуют и другие варианты, например, интегрированная безопасность и коммуникации между равноправными участниками (пиринг). Они заслуживают более глубокого обсуждения и будут рассмотрены в главах 8 и 12 соответственно.

Пример приложения

Теперь приступим к рассмотрению каждой из готовых привязок, включенных в комплект поставки WCF. Мы продемонстрируем их на примере приложения, относящегося к котировкам акций. Клиент будет передавать символ акции (тикер), и получать в ответ ее цену. Наша цель состоит в том, чтобы предоставить и потребить одну и ту же службу, пользуясь разными привязками, и посмотреть, какие понадобятся изменения в коде или в конфигурационном файле.

В листинге 4.4 приведен код службы, информирующей о котировках акций.

Таблица 4.2. Функции, поддерживаемые различными привязками

	Безопасность на транспортном уровне	Безопасность на уровне сообщений	Интероперабельность Ws*	Транзакции Ws*	Гарантированная доставка сообщений	Надежные сеансы	Производительность	Коммуникации: Запрос-ответ	Коммуникации: Одно-сторонняя	Коммуникации: Дуплексная
basicHttpBinding	X	X	X				Хорошая	X	X	
wsHttpBinding	X	X	X	X		RS*	Хорошая	X	X	
wsDualHttpBinding	X	X	X	X		RS*	Хорошая	X	X	X
netTcpBinding	X	X		X		RS*	Удовлет.	X	X	X
netNamedPipeBinding	X			X			Отличная	X	X	X
netMsmqBinding	X	X	X		X		Удовлет.		X	
netPeerTcpBinding	X						Хорошая		X	X
msmqIntegrationBinding	X				X		Удовлет.		X	
wsFederationHttpBinding	X	X	X			RS*	Хорошая	X	X	
ws2007HttpBinding	X	X	X	X		RS*	Хорошая	X	X	
ws2007FederationHttpBinding	X	X	X			RS*	Хорошая	X	X	

*RS – надежные сеансы в WCF – это реализация надежной передачи сообщений SOAP, определенной в спецификации WS-Reliable Messaging (WS-RM)

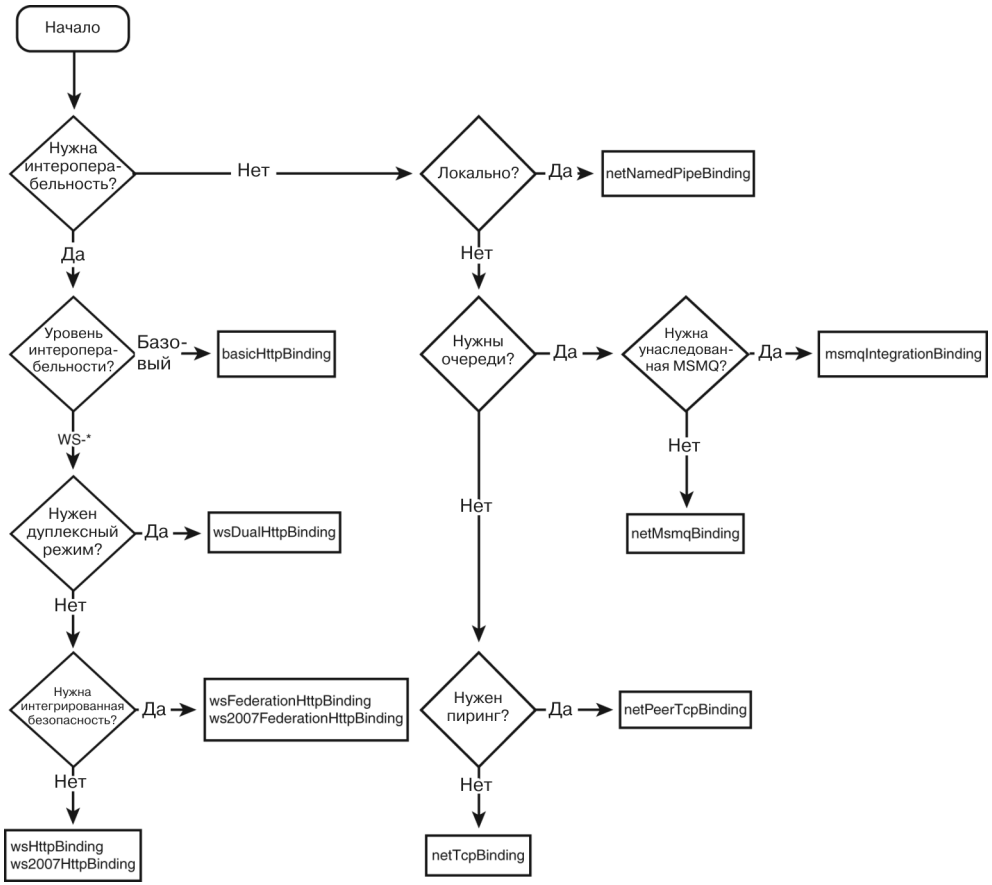

Рис. 4.2. Выбор привязки

Листинг 4.4. Служба StockQuoteService

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
using System.Runtime.Serialization;

namespace EssentialWCF
{
   [ServiceContract]
   public interface IStockQuoteService
   {
      [OperationContract]
      double GetQuote(string symbol);
   }

   public class StockQuoteService : IStockQuoteService
```



```
   {
      public double GetQuote(string symbol)
      {
         double value;

         if (symbol == "MSFT")
               value = 31.15;
         else if (symbol == "YHOO")
               value = 28.10;
         else if (symbol == "GOOG")
               value = 450.75;
         else
            value = double.NaN;

         return value;
      }
   }
}
```

В листинге 4.5 приведен клиентский прокси-класс, сгенерированный с помощью операции Add Service Reference в Visual Studio. Мы вручную удалили из него комментарии и добавили предложения `using`, включив общеупотребительные пространства имен. Далее мы увидим, какие изменения в код или конфигурацию нужно внести, чтобы пользоваться одной и той же клиентской программой с разными привязками.

Листинг 4.5. Клиентский прокси-класс для службы StockQuoteService

```
using System.CodeDom.Compiler;
using System.Diagnostics;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{

   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   [ServiceContractAttribute(
      ConfigurationName="IStockQuoteDuplexService",
      CallbackContract=typeof(IStockQuoteDuplexServiceCallback),
      SessionMode=SessionMode.Required)]
   public interface IStockQuoteDuplexService
   {

      [OperationContractAttribute(IsOneWay=true,
            Action="http://tempuri.org/IStockQuoteDuplexService/
            ➥SendQuoteRequest")]
      void SendQuoteRequest(string symbol);
   }

   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   public interface IStockQuoteDuplexServiceCallback
   {

   [OperationContractAttribute(IsOneWay=true,
```

```
            Action="http://tempuri.org/IStockQuoteDuplexService/
            ➥SendQuoteResponse")]
      void SendQuoteResponse(string symbol, double price);
   }

   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   public interface IStockQuoteDuplexServiceChannel :
      IStockQuoteDuplexService, IClientChannel
   {
   }

   [DebuggerStepThroughAttribute()]
   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   public partial class StockQuoteDuplexServiceClient :
      DuplexClientBase<IStockQuoteDuplexService>,
      IStockQuoteDuplexService
   {

      public StockQuoteDuplexServiceClient(
               InstanceContext callbackInstance)
               : base(callbackInstance)
      {
      }

      public StockQuoteDuplexServiceClient(
               InstanceContext callbackInstance,
                  string endpointConfigurationName)
                     : base(callbackInstance,
                              endpointConfigurationName)
      {
      }

      public StockQuoteDuplexServiceClient(
               InstanceContext callbackInstance,
               string endpointConfigurationName,
               string remoteAddress)
               : base(callbackInstance,
                           endpointConfigurationName,
                           remoteAddress)
      {
      }

      public StockQuoteDuplexServiceClient(
               InstanceContext callbackInstance,
               string endpointConfigurationName,
               EndpointAddress remoteAddress)
               : base(callbackInstance,
                           endpointConfigurationName,
                           remoteAddress)
      {
      }

      public StockQuoteDuplexServiceClient(InstanceContext callbackInstance,
Binding binding, EndpointAddress remoteAddress)
```



```
               :
                  base(callbackInstance, binding, remoteAddress)
      {
      }

      public void SendQuoteRequest(string symbol)
      {
         base.Channel.SendQuoteRequest(symbol);
      }
   }
}
```

В следующем приложении (листинг 4.6) применено авторазмещение службы StockQuoteService. Подробнее об авторазмещении см. главу 7.

Листинг 4.6. Класс ServiceHost для службы StockQuoteService

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Text;
using System.ServiceModel;

namespace EssentialWCF
{
   internal class MyServiceHost
   {
      internal static ServiceHost myServiceHost = null;

      internal static void StartService()
      {
         myServiceHost = new
ServiceHost(typeof(EssentialWCF.StockQuoteService));
         myServiceHost.Open();
      }

      internal static void StopService()
      {
         if (myServiceHost.State != CommunicationState.Closed)
               myServiceHost.Close();
      }
   }
}
```

Коммуникация между .NET-приложениями на разных машинах

В этом разделе описываются привязки для коммуникации между .NET-приложениями на разных машинах. Мы покажем, как настроить привязки в коде и в конфигурационном файле. Каждая привязка рассматривается в контексте типичного сценария.

Привязки, начинающиеся с префикса net, предназначены для коммуникации между .NET-приложениями. В WCF имена привязок, предназначенных для коммуникации между приложениями на платформе .NET, начинаются с префикса «net». Префикс имени служит подсказкой при выборе привязки. В частности, префикс «net» означает, что в привязке используются возможности, доступные только .NET-приложениям. Напротив, привязки, начинающиеся с префикса «ws», обеспечивают интероперабельность с приложениями на других платформах, поскольку в них применяются Web-службы.

Привязка netTcpBinding

Привязка `netTcpBinding` предназначена для взаимодействия между .NET-приложениями, развернутыми на разных машинах в сети – Интернет или Интранет. Такой вид взаимодействия мы будем называть *межмашинной коммуникацией*. В этой ситуации интероперабельность не нужна, так как оба приложения написаны для платформы .NET. Это дает нам большую гибкость в организации сетевого взаимодействия и позволяет добиться максимальной производительности.

В привязке `netTcpBinding` для оптимизации скорости передачи данных применяется двоичное кодирование и протокол TCP. Ее рекомендуется использовать для организации взаимодействия между .NET-приложениями на разных машинах. Это правило нельзя назвать непреложным, но в большинстве ситуаций им можно руководствоваться. Неприменимо оно, например, в случае когда между двумя машинами находится брандмауэр. Часто единственный способ передать что-то через брандмауэр состоит в том, чтобы воспользоваться протоколом HTTP. Если дело обстоит именно так, то придется ограничиться привязкой, работающей по этому протоколу, например, `basicHttpBinding`.

Ниже показан формат адреса для привязки `netTcpBinding`:

```
net.tcp://{hostname}[:port]/{service location}
```

По умолчанию для транспорта по протоколу TCP используется порт 808. Это относится к любой привязке, включающей элемент `TcpTransportBindingElement`, в том числе и `netTcpBinding`.

В таблице 4.3 перечислены свойства, которые можно сконфигурировать для привязки `netTcpBinding`. Все они могут оказаться важными в той или иной ситуации. Например, по умолчанию обобществление портов отключено. Но его необходимо включить, если вы планируете размещать несколько служб на одном порту. Подробнее об этом см. раздел «Обобществление портов» в приложении. Еще одно важное свойство привязки `netTcpBinding` – `maxConnections`. Оно ограничивает количество одновременных соединений с оконечной точкой и по умолчанию равно 10. Для увеличения пропускной способности следует задать большее значение.

***Таблица 4.3. Свойства привязки* `netTcpBinding`**

Имя атрибута	Описание	Значение по умолчанию
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00

Таблица 4.3. Свойства привязки* `netTcpBinding` *(окончание)

Имя атрибута	Описание	Значение по умолчанию
`hostNameComparisonMode`	Способ сравнения имен узлов при разборе URI	StrongWildCard
`listenBacklog`	Максимальное число каналов, готовых к обслуживанию запросов. Последующие запросы на соединение ставятся в очередь	10
`maxBufferPoolSize`	Максимальный размер пула буферов для транспортного протокола	524 888
`maxBufferSize`	Максимальный размер буфера, используемого для хранения входящих сообщений в памяти (в байтах)	65 536
`maxConnections`	Максимальное число входящих или исходящих соединений. Входящие и исходящие соединения считаются порознь	10
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`portSharingEnabled`	Разрешать ли обобществление портов прослушивателями служб	false
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`reliableSession`	Гарантирует ли привязка доставку каждого сообщения ровно один раз в соответствии со спецификацией WS-Reliable Messaging	Нет
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`transactionFlow`	Разрешен ли поток транзакций от клиента к серверу	false
`transactionProtocol`	Тип поддерживаемых транзакций: OleTransactions или WSAtomicTransactions	OleTransactions

«Нет» означает, что данная характеристика является дочерним элементом, для которого нужно задать несколько свойств, или что она неприменима, если одновременно не установлено какое-то другое свойство.

Ниже приведена конфигурация, которую предполагается использовать для приложения, представленного в листингах 4.2–4.4. В данном случае служба `StockQuoteService` раскрывается через привязку `netTcpBinding`.

Листинг 4.7. Конфигурация службы с помощью привязки netTcpBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteService">
            <host>
               <baseAddresses>
                  <add baseAddress="net.tcp://localhost/stockquoteservice" />
               </baseAddresses>
            </host>
            <endpoint address=""
                      contract="EssentialWCF.IStockQuoteService"
                      binding="netTcpBinding" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

В листинге 4.8 показана конфигурация клиента, который собирается обращаться к этой службе с помощью привязки netTcpBinding.

Листинг 4.8. Конфигурация клиента с помощью привязки netTcpBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="net.tcp://localhost/stockquoteservice"
                   binding="netTcpBinding"
                   contract="EssentialWCF.IStockQuoteService">
         </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```

Коммуникация между .NET-приложениями на одной машине

Под межпроцессной коммуникацией понимается взаимодействие двух процессов, работающих на одной и той же машине. Внутрипроцессной коммуникацией называется взаимодействие двух программных компонентов, работающих внутри одного процесса. То и другое являются частными случаями *локальной* коммуникации.

В .NET есть понятие *домена приложения*; это механизм подразделения процесса Windows на области, в которых работают различные .NET-приложения. Домен определяет границы безопасности и активации приложения. Это означает, что в .NET-приложениях существует еще один рубеж, для преодоления которого нужны средства коммуникации. Поэтому мы введем два дополнительных термина: междоменный и внутридоменный.

- **междоменная, или кросс-доменная коммуникация** возникает в случае, когда два .NET-приложения работают в различных доменах приложений внутри одного процесса Windows. Это может быть и коммуникация в пределах одного .NET-приложения, но спроектированная так, что ее можно обобщить на несколько доменов;
- **внутридоменная коммуникация** имеет место в одном .NET-приложении, которое работает в одном домене приложения. Для наших целей можно считать, что домен приложения – это некий вид процесса .NET. работающего внутри процесса Windows.

WCF не различает межпроцессные, внутрипроцессные, междоменные и внутридоменные коммуникации. Вместо этого WCF предлагает единый локальный транспортный механизм, основанный на применении именованных каналов. Именованные каналы – это стандартный способ межпроцессных коммуникаций в Windows и в UNIX. Разработчики WCF думали о том, чтобы включить привязку для внутрипроцессных коммуникаций, но решили, что она нужна очень редко. Пусть это решение вас не смущает, никакой потери функциональности оно за собой не влечет. Единственное различие между именованным каналом и настоящей внутрипроцессной привязкой заключается в производительности.

Производительность привязки к именованным каналам достаточна для большинства ситуаций, когда возникает нужда во внутрипроцессных коммуникациях. Если обнаружится, что единственного локального транспорта недостаточно, вы всегда можете написать заказную привязку к специализированному транспортному каналу. О том, как это делается, см. раздел «Создание заказной привязки» ниже в этой главе.

Привязка netNamedPipeBinding

WCF поддерживает межпроцессные и внутрипроцессные коммуникации с помощью привязки `netNamedPipeBinding`, в которой используются именованные каналы. Она очень удобна для межпроцессных коммуникаций (IPC), так как обеспечивает в этой ситуации гораздо более высокую производительность, чем другие имеющиеся в WCF стандартные привязки (см. раздел «Сравнение производительности и масштабируемости привязок» ниже в этой главе).

WCF позволяет использовать привязку `netNamedPipeBinding` только для локальных коммуникаций! Хотя Windows допускает использование именованных каналов для передачи данных по сети, WCF ограничивает применение этого механизма только коммуникациями в пределах одной машины. Это означает, что, указывая привязку `netNamedPipeBinding` (и любую другую, включающую элемент `namedPipeTransport`), вы гарантируете, что ваша служба не будет доступна через сеть. Достигается это с помощью двух приемов. Во-первых, сетевому идентификатору безопасности (SID: S-1-5-2) запрещается доступ

к именованному каналу. А, во-вторых, имя канала генерируется случайным образом и хранится в разделяемой памяти, поэтому получить к нему доступ могут только клиенты, работающие на той же машине.

Адрес, в котором используется привязка `netNamedPipeBinding`, выглядит следующим образом:

```
net.pipe://localhost/{service location}
```

В таблице 4.4 перечислены свойства, которые можно сконфигурировать для привязки `netNamedPipeBinding`. Особенно важно свойство `maxConnections`, которое ограничивает количество одновременных соединений с оконечной точкой. По умолчанию оно равно 10. Для увеличения пропускной способности следует задать большее значение.

***Таблица 4.4. Свойства привязки* `netNamedPipeBinding`**

Имя атрибута	Описание	Значение по умолчанию
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00
`hostNameComparisonMode`	Способ сравнения имен узлов при разборе URI	StrongWildCard
`maxBufferPoolSize`	Максимальный размер пула буферов для транспортного протокола	524 888
`maxBufferSize`	Максимальный размер буфера, используемого для хранения входящих сообщений в памяти (в байтах)	65 536
`maxConnections`	Максимальное число входящих или исходящих соединений. Входящие и исходящие соединения считаются порознь	10
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`portSharingEnabled`	Разрешать ли обобществление портов прослушивателями служб	false
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00

Таблица 4.4. Свойства привязки* `netNamedPipeBinding` *(окончание)

Имя атрибута	Описание	Значение по умолчанию
`transactionFlow`	Разрешен ли поток транзакций от клиента к серверу	false
`transactionProtocol`	Тип поддерживаемых транзакций: OleTransactions или WSAtomicTransactions	OleTransactions

«Нет» означает, что данная характеристика является дочерним элементом, для которого нужно задать несколько свойств, или что она неприменима, если одновременно не установлено какое-то другое свойство.

Ниже приведена конфигурация, которую предполагается использовать для приложения, представленного в листингах 4.2–4.4. В данном случае служба `StockQuoteService` раскрывается через привязку `netNamedPipeBinding`.

Листинг 4.9. Конфигурация службы с помощью привязки `netNamedPipeBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteService">
            <host>
               <baseAddresses>
                  <add baseAddress="net.pipe://localhost/stockquoteservice" />
               </baseAddresses>
            </host>
            <endpoint address=""
                        contract="EssentialWCF.IStockQuoteService"
                        binding="netNamedPipeBinding" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

В листинге 4.10 показана конфигурация клиента, который собирается обращаться к этой службе с помощью привязки `netNamedPipeBinding`.

Листинг 4.10. Конфигурация клиента с помощью привязки `netNamedPipeBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="net.pipe://localhost/stockquoteservice"
                     binding="netNamedPipeBinding"
                     contract="EssentialWCF.IStockQuoteService">
```

```
        </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```

Коммуникация с использованием Web-служб

Web-службы – это основа интероперабельности гетерогенных систем. Так, службы, работающие на платформе Java, например IBM Websphere или BEAWebLogic, должны взаимодействовать с клиентами и службами на платформе .NET и наоборот. До появления WCF такую возможность обеспечивали ASMX-службы в ASP.NET и технология Web Service Enhancements (WSE). В .NET 3.0 WCF заменяет обе эти технологии и предоставляет унифицированный каркас для разработки Web-служб. В WCF включено несколько привязок для раскрытия интероперабельных Web-служб, в том числе: `basicHttpBinding`, `wsHttpBinding`, `wsDualHttpBinding` и `wsFederationHttpBinding`

В этом разделе мы рассмотрим привязку `basicHttpBinding`, поддерживающую Web-службы, основанные на спецификации WS-I Basic Profile 1.1. В 2007 году эта спецификация покрывала абсолютное большинство развернутых в мире Web-служб, поэтому разработчики лучше всего знакомы именно с этой технологией. Другие привязки, относящиеся к Web-службам, обсуждаются в разделе «Коммуникации с помощью продвинутых Web-служб» ниже в этой главе, а также в главе 8.

Привязка basicHttpBinding

Привязка `basicHttpBinding` поддерживает коммуникации с помощью Web-служб, основанных на спецификации WS-I Basic Profile 1.1 (WS-BP 1.1). Последняя включает стандарты SOAP 1.1, WSDL 1.1 и Message Security 1.0 (в том числе X.509 и UserName Tokens Profile v1.0). Эта спецификация существует с 2004 года. Хотя привязка `basicHttpBinding` и обеспечивает интероперабельность с гетерогенными системами, она не поддерживает более поздние стандарты Web-служб, такие, как транзакции и надежная доставка сообщений. Предполагается, что эта привязка будет использоваться только в приложениях на базе Web-служб, основанных на на спецификации WS-BP 1.1, в каковым относятся и ASMX-службы.

Создавайте службы, совместимые с последними стандартами. Привязка `basicHttpBinding` предназначена для работы с унаследованными Web-службами, основанными на старых технологиях, в частности ASP.NET. Следовательно, по умолчанию она сконфигурирована для работы с устаревшими версиями стандартов, например SOAP 1.1. К тому же это единственная привязка, которая по умолчанию небезопасна. Если вы собираетесь создавать новые Web-службы, рекомендуем пользоваться привязкой `ws2007HttpBinding`, которая совместима с современными стандартами и безопасна по умолчанию.

Ниже показаны форматы адресов для привязки `basicHttpBinding`:

```
http://{hostname}[:port]/{service location}
https://{hostname}[:port]/{service location}
```


По умолчанию для протокола http используется порт 80, а для протокола https – порт 443. Это справедливо для любой привязки, включающей элемент `HttpTransportBindingElement`, в том числе для `basicHttpBinding`. Самый распространенный способ сделать привязку `basicHttpBinding` безопасной – воспользоваться протоколом https, так как в этом случае передаваемые данные шифруются по протоколу SSL/TLS.

В таблице 4.5 перечислены свойства, которые можно сконфигурировать для привязки `basicHttpBinding`.

Таблица 4.5. Свойства привязки `basicHttpBinding`

Имя атрибута	Описание	Значение по умолчанию
`bypassProxyOnLocal`	Игнорировать настройки прокси-сервера при соединении с локальной оконечной точкой	false
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00
`hostNameComparisonMode`	Способ сравнения имен узлов при разборе URI	StrongWildCard
`maxBufferPoolSize`	Максимальный размер пула буферов для транспортного протокола	524 888
`maxBufferSize`	Максимальный размер буфера, используемого для хранения входящих сообщений в памяти (в байтах)	65 536
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`messageEncoding`	Способ кодирования сообщений	Text
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`proxyAddress`	Адрес используемого прокси-сервера. Это свойство принимается во внимание, только если `useDefaultWebProxy` равно false	Нет
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`textEncoding`	Метод кодировки символов в сообщениях. Это свойство принимается во внимание, только если	utf-8

	`messageEncoding` равно Text	
`transferMode`	Определяет, как сообщения посылаются по сети. Возможна буферизованная или потоковая передача	Buffered
`useDefaultWebProxy`	Использовать ли прокси-сервер, указанный в операционной системе	true

Ниже приведена конфигурация, которую предполагается использовать для приложения, представленного в листингах 4.2–4.4. В данном случае служба `StockQuoteService` раскрывается через привязку `basicHttpBinding`.

Листинг 4.11. Конфигурация службы с помощью привязки `basicHttpBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteService">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost/stockquoteservice" />
               </baseAddresses>
            </host>
            <endpoint address=""
                      contract="EssentialWCF.IStockQuoteService"
                      binding="basicHttpBinding" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

В листинге 4.12 показана конфигурация клиента, который собирается обращаться к этой службе с помощью привязки `basicHttpBinding`.

Листинг 4.12. Конфигурация клиента с помощью привязки `basicHttpBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="http://localhost/stockquoteservice"
                   binding="basicHttpBinding"
                   contract="EssentialWCF.IStockQuoteService">
         </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```


Коммуникации с помощью продвинутых Web-служб

Выше уже упоминалось, что Web-службы составляют основу интероперабельных коммуникаций между гетерогенными системами. Продвинутыми мы называем Web-службы, удовлетворяющие спецификациям WS-*. В WCF имеется поддержка этих спецификаций, в том числе безопасности, надежной доставки сообщений и транзакционности. Перечень поддерживаемых спецификаций приведен в таблице 4.6. Поддержка этих функций встроена в привязки `wsHttpBinding`, `wsDualHttpBinding` и `wsFederationHttpBinding`.

Привязки, начинающиеся с префикса «ws», следует использовать, когда необходимо реализовать интероперабельность с помощью Web-служб. В WCF имена всех привязок, предназначенных для реализации интероперабельности посредством Web-служб, начинаются с префикса «ws». Напротив, привязки, начинающиеся с префикса «net», следует использовать только для организации взаимодействия между .NET-приложениями.

Таблица 4.6. Спецификации WS-*, поддерживаемые привязкой `wsHttpBinding`

Стандарт	Описание
SOAP 1.2	Облегченный протокол обмена информацией в децентрализованной распределенной среде
WS-Addressing 2005/08	Независимые от транспорта механизмы адресации Web-служб и сообщений
WSS Message Security 1.0	Спецификация обеспечения безопасности Web-служб с помощью различных механизмов, включая PKI, Kerberos и SSL
WSS Message Security UsernameToken Profile 1.1	Поддержка маркеров безопасности на основе имени пользователя и необязательного пароля (либо эквивалента пароля, например разделяемого секрета)
WSS SOAP Message Security X509 Token Profile 1.1	Поддержка маркеров на основе сертификатов X.509
WS-SecureConversation	Расширения WS-Security с целью предоставления безопасного контекста для обмена несколькими сообщениями
WS-Trust	Расширения WS-Security с целью запроса и выпуска маркеров и управления отношениями доверия
WS-SecurityPolicy	Политики для WS-Security, WS-Secure-Conversation и WS-Trust, выраженные с помощью WS-Policy
WS-ReliableMessaging	Протокол гарантированной доставки сообщений в правильном порядке и без дубликатов
WS-Coordination	Каркас для создания протоколов координации действий в распределенных приложениях

Таблица 4.6. Спецификации WS-*, поддерживаемые привязкой `wsHttpBinding` (окончание)

Стандарт	Описание
WS-Atomic Transactions	Протокол координации действий в распределенных приложениях, основанный на атомарных транзакциях
WS-Addressing	Независимый от транспорта механизм адресации Web-служб

Привязка wsHttpBinding

В WCF включена поддержка спецификаций WS-*. Примером такой поддержки служит привязка `wsHttpBinding`. Она обеспечивает интероперабельность гетерогенных систем, а также дополнительные протоколы инфраструктурного уровня, например, безопасность, надежную доставку сообщений и транзакционность. В версии .NET Framework 3.0 привязка `wsHttpBinding` принимается по умолчанию в тех случаях, когда необходимы интероперабельные коммуникации на базе Web-служб.

Ниже показаны форматы адресов для привязки `wsHttpBinding`:

```
http://{hostname}:{port}/{service location}
https://{hostname}:{port}/{service location}
```

По умолчанию для протокола http используется порт 80, а для протокола https – порт 443. Это справедливо для любой привязки, включающей элемент `HttpTransportBindingElement`, в том числе для `wsHttpBinding`.

В таблице 4.7 перечислены свойства, которые можно сконфигурировать для привязки `wsHttpBinding`.

Таблица 4.7. Свойства привязки `wsHttpBinding`

Имя атрибута	Описание	Значение по умолчанию
`bypassProxyOnLocal`	Игнорировать настройки прокси-сервера при соединении с локальной оконечной точкой	false
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00
`hostNameComparisonMode`	Способ сравнения имен узлов при разборе URI	StrongWildCard
`maxBufferPoolSize`	Максимальный размер пула буферов для транспортного протокола	524 888
`maxBufferSize`	Максимальный размер буфера, используемого для хранения входящих сообщений в памяти (в байтах)	65 536
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536

Таблица 4.7. Свойства привязки `wsHttpBinding` (окончание)

Имя атрибута	Описание	Значение по умолчанию
`messageEncoding`	Способ кодирования сообщений	Text
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`proxyAddress`	Адрес используемого прокси-сервера. Это свойство принимается во внимание, только если `useDefaultWebProxy` равно false	Нет
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`reliableSession`	Гарантирует ли привязка доставку каждого сообщения ровно один раз в соответствии со спецификацией WS-Reliable Messaging	Нет
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`textEncoding`	Метод кодирования символов в сообщениях. Это свойство принимается во внимание, только если `messageEncoding` равно Text	utf-8
`transactionFlow`	Разрешен ли поток транзакций от клиента к серверу	false
`useDefaultWebProxy`	Использовать ли прокси-сервер, указанный в операционной системе	true

Ниже приведена конфигурация, которую предполагается использовать для приложения, представленного в листингах 4.2–4.4. В данном случае служба `StockQuoteService` раскрывается через привязку `wsHttpBinding`.

Листинг 4.13. Конфигурация службы с помощью привязки `wsHttpBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteService">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost/stockquoteservice" />
               </baseAddresses>
```

```
            </host>
            <endpoint address=""
                      contract="EssentialWCF.IStockQuoteService"
                      binding="wsHttpBinding" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

В листинге 4.14 показана конфигурация клиента, который собирается обращаться к этой службе с помощью привязки `wsHttpBinding`.

Листинг 4.14. Конфигурация клиента с помощью привязки `wsHttpBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="http://localhost/stockquoteservice"
                   binding="wsHttpBinding"
                   contract="EssentialWCF.IStockQuoteService">
         </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```

Привязка ws2007HttpBinding

В версии .NET Framework 3.5 появилась новая привязка для обеспечения интероперабельности Web-служб – `ws2007HttpBinding`. Она аналогична привязке `wsHttpBinding`, но поддерживает самые последние стандарты WS-*, описывающие передачу сообщений, безопасность, надежную доставку и транзакционность. В таблице 4.8 перечислены новые стандарты, которые поддерживает привязка `ws2007HttpBinding`.

Таблица 4.8. Спецификации WS-*, поддерживаемые привязкой `ws2007HttpBinding`

Стандарт	Описание
WS-SecureConversation v1.3	Расширения WS-Security с целью предоставления безопасного контекста для обмена несколькими сообщениями
WS-Trust v1.3	Расширения WS-Security с целью запроса и выпуска маркеров и управления отношениями доверия
WS-SecurityPolicy v1.2	Политики для WS-Security, WS-Secure-Conversation и WS-Trust, выраженные с помощью WS-Policy
WS-ReliableMessaging v1.1	Протокол гарантированной доставки сообщений в правильном порядке и без дубликатов
WS-Atomic Transactions v1.1	Протокол координации действий в распределенных приложениях, основанный на атомарных транзакциях

Таблица 4.8. Спецификации WS-*, поддерживаемые привязкой `ws2007HttpBinding` (окончание)

Стандарт	Описание
WS-Addressing	Независимый от транспорта механизм адресации Web-служб
Web Services Coordination v1.1	Каркас для создания протоколов координации действий в распределенных приложениях

Ниже показаны форматы адресов для привязки `ws2007HttpBinding`:

```
http://{hostname}:{port}/{service location}
https://{hostname}:{port}/{service location}
```

По умолчанию для протокола http используется порт 80, а для протокола https – порт 443. Это справедливо для любой привязки, включающей элемент `HttpTransportBindingElement`, в том числе для `ws2007HttpBinding`.

В таблице 4.9 перечислены свойства, которые можно сконфигурировать для привязки `ws2007HttpBinding`.

Таблица 4.9. Свойства привязки `ws2007HttpBinding`

Имя атрибута	Описание	Значение по умолчанию
`bypassProxyOnLocal`	Игнорировать настройки прокси-сервера при соединении с локальной оконечной точкой	false
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00
`hostNameComparisonMode`	Способ сравнения имен узлов при разборе URI	StrongWildCard
`maxBufferPoolSize`	Максимальный размер пула буферов для транспортного протокола	524 888
`maxBufferSize`	Максимальный размер буфера, используемого для хранения входящих сообщений в памяти (в байтах)	65 536
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`messageEncoding`	Способ кодирования сообщений	Text
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`proxyAddress`	Адрес используемого прокси-сервера. Это свойство принимается во внимание, только если `useDefaultWebProxy` равно false	Нет

Таблица 4.9. Свойства привязки* `ws2007HttpBinding` *(окончание)

Имя атрибута	Описание	Значение по умолчанию
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`reliableSession`	Гарантирует ли привязка доставку каждого сообщения ровно один раз в соответствии со спецификацией WS-Reliable Messaging	Нет
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`textEncoding`	Метод кодирования символов в сообщениях. Это свойство принимается во внимание, только если `messageEncoding` равно Text	utf-8
`transactionFlow`	Разрешен ли поток транзакций от клиента к серверу	false
`useDefaultWebProxy`	Использовать ли прокси-сервер, указанный в операционной системе	true

Ниже приведена конфигурация, которую предполагается использовать для приложения, представленного в листингах 4.2–4.4. В данном случае служба `StockQuoteService` раскрывается через привязку `ws2007HttpBinding`.

Листинг 4.15. Конфигурация службы с помощью привязки `ws2007HttpBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteService">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost/stockquoteservice" />
               </baseAddresses>
            </host>
            <endpoint address=""
                      contract="EssentialWCF.IStockQuoteService"
                      binding="ws2007HttpBinding" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

В листинге 4.16 показана конфигурация клиента, который собирается обращаться к этой службе с помощью привязки `ws2007HttpBinding`.

Листинг 4.16. Конфигурация клиента с помощью привязки `ws2007HttpBinding`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="http://localhost/stockquoteservice"
                   binding="ws2007HttpBinding"
                   contract="EssentialWCF.IStockQuoteService">
         </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```

Привязка wsDualHttpBinding

Привязка `wsDualHttpBinding` аналогична `wsHttpBinding`, но обеспечивает дополнительную поддержку для дуплексных коммуникаций и не поддерживает безопасность на транспортном уровне. Дуплексность достигается за счет двух изменяющих канальную форму элементов привязки: `OneWayBindingElement` и `CompositeDuplexBindingElement`. Элемент `CompositeDuplexBindingElement` помещает дуплексный коммуникационный канал поверх двух односторонних каналов. В привязке `wsDualHttpBinding` используется элемент `HttpTransportBindingElement`. Этот транспортный протокол поддерживает только обмен сообщениями вида запрос-ответ. Элемент `OneWayBindingElement` позволяет использовать элемент `HttpTransportBindingElement` вместе с элементом `CompositeDuplexBindingElement`.

Привязка `wsDualHttpBinding` не поддерживает безопасность на транспортном уровне, то есть шифрование по протоколу SSL/TLS для нее невозможно.

Ниже показан формат адреса для привязки `wsDualHttpBinding`:

```
http://{hostname}:{port}/{service location}
```

По умолчанию для протокола http используется порт 80. Это справедливо для любой привязки, включающей элемент `HttpTransportBindingElement`, в том числе для `wsDualHttpBinding`.

В таблице 4.10 перечислены свойства, которые можно сконфигурировать для привязки `wsDualHttpBinding`.

***Таблица 4.10. Свойства привязки* `wsDualHttpBinding`**

Имя атрибута	Описание	Значение по умолчанию
`bypassProxyOnLocal`	Игнорировать настройки прокси-сервера при соединении с локальной оконечной точкой	False
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00

Таблица 4.10. Свойства привязки* `wsDualHttpBinding` *(окончание)

Имя атрибута	Описание	Значение по умолчанию
`hostNameComparisonMode`	Способ сравнения имен узлов при разборе URI	StrongWildCard
`maxBufferPoolSize`	Максимальный размер пула буферов для транспортного протокола	524 888
`maxBufferSize`	Максимальный размер буфера, используемого для хранения входящих сообщений в памяти (в байтах)	65 536
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`messageEncoding`	Способ кодирования сообщений	Text
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`proxyAddress`	Адрес используемого прокси-сервера. Это свойство принимается во внимание, только если `useDefaultWebProxy` равно false	Нет
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`reliableSession`	Гарантирует ли привязка доставку каждого сообщения ровно один раз в соответствии со спецификацией WS-Reliable Messaging	Нет
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`textEncoding`	Метод кодировки символов в сообщениях. Это свойство принимается во внимание, только если `messageEncoding` равно Text	utf-8
`transactionFlow`	Разрешен ли поток транзакций от клиента к серверу	false
`useDefaultWebProxy`	Использовать ли прокси-сервер, указанный в операционной системе	true

Мы модифицировали приложение `StockQuoteService` для поддержки дуплексной коммуникации с помощью привязки `wsDualHttpBinding`. В листинге 4.17 приведена реализация службы `StockQuoteDuplexService`. Она поддерживает дуплексный обмен сообщениями по контракту `IStockQuoteCallback`, который представляет собой контракт обратного вызова для контракта `IStockQuoteDuplexService`.

Листинг 4.17. Интерфейсы IStockQuoteDuplexService, IStockQuoteCallback и класс StockQuoteDuplexService

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.Text;

namespace EssentialWCF
{
   [ServiceContract(CallbackContract = typeof(IStockQuoteCallback),
                                SessionMode = SessionMode.Required)]
   public interface IStockQuoteDuplexService
   {
      [OperationContract(IsOneWay = true)]
      void SendQuoteRequest(string symbol);
   }

   public interface IStockQuoteCallback
   {
      [OperationContract(IsOneWay = true)]
      void SendQuoteResponse(string symbol, double price);
   }

   [ServiceBehavior(InstanceContextMode =
            InstanceContextMode.PerSession)]
   public class StockQuoteDuplexService : IStockQuoteDuplexService
   {
      public void SendQuoteRequest(string symbol)
      {
         double value;

         if (symbol == "MSFT")
               value = 31.15;
         else if (symbol == "YHOO")
               value = 28.10;
         else if (symbol == "GOOG")
               value = 450.75;
         else
               value = double.NaN;

         OperationContext ctx = OperationContext.Current;
         IStockQuoteCallback callback =
                  ctx.GetCallbackChannel<IStockQuoteCallback>();
         callback.SendQuoteResponse(symbol, value);
      }
   }
}
```

Мы должны также изменить код авторазмещения, поскольку теперь реализация поддерживает дуплексный обмен сообщениями. В листинге 4.18 приведен код размещения службы `StockQuoteDuplexService`.

Листинг 4.18. Класс ServiceHost для службы StockQuoteDuplexService

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Text;
using System.ServiceModel;

namespace EssentialWCF
{
   internal class MyServiceHost
   {
      internal static ServiceHost myServiceHost = null;

      internal static void StartService()
      {
         myServiceHost =
            new ServiceHost(typeof(EssentialWCF.StockQuoteDuplexService));
         myServiceHost.Open();
      }

      internal static void StopService()
      {
         if (myServiceHost.State != CommunicationState.Closed)
               myServiceHost.Close();
      }
   }
}
```

В листинге 4.19 показан конфигурационный файл, в котором служба `StockQuoteDuplexService` раскрывается с помощью привязки `wsDualHttpBinding`.

Листинг 4.19. Конфигурация службы с помощью привязки wsDualHttpBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteDuplexService">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost/stockquoteservice" />
               </baseAddresses>
            </host>
            <endpoint address=""
                        binding="wsDualHttpBinding"
                        contract="EssentialWCF.IStockQuoteDuplexService">
            </endpoint>
         </service>
      </services>
   </system.serviceModel>
</configuration>
```


В листинге 4.20 показана конфигурация клиента, который собирается обращаться к этой службе по контракту `IStockQuoteDuplexService` с помощью привязки `wsDualHttpBinding`. Атрибут `clientBaseAddress` задает оконечную точку, которую клиент будет прослушивать в ожидании сообщений обратного вызова.

Листинг 4.20. Конфигурация клиента с помощью привязки wsDualHttpBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="http://localhost/stockquoteservice"
                        binding="wsDualHttpBinding"
                        bindingConfiguration="SpecifyClientBaseAddress"
                        contract="IStockQuoteDuplexService">
         </endpoint>
      </client>
      <bindings>
         <wsDualHttpBinding>
            <binding name="SpecifyClientBaseAddress"
                        clientBaseAddress="http://localhost:8001/client/" />
         </wsDualHttpBinding>
      </bindings>
   </system.serviceModel>
</configuration>
```

В листинге 4.21 показано клиентское приложение. Клиент реализует интерфейс `IStockQuoteDuplexServiceCallback`, чтобы получать от службы сообщения обратного вызова, и передает ссылку на этот интерфейс с помощью класса `InstanceContext`. Объект класса `InstanceContext` передается конструктору клиентского прокси-класса.

Листинг 4.21. Клиентское приложение, в котором используется привязка wsDualHttpBinding

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Text;
using System.Threading;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{
   public class Program : IStockQuoteDuplexServiceCallback
   {
      private static AutoResetEvent waitForResponse;

      static void Main(string[] args)
```

```
      {
         string symbol = "MSFT";

         waitForResponse = new AutoResetEvent(false);

         InstanceContext callbackInstance =
                     new InstanceContext(new Program());
         using (StockQuoteDuplexServiceClient client =
                  new StockQuoteDuplexServiceClient(callbackInstance))
         {
            client.SendQuoteRequest(symbol);
            waitForResponse.WaitOne();
         }

         Console.ReadLine();
      }

      #region IStockQuoteDuplexServiceCallback Members

      public void SendQuoteResponse(string symbol, double price)
      {
            Console.WriteLine("{0} @ ${1}", symbol, price);
            waitForResponse.Set();
      }

      #endregion
   }
}
```

В листинге 4.22 приведен код клиентского прокси-класса, сгенерированный `svcutil.exe`. Существенное отличие от предыдущих реализаций заключается в том, что прокси-класс наследует `DuplexClientBase`, а не `ClientBase`. Класс `DuplexClientBase` добавляет поддержку дуплексного обмена сообщениями.

Листинг 4.22. Клиентский прокси-класс для привязки wsDualHttpBinding

```
using System.CodeDom.Compiler;
using System.Diagnostics;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace EssentialWCF
{

   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   [ServiceContractAttribute(
         ConfigurationName = "IStockQuoteDuplexService",
         CallbackContract = typeof(IStockQuoteDuplexServiceCallback),
         SessionMode = SessionMode.Required)]
   public interface IStockQuoteDuplexService
   {

      [OperationContractAttribute(IsOneWay = true,
            Action="http://tempuri.org/IStockQuoteDuplexService/
➥SendQuoteRequest")]
```



```
      void SendQuoteRequest(string symbol);
   }

   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   public interface IStockQuoteDuplexServiceCallback
   {

      [OperationContractAttribute(IsOneWay = true,
            Action="http://tempuri.org/IStockQuoteDuplexService/
➥SendQuoteResponse")]
      void SendQuoteResponse(string symbol, double price);
   }

   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   public interface IStockQuoteDuplexServiceChannel :
               IStockQuoteDuplexService, IClientChannel
   {
   }

   [DebuggerStepThroughAttribute()]
   [GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
   public partial class StockQuoteDuplexServiceClient :
            DuplexClientBase<IStockQuoteDuplexService>,
            IStockQuoteDuplexService
   {

      public StockQuoteDuplexServiceClient(
                        InstanceContext callbackInstance)
            : base(callbackInstance)
      {
      }

      public StockQuoteDuplexServiceClient(
                        InstanceContext callbackInstance,
                        string endpointConfigurationName)
            : base(callbackInstance, endpointConfigurationName)
      {
      }

      public StockQuoteDuplexServiceClient(
                        InstanceContext callbackInstance,
                        string endpointConfigurationName,
                        string remoteAddress)
            : base(callbackInstance,
                        endpointConfigurationName,
                        remoteAddress)
      {
      }

      public StockQuoteDuplexServiceClient(
                        InstanceContext callbackInstance,
                        string endpointConfigurationName,
                        EndpointAddress remoteAddress)
            : base(callbackInstance,
                        endpointConfigurationName,
```

```
                    remoteAddress)
      {
      }

      public StockQuoteDuplexServiceClient(
                    InstanceContext callbackInstance,
                    Binding binding,
                    EndpointAddress remoteAddress)
            : base(callbackInstance, binding, remoteAddress)
      {
      }

      public void SendQuoteRequest(string symbol)
      {
         base.Channel.SendQuoteRequest(symbol);
      }
   }
}
```

Сравнение производительности и масштабируемости привязок

Разработчики должны знать о характеристиках производительности и масштабируемости привязок. Это важно, когда вы пишете реальные приложения, для которых имеют значение условия контракта о сопровождении и реакция пользователей. Если приложение работает медленно, пользователи будут недовольны. Если оно плохо масштабируется, пострадает бизнес.

Мы провели простые тесты для сравнения четырех имеющихся в WCF привязок. Тестировалась простая операция, которая возвращает строку из 256 символов. В листинге 4.23 приведен код соответствующей службы.

Листинг 4.23. Служба для тестирования производительности

```
public class PerformanceTestService : IPerformanceTestService
{
   private static string String256;

   static PerformanceTestService()
   {
      String256 = "" . PadRight(256, "X");
   }

   public string Get256Bytes()
   {
      return String256;
   }
}
```

Эта служба раскрывалась с помощью привязок `netNamedPipeBinding`, `netTcpBinding`, `wsHttpBinding` и `basicHttpBinding`. Тестовый клиент последовательно вызывал операцию `Get256Bytes` 50000 раз, после чего мы сравнивали истекшее время, число операций в секунду и затраченное процессорное время.

Все тесты выполнялись на одной рабочей станции, где были запущены и клиент, и сервер. На рис. 4.3 показан среднее время реакции для каждой привязки. Зная время реакции, вы сможете поставить себя на место пользователя.

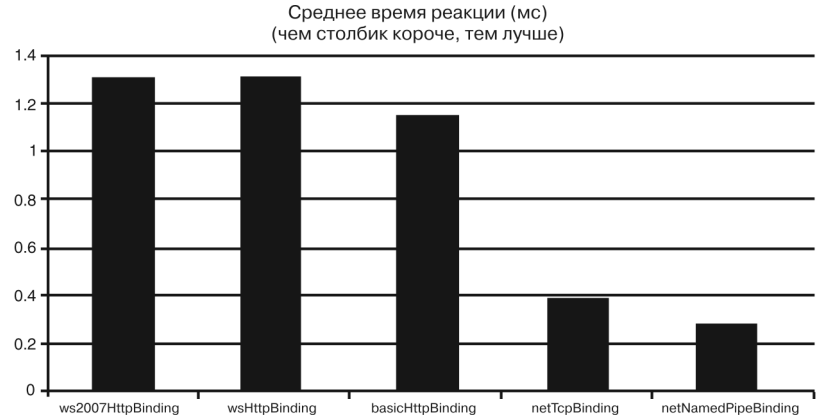

Рис. 4.3. Результаты измерения среднего времени реакции

На рис. 4.4 показано среднее число операций в секунду для каждой привязки. Результаты этого измерения отражают пропускную способность. Тестирование проводилось с одним экземпляром клиента. Использование нескольких клиентов могло бы повысить пропускную способность. Число операций в секунду – один из способов оценить масштабируемость.

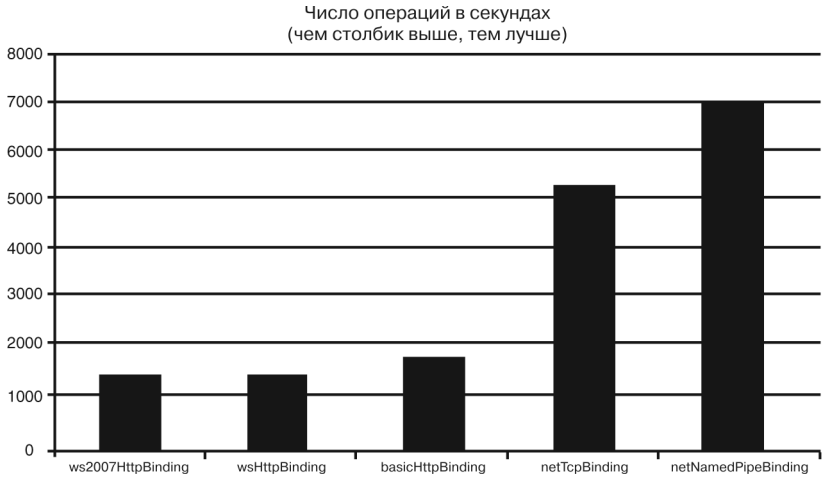

Рис. 4.4. Результаты измерения числа операций в секунду

При оценке масштабируемости надо также принимать во внимание объем аппаратных ресурсов, затрачиваемых на выполнение одной операции. На рис. 4.5 показана стоимость операции в мегациклах. Мегациклы отражают затраты процессорного времени. Мы прогоняли тест на компьютере Dell 4700 с процессором Pentium 4 3.4GHz, мощность которого эквивалентна 3400 мегациклам. Отметим, что затраты на работу с привязками `ws2007HttpBinding`, `wsHttpBinding` и `basicHttpBinding` оказались заметно выше, чем для привязок `netTcpBinding` и `netNamedPipeBinding`. Такова плата за интероперабельность.

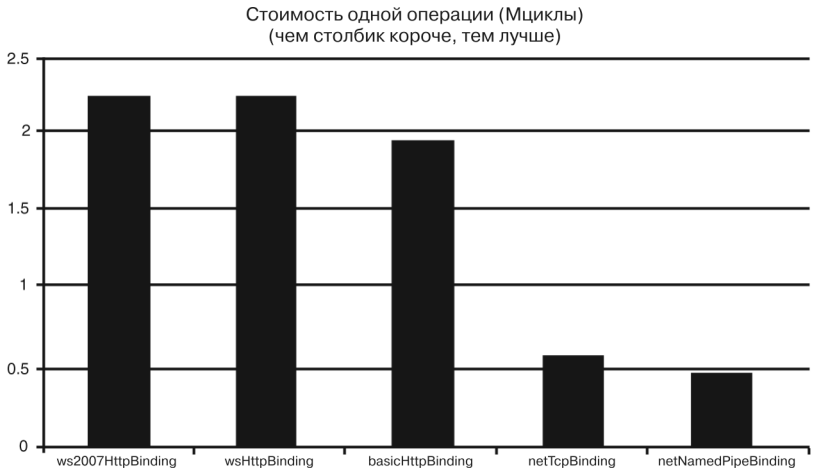

Рис. 4.5. Результаты измерения стоимости одной операции

Разработчики WCF опубликовали документ о производительности WCF (http://msdn2.microsoft.com/en-us/library/bb310550.aspx). В нем эта тема разбирается гораздо более детально, учитываются настройки безопасности на транспортном уровне, на уровне сообщений и в смешанном режиме и проводится сравнение с предшествующими технологиями, в частности NET Remoting, Web Service Enhancements, ASP.NET Web Services и Enterprise Services.

Коммуникация со службами на базе очередей

Связанными называются приложения, в которых клиент и сервер должны работать одновременно и быть доступны по сети. В *несвязанном* приложении клиент может работать и в отсутствие связи с сервером, но при этом будут доступны не все его возможности. Несвязанное приложение должно локально кэшировать данные, обеспечивать асинхронный режим работы и каким-то образом сохранять сообщения для доставки в более позднее время, когда связь будет восстановлена.

Обычно для построения несвязанных приложений применяют очереди с промежуточным хранением. Очередь можно реализовать на базе файловой системы в виде набора папок и файлов, с помощью таблиц в реляционной базе данных или

специализированного ПО. При любой реализации очереди предоставляют ряд преимуществ, в том числе асинхронную доставку сообщений и автоматическое балансирование нагрузки. WCF поддерживает коммуникации с очередями посредством технологии Microsoft Message Queue (MSMQ). Для работы с MSMQ есть две привязки: `netMsmqBinding` и `msmqIntegrationBinding`. Привязку `netMqmqBinding` рекомендуется применять при разработке новых приложений WCF с транспортом MSMQ. Привязка `msmqIntegrationBinding` используется для обеспечения интероперабельности с написанными ранее приложениями MSMQ.

Привязка netMsmqBinding

Технология MSMQ предназначена для создания распределенных приложений на базе очередей. WCF поддерживает коммуникации, в которых MSMQ выступает в роли транспортного механизма, с помощью привязки `netMsmqBinding`, позволяющей клиентам отправлять сообщения непосредственно в очередь, а службам – читать сообщения из очереди. Между клиентом и сервером нет прямой связи, коммуникация по природе своей несвязанная. Попутно это означает, что обмен данными односторонний, поэтому у всех операций в контракте должно быть установлено свойство `IsOneWay=true`.

Динамическое создание очередей. При работе с привязкой `netMsmqBinding` очереди обычно создают динамически. В особенности это относится к несвязанным клиентам, когда очередь находится на рабочей станции пользователя. Создать очередь позволяет статический метод `Create` класса `System.Messaging.MessageQueue`.

Ниже показан формат адреса для привязки `netMsmqBinding`:

```
net.msmq://{hostname}/[private/|[public/]]{queue name}
```

По умолчанию для MSMQ используется порт 1801, и изменить его в схеме адресации нельзя. Обратите внимание на слова `public` и `private` в адресе. Вы можете явно указать, является ли очередь частной или публичной. Если ничего не указано, по умолчанию предполагается, что очередь публичная.

В таблице 4.11 перечислены свойства, которые можно сконфигурировать для привязки `netMsmqBinding`.

***Таблица 4.11. Свойства привязки* `netMsmqBinding`**

Имя атрибута	Описание	Значение по умолчанию
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00
`customDeadLetterQueue`	Место нахождения очереди недоставленных писем для конкретного приложения. Недоставленным письмом называется сообщение, для которого истек срок хранения или доставка невозможна	Нет

Таблица 4.11. Свойства привязки* `netMsmqBinding` *(продолжение)

Имя атрибута	Описание	Значение по умолчанию
`deadLetterQueue`	Какую очередь использовать для недоставленных писем. Возможные значения: None, System и Custom	None
`Durable`	Указывает, является ли очередь постоянной (durable) или временной	true
`exactlyOnce`	Нужно ли гарантировать строго однократную доставку сообщения	true
`maxBufferPoolSize`	Максимальный размер пула буферов для транспортного протокола	524 888
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`maxRetryCycles`	После какого числа попыток считать сообщение «отравленным»	2
`queueTransferProtocol`	Транспортный протокол с очередями. Допустимы следующие протоколы: Native, Srmp и SrmpSecure. Native – это внутренний протокол MSMQ, а Srmp расшифровывается как Soap Reliable Messaging Protocol (Протокол надежной доставки сообщений Soap)	Native
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveErrorHandling`	Как обрабатывать «отравленные» сообщения. Допустимые значения: Drop, Fault, Move и Reject	Fault
`receiveRetryCount`	Максимальное число попыток отправить сообщение перед тем, как поместить его в очередь для повторной отправки	5
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`retryCycleDelay`	Пауза между циклами повторной отправки	00:10:00
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`timeToLive`	Сколько времени хранить сообщение перед тем, как поместить в очередь неотправленных писем	1.00:00:00

Таблица 4.11. Свойства привязки* `netMsmqBinding` *(окончание)

Имя атрибута	Описание	Значение по умолчанию
`useActiveDirectory`	Должен ли протокол разрешать имя компьютера с помощью Active Directory вместо DNS, NetBIOS или IP	false
`useMsmqTracing`	Включить ли режим трассировки MSMQ. Трассировочные сообщения отправляются в очередь отчета всякий раз, как сообщение поступает в очередь или покидает ее	false
`useSourceJournal`	Надо ли сохранять копию каждого сообщения в очереди журнала	false

Для работы с привязкой `netMsmqBinding` приложение `StockQuoteService`, представленное в листингах 4.2–4.4, придется модифицировать. Привязка `netMsmqBinding` поддерживает только односторонние операции (см. таблицу 4.2), а в исходном контракте об операции была задана коммуникация вида запрос-ответ (листинг 4.4). Вместо того чтобы создавать новый пример, мы изменим код `StockQuoteService`, чтобы продемонстрировать, как с помощью привязки `netMsmqBinding` можно реализовать двусторонний обмен сообщениями.

Для организации двусторонней коммуникации между клиентом и сервером нам понадобятся два односторонних контракта об операции. Следовательно, старые контракты придется переопределить. В листинге 4.24 показано, что нужно сделать. Во-первых, отметим, что мы вынесли контракты на запрос и ответ в два отдельных интерфейса: `IStockQuoteRequest` и `IStockQuoteResponse`. В каждом контракте операции односторонние. По контракту `IStockQuoteRequest` мы будем отправлять сообщения серверу, а по контракту `IStockQuoteResponse` сервер будет отправлять сообщения клиенту. Следовательно, на обеих сторонах будут размещены службы для приема сообщений.

Листинг 4.24. Интерфейсы IStockQuoteRequest, IstockQuoteResponse и класс StockQuoteRequestService

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Text;
using System.Transactions;

namespace EssentialWCF
{
   [ServiceContract]
   public interface IStockQuoteRequest
   {
      [OperationContract(IsOneWay = true)]
      void SendQuoteRequest(string symbol);
```

```
   }

   [ServiceContract]
   public interface IStockQuoteResponse
   {
      [OperationContract(IsOneWay = true)]
      void SendQuoteResponse(string symbol, double price);
   }

   public class StockQuoteRequestService : IStockQuoteRequest
   {
      public void SendQuoteRequest(string symbol)
      {
         double value;

         if (symbol == "MSFT")
               value = 31.15;
         else if (symbol == "YHOO")
               value = 28.10;
         else if (symbol == "GOOG")
               value = 450.75;
         else
               value = double.NaN;

         // Отправить ответ клиенту в отдельную очередь
         NetMsmqBinding msmqResponseBinding = new NetMsmqBinding();
         using (ChannelFactory<IStockQuoteResponse> cf =
➥new ChannelFactory<IStockQuoteResponse>("NetMsmqResponseClient"))
         {
            IStockQuoteResponse client = cf.CreateChannel();

            using (TransactionScope scope =
➥new TransactionScope(TransactionScopeOption.Required))
            {
               client.SendQuoteResponse(symbol, value);
               scope.Complete();
            }

            cf.Close();
         }
      }
   }
}
```

Следующее, чему надо уделить внимание при работе с привязкой `netMsmqBinding`, – это класс `ServiceHost`. В предыдущих примерах мы могли использовать один и тот же код этого класса с разными привязками. Теперь это уже не так. Обновленный код `ServiceHost` для размещения службы `StockServiceRequestService` приведен в листинге 4.25. Мы динамически создаем очередь MSMQ с именем `queueName`, указанным в конфигурационном файле. Это упрощает дело, поскольку наше приложение можно развернуть без дополнительного конфигурирования MSMQ.

Листинг 4.25. Класс ServiceHost для размещения службы StockQuoteRequestService

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Messaging;
using System.Text;
using System.ServiceModel;

namespace EssentialWCF
{
   internal class MyServiceHost
   {
      internal static string queueName = null;
      internal static ServiceHost myServiceHost = null;

      internal static void StartService()
      {
         queueName = ConfigurationManager.AppSettings["queueName"];

         if (!MessageQueue.Exists(queueName))
               MessageQueue.Create(queueName, true);

         myServiceHost =
➥new ServiceHost(typeof(EssentialWCF.StockQuoteRequestService));

         myServiceHost.Open();
      }

      internal static void StopService()
      {
         if (myServiceHost.State != CommunicationState.Closed)
               myServiceHost.Close();
      }
   }
}
```

В листинге 4.26 показан конфигурационный файл, в котором служба `StockQuoteRequestService` раскрывается с помощью привязки `netMsmqBinding`. Здесь же конфигурируется оконечная точка клиента для контракта `IStockQuoteResponse`, чтобы клиенту можно было посылать сообщения.

Листинг 4.26. Конфигурация службы для привязки netMsmqBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint
            address="net.msmq://localhost/private/stockquoteresponse"
            contract="EssentialWCF.IStockQuoteResponse"
            binding="netMsmqBinding"
            bindingConfiguration="NoMsmqSecurity"
```

```
            name="NetMsmqResponseClient"
         />
      </client>
      <services>
         <service name="EssentialWCF.StockQuoteRequestService">
            <endpoint
               address="net.msmq://localhost/private/stockquoterequest"
               contract="EssentialWCF.IStockQuoteRequest"
               bindingConfiguration="NoMsmqSecurity"
               binding="netMsmqBinding"
            />
         </service>
      </services>
      <bindings>
         <netMsmqBinding>
            <binding name="NoMsmqSecurity">
               <security mode="None" />
            </binding>
         </netMsmqBinding>
      </bindings>
   </system.serviceModel>
   <appSettings>
      <add key="queueName" value=".\private$\stockquoterequest" />
   </appSettings>
</configuration>
```

В клиентском приложении необходимо разместить службу, чтобы оно могло принимать ответы, а также сконфигурировать оконечную точку для отправки сообщений серверу. В листинге 4.27 приведен код класса `ServiceHost`, который используется на стороне клиента для размещения службы; он реализует контракт `IStockQuoteResponse`. Мы добавили код для динамического создания очереди, которую прослушивает клиент. И это снова позволяет нам обойтись без конфигурирования MSMQ.

Листинг 4.27. Клиентский класс ServiceHost для размещения StockQuoteResponseService

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Messaging;
using System.Text;
using System.ServiceModel;

namespace EssentialWCF
{
   internal class MyServiceHost
   {
      internal static ServiceHost myServiceHost = null;

      internal static void StartService()
      {
         string queueName =
```



```
            ConfigurationManager.AppSettings["queueName"];

            if (!MessageQueue.Exists(queueName))
                  MessageQueue.Create(queueName, true);

            myServiceHost =
                  new ServiceHost(typeof(EssentialWCF.Program));

            myServiceHost.Open();
      }

      internal static void StopService()
      {
         if (myServiceHost.State != CommunicationState.Closed)
               myServiceHost.Close();
      }
   }
}
```

В листинге 4.28 приведена реализация интерфейса `IStockQuoteResponse`. Клиент реализует этот интерфейс, а сервер пользуется им как функцией обратного вызова, которой посылается ответ. Здесь не задействованы имеющиеся в WCF механизмы дуплексного обмена; обратный вызов реализован с помощью отдельной односторонней привязки.

Листинг 4.28. Реализация интерфейса IStockQuoteResponse клиентом

```
using System;
using System.Collections.Generic;
using System.Messaging;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Text;
using System.Threading;
using System.Transactions;

namespace EssentialWCF
{
   public class Program : IStockQuoteResponse
   {
      private static AutoResetEvent waitForResponse;

      static void Main(string[] args)
      {
         // Запустить владельца отвечающей службы
         MyServiceHost.StartService();
         try
         {
            waitForResponse = new AutoResetEvent(false);

            // Послать запрос серверу
            using (ChannelFactory<IStockQuoteRequest> cf =
➥new ChannelFactory<IStockQuoteRequest>("NetMsmqRequestClient"))
            {
```

```
            IStockQuoteRequest client = cf.CreateChannel();

            using (TransactionScope scope =
➥new TransactionScope(TransactionScopeOption.Required))
            {
               client.SendQuoteRequest("MSFT");
               scope.Complete();
            }

            cf.Close();
         }

         waitForResponse.WaitOne();
      }
      finally
      {
         MyServiceHost.StopService();
      }

      Console.ReadLine();
   }

   #region Члены IStockQuoteResponseService

   public void SendQuoteResponse(string symbol, double price)
   {
      Console.WriteLine("{0} @ ${1}", symbol, price);
      waitForResponse.Set();
   }

   #endregion
  }
}
```

Чтобы пример с привязкой `netMsmqBinding` заработал, осталось только сконфигурировать клиента. В листинге 4.29 показан соответствующий конфигурационный файл, в котором содержится информация о размещении службы, реализующей `IStockQuoteResponse`, и о конфигурации оконечной точки для вызова службы `IStockQuoteRequest`.

Листинг 4.29. Конфигурация клиента с привязкой netMsmqBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="net.msmq://localhost/private/stockquoterequest"
                   contract="EssentialWCF.IStockQuoteRequest"
                   binding="netMsmqBinding"
                   bindingConfiguration="NoMsmqSecurity"
                   name="NetMsmqRequestClient"
         />
      </client>
      <services>
         <service name="EssentialWCF.Program">
            <endpoint
```



```
               address="net.msmq://localhost/private/stockquoteresponse"
               contract="EssentialWCF.IStockQuoteResponse"
               binding="netMsmqBinding"
               bindingConfiguration="NoMsmqSecurity"
            />
         </service>
      </services>
      <bindings>
         <netMsmqBinding>
            <binding name="NoMsmqSecurity">
               <security mode="None" />
            </binding>
         </netMsmqBinding>
      </bindings>
   </system.serviceModel>
   <appSettings>
      <add key="queueName" value=".\private$\stockquoteresponse" />
   </appSettings>
</configuration>
```

Привязка msmqIntegrationBinding

Привязка `msmqIntegrationBinding` применяется для коммуникации между приложением WCF и приложением, которое пользуется MSMQ непосредственно – например, с помощью классов из пространства имен `System.Messaging`. Это позволяет разработчикам применять WCF, не отказываясь от существующих приложений MSMQ. Привязка `msmqIntegrationBinding` отображает сообщения MSMQ на сообщения WCF. Достигается это путем обертывания сообщений MSMQ в обобщенный класс `MsmqMessage`, который находится в пространстве имен `System.ServiceModel.MsmqIntegration`. Объекты этого класса можно отправлять или применять согласно односторонним контрактам.

Ниже показан формат адреса для привязки `msmqIntegrationBinding`:

```
msmq.formatname:{MSMQ format name}
```

В адресе для MSMQ порт не указывается. Однако для работы MSMQ некоторые порты все же должны быть открыты, в частности 1801. В таблице 4.13 перечислены свойства, которые можно сконфигурировать для привязки `msmqIntegrationBinding`.

***Таблица 4.13. Свойства привязки* `msmqIntegrationBinding`**

Имя атрибута	Описание	Значение по умолчанию
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00
`customDeadLetterQueue`	Место нахождения очереди недоставленных писем для конкретного приложения. Недоставленным письмом называется сообщение, для которого истек срок хранения или доставка невозможна	Нет

Таблица 4.13. Свойства привязки* `msmqIntegrationBinding` *(продолжение)

Имя атрибута	Описание	Значение по умолчанию
`deadLetterQueue`	Какую очередь использовать для недоставленных писем. Возможные значения: None, System и Custom	None
`Durable`	Указывает, является ли очередь постоянной (durable) или временной	true
`exactlyOnce`	Нужно ли гарантировать строго однократную доставку сообщения	true
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`maxRetryCycles`	После какого числа попыток считать сообщение «отравленным»	2
`name`	Имя привязки.	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveErrorHandling`	Как обрабатывать «отравленные» сообщения. Допустимые значения: Drop, Fault, Move и Reject	Fault
`receiveRetryCount`	Максимальное число попыток отправить сообщение перед тем, как поместить его в очередь для повторной отправки	5
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`retryCycleDelay`	Пауза между циклами повторной отправки	00:10:00
`security`	Параметры безопасности привязки	Нет
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`serializationFormat`	Задает способ сериализации тела сообщения. Допустимые значения: XML, Binary, ActiveX, ByteArray и Stream	Xml
`timeToLive`	Сколько времени хранить сообщение перед тем, как поместить в очередь неотправленных писем	1.00:00:00
`useActiveDirectory`	Должен ли протокол разрешать имя компьютера с помощью Active Directory вместо DNS, NetBIOS или IP	false

Таблица 4.13. Свойства привязки* `msmqIntegrationBinding` *(окончание)

Имя атрибута	Описание	Значение по умолчанию
`useMsmqTracing`	Включить ли режим трассировки MSMQ. Трассировочные сообщения отправляются в очередь отчета всякий раз, как сообщение поступает в очередь или покидает ее	false
`useSourceJournal`	Надо ли сохранять копию каждого сообщения в очереди журнала	false

В листинге 4.30 показана минимальная конфигурация для раскрытия службы с помощью привязки `msmqIntegrationBinding`.

Листинг 4.30. Конфигурация службы для привязки msmqIntegrationBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteRequestService">
            <endpoint binding="msmqIntegrationBinding"
                      contract="EssentialWCF.IStockQuoteRequest”
               address="msmq.formatname:DIRECT=OS:.\private$\stockrequest" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

В листинге 4.31 показана минимальная конфигурация для потребления службы с помощью привязки `msmqIntegrationBinding`.

Листинг 4.30. Конфигурация клиента для привязки msmqIntegrationBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.HelloWorld">
            <endpoint binding="msmqIntegrationBinding"
                      contract="EssentialWCF.IStockQuoteRequestService”
         address="msmq.formatname:DIRECT=OS:.\private$\stockquoterequest" />
            </service>
      </services>
   </system.serviceModel>
</configuration>
```

Создание заказной привязки

Бывает, что ни одна из готовых привязок не отвечает предъявляемым к службе требованиям. Типичная ситуация – когда необходимы специальные условия безопасности и дополнительные транспортные механизмы, не поддерживаемые WCF. Например, в WCF отсутствует поддержка протокола UDP, хотя ее реализация имеется в примерах из SDK. Для таких случаев WCF предоставляет средства создания заказных привязок. Делать это можно как в коде, так и в конфигурационном файле. В коде заказная привязка создается с помощью класса `CustomBinding` из пространства имен `System.ServiceModel.Channels`. Этот класс раскрывает набор, в который вы можете добавлять элементы привязки и тем самым составлять новую привязку из имеющихся элементов. В листинге 4.32 демонстрируется создание заказной привязки чисто программным путем.

Листинг 4.32. Создание заказной привязки в коде

```
CustomBinding customBinding = new CustomBinding();
customBinding.Elements.Add(new BinaryMessageEncodingBindingElement());
customBinding.Elements.Add(new UdpBindingElement());
```

Заказную привязку можно создать и в конфигурационном файле с помощью элемента `customBinding`, как показано в листинге 4.33. При таком способе создания заказная привязка обязательно должна быть именованной.

Листинг 4.33. Создание заказной привязки в конфигурационном файле

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <customBinding>
            <binding name="CustomBinding">
               <binaryMessageEncoding />
               <udpTransport />
            </binding>
         </customBinding>
      </bindings>
   </system.serviceModel>
</configuration>
```

Обратите внимание, что в описании указаны транспортный протокол и кодировщик. Это все, что необходимо для создания заказной привязки. Кодировщик можно опускать, если для транспортного протокола имеется кодировщик по умолчанию. Для изменения способа работы привязки достаточно модифицировать несколько строк в коде или в конфигурации. Но будьте осторожны при работе с конфигурационным файлом, так как гарантий неизменности привязки нет. Создавайте привязки программно, если не планируете их модифицировать.

Показанную ниже конфигурацию можно использовать для приложения из листингов 4.3–4.4. В ней служба `StockQuoteService` раскрывается с помощью привязки `customBinding`, в которой используется протокол TCP с двоичным кодированием. Эта заказная привязка аналогична привязке `netTcpBinding`, но без поддержки надежной доставки, транзакций и безопасности.

Листинг 4.34. Конфигурация службы с привязкой customBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteService">
            <host>
               <baseAddresses>
                  <add baseAddress="net.tcp://localhost/stockquoteservice" />
               </baseAddresses>
            </host>
            <endpoint address=""
                        contract="EssentialWCF.IStockQuoteService"
                        binding="customBinding"
                        bindingConfiguration="customBinding"/>
         </service>
      </services>
      <bindings>
         <customBinding>
            <binding name="customBinding">
               <binaryMessageEncoding />
               <tcpTransport />
            </binding>
         </customBinding>
      </bindings>
   </system.serviceModel>
</configuration>
```

В листинге 4.35 показана конфигурация клиента, потребляющего эту службу с помощью той же привязки `customBinding`.

Листинг 4.35. Конфигурация клиента с привязкой customBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint address="net.tcp://localhost/stockquoteservice"
                     binding="customBinding"
                     bindingConfiguration="customBinding"
                     contract="EssentialWCF.IStockQuoteService">
         </endpoint>
      </client>
      <bindings>
         <customBinding>
            <binding name="customBinding">
               <binaryMessageEncoding />
               <tcpTransport />
            </binding>
         </customBinding>
```

```
      </bindings>
   </system.serviceModel>
/configuration>
```

Привязки, определяемые пользователем

Привязки можно определить целиком в коде или в конфигурационном файле либо создать, унаследовав класс `Binding`. Такая привязка называется определенной пользователем. Но вам все равно предстоит указывать поддерживаемые элементы привязки.

Основное различие между заказными и определяемыми пользователем привязками состоит в том, что последняя представляет собой экземпляр класса, выполняющего все шаги, необходимые для создания привязки. Этот подход стоит выбрать, если вы планируете повторно использовать привязку в нескольких приложениях. При этом авторы рекомендуют поддерживать в пользовательских привязках возможность создания из конфигурационного файла с помощью механизма *расширения привязки*. Для этого нужно создать новый класс, наследующий классу `BindingElementExtensionElement` из пространства имен `System.ServiceModel.Configuration`.

Предоставляйте для своей привязки также расширение. Мы настоятельно рекомендуем раскрывать свои привязки с помощью расширений, а не конфигурационного элемента `<customBinding>`. Это поможет избежать ошибок в конфигурационных файлах и сопутствующих им проблем.

Элементы привязки

WCF предоставляет многочисленные каналы и кодировщики, используемые в готовых привязках. Но эти каналы составлены из элементов, которыми вы можете пользоваться в собственных привязках. В этом разделе мы приведем перечень имеющихся в WCF элементов привязки и поясним их назначение.

Транспортные каналы

Ниже приведен перечень транспортных протоколов и ассоциированных с ними классов привязок, расширений привязок и элементов XML в конфигурационных файлах. Каждая таблица соответствует отдельному транспортному каналу, который можно включать в заказные привязки. Поддерживаются следующие транспорты: TCP, HTTP, именованные каналы, HTTP с поддержкой SSL/TLS-шифрования, MSMQ и пиринговые сети.

Транспортный канал TCP	Транспортный канал, основанный на протоколе TCP
Класс привязки	`TcpTransportBindingElement`
Расширение привязки	`TcpTransportElement`
Конфигурационный элемент	`<tcpTransport>`

Именованный канал	Транспортный канал, основанный на протоколе Named Pipe
Класс привязки	`NamedPipeBindingElement`
Расширение привязки	`NamedPipeTransportElement`
Конфигурационный элемент	`<namedPipeTransport>`
Транспортный канал HTTP	Транспортный канал, основанный на протоколе HTTP
Класс привязки	`HttpBindingElement`
Расширение привязки	`HttpTransportElement`
Конфигурационный элемент	`<httpTransport>`
Транспортный канал HTTPS	Транспортный канал, основанный на протоколе HTTPS
Класс привязки	`HttpBindingElement`
Расширение привязки	`HttpTransportElement`
Конфигурационный элемент	`<httpTransport>`
Транспортный канал MSMQ	Транспортный канал, основанный на протоколе MSMQ
Класс привязки	`MSMQTransportBindingElement`
Расширение привязки	`MSMQTransportElement`
Конфигурационный элемент	`<msmqTransport>`
Транспортный канал MSMQ Integration	Транспортный канал, основанный на протоколе MSMQ
Класс привязки	`MSMQIntegrationBindingElement`
Расширение привязки	`MSMQIntegrationBindingElement`
Конфигурационный элемент	`<msmqIntegration>`
Пиринговый транспортный канал	Транспортный канал, основанный на одноранговом протоколе
Класс привязки	`PeerBindingElement`
Расширение привязки	`PeerTransportElement`
Конфигурационный элемент	`<peerTransport>`

Транспортный канал UDP не входит в версию .NET 3.5. Его реализация предлагается в качестве примера в Windows SDK. Мы включили его, поскольку это востребованное дополнение к WCF.

Транспортный канал UDP	Транспортный канал, основанный на протоколе UDP
Класс привязки	`UdpTransportBindingElement`
Расширение привязки	`UdpTransportElement`
Конфигурационный элемент	`<udpTransport>`

Кодировщики

Ниже приведен перечень кодировщиков, включенных в состав WCF. Они описывают способ трансформации класса `Message` в поток байтов, передаваемых по транспортному каналу. В число кодировщиков входят Text, MTOM, Binary и JSON. Подробнее о них см. Главу 6 «Сериализация и кодирование».

Текстовое кодирование сообщений	Поддерживает текстовую кодировку сообщений SOAP
Класс привязки	`TextMessageEncodingBindingElement`
Расширение привязки	`TextMessageEncodingElement`
Конфигурационный элемент	`<textMessageEncoding>`
Двоичное кодирование сообщений	Поддерживает двоичную кодировку сообщений SOAP
Класс привязки	`BinaryMessageEncodingBindingElement`
Расширение привязки	`BinaryMessageEncodingElement`
Конфигурационный элемент	`<binaryMessageEncoding>`
Кодирование сообщений MTOM	Поддерживает MTOM-кодировку сообщений SOAP
Класс привязки	`MTOMMessageEncodingBindingElement`
Расширение привязки	`MTOMMessageEncodingElement`
Конфигурационный элемент	`<mtomMessageEncoding>`

В версию .NET 3.5, поставляемую вместе с Visual Studio 2008 включены кодировщики `JsonMessageEncoder` и `WebMessageEncoder`. Они реализованы в виде поведений, поэтому здесь не обсуждаются. Дополнительную информацию см. в главе 6 или 13.

Безопасность

Ниже перечислены применяемые в WCF протоколы безопасности. По большей части они предназначены для создания с помощью конфигурационного элемента `<security>` или статических методов класса `SecurityBindingElement`. Рекомендуется создавать их именно таким образом, потому что класс `SecurityBindingElement` в какой-то мере решает за вас задачу конфигурирования элементов привязки, предлагая статические методы, которыми можно пользоваться для создания других привязок безопасности.

Асимметричная безопасность	Безопасность канала на базе асимметричного шифрования
Класс привязки	`AsymmetricSecurityBindingElement`
Расширение привязки	`SecurityElement`
Конфигурационный элемент	`<security>`

Симметричная безопасность	Безопасность канала на базе симметричного шифрования
Класс привязки	`SymmetricSecurityBindingElement`
Расширение привязки	`SecurityElement`
Конфигурационный элемент	`<security>`
Транспортная безопасность	Смешанный режим безопасности
Класс привязки	`TransportSecurityBindingElement`
Расширение привязки	`SecurityElement`
Конфигурационный элемент	`<security>`

Модификаторы и вспомогательные средства для транспортных протоколов

Ниже приведен перечень элементов привязки, содержащих добавления или вспомогательные средства для транспортных протоколов. В WCF к привязкам, в которых используются потоковые протоколы, например TCP и именованные каналы, могут применяться модификаторы. Так, элемент `SslStreamSecurityBindingElement` модифицирует безопасность канала, добавляя SSL-поток.

Обнаружение равноправных клиентов (пиров) по протоколу PNRP	Разрешение имен пиров по протоколу PNRP
Класс привязки	`PnrpPeerResolverBindingElement`
Расширение привязки	`PnrpPeerResolverElement`
Конфигурационный элемент	`<pnrpPeerResolver>`
Безопасность с помощью SSL-потока	Безопасность канала с помощью SSL-потока
Класс привязки	`SslStreamSecurityBindingElement`
Расширение привязки	`SslStreamSecurityElement`
Конфигурационный элемент	`<sslStreamSecurity>`
Безопасность с помощью потока Windows	Используется для задания настроек безопасности с помощью потока Windows
Класс привязки	`WindowsStreamSecurityBindingElement`
Расширение привязки	`WindowsStreamSecurityElement`
Конфигурационный элемент	`<windowsStreamSecurity>`

Изменение канальной формы

Ниже перечислены элементы привязки, изменяющие форму стека каналов. Они позволяют изменить способа обмена сообщениями по каналу. Дополнительную информацию о канальных формах и их изменении см. в разделе «Канальные формы» главы 3.

Изменение формы на дуплексную	Позволяет осуществлять дуплексный обмен сообщениями по транспортным каналам, которые изначально не поддерживают дуплексных коммуникаций
Класс привязки	`CompositeDuplexBindingElement`
Расширение привязки	`CompositeDuplexElement`
Конфигурационный элемент	`<compositeDuplex>`
Изменение формы на одностороннюю	Позволяет осуществлять одностороннюю коммуникацию по транспортным каналам, которые изначально такой режим не поддерживают
Класс привязки	`OneWayBindingElement`
Расширение привязки	`OneWayElement`
Конфигурационный элемент	`<oneWay>`

Другие протоколы

Ниже приведен перечень элементов привязки, которые добавляют поддержку других протоколов, например, для обеспечения транзакционности и надежной доставки.

Надежные сеансы	Поддержка доставки сообщений SOAP по порядку и без дубликатов
Класс привязки	`ReliableSessionBindingElement`
Расширение привязки	`ReliableSessionElement`
Конфигурационный элемент	`<reliableSession>`
Поток транзакций	Поддержка потока транзакций от клиента к серверу
Класс привязки	`TransactionFlowBindingElement`
Расширение привязки	`TransactionFlowElement`
Конфигурационный элемент	`<transactionFlow>`

Раскрытие контракта о службе с помощью нескольких привязок

В предыдущих разделах мы продемонстрировали, как раскрывать службы с помощью привязок `netTcpBinding` и `wsHttpBinding`. Каждая из них применяется для поддержки коммуникаций определенного вида. Например, привязка `netTcpBinding` оптимизирована для коммуникации между .NET-приложениями, `wsHttpBinding` – для коммуникации между приложениями на разных платформах с помощью Web-служб, а `basicHttpBinding` – для коммуникации с помощью Web-служб, не поддерживающих новейшие протоколы.

Заведя в службе несколько оконечных точек, вы сможете сконфигурировать ее так, чтобы она раскрывалась с помощью нескольких привязок, как описано в разделе «Службы с несколькими контрактами и оконечными точками» главы 2. Это означает, что клиент сможет соединяться со службой, применяя наиболее оптимальную из поддерживаемых привязок. Например, .NET-приложение будет обращаться к службе с помощью привязки `netTcpBinding`, Java-приложение – с помощью привязки `wsHttpBinding`, а более старые клиенты – с помощью привязки `basicHttpBinding`.

WCF достигает этого, абстрагируя низкоуровневые механизмы коммуникации и позволяя разработчику сосредоточиться на создании служб. Как именно они раскрываются, не имеет значениея, коль скоро привязка поддерживает необходимые приложению возможности.

Применяйте несколько привязок при создании интероперабельных служб. Возможность раскрывать службу с помощью нескольких привязок заметно повышает ее гибкость. Различные привязки можно использовать одновременно. Поэтому к службе могут обращаться клиенты, написанные как на платформе WCF, так и на других платформах без потери производительности на обеспечение интероперабельности. Например, можно раскрыть одну и ту же службу с помощью привязок `netTcpBinding` и `wsHttpBinding`. WCF-клиенты будут пользоваться привязкой `netTcpBinding`, а все прочие (например, написанные на Java) – привязкой `wsHttpBinding`. Не забывайте только, что все используемые привязки должны поддерживать необходимые приложению функции. Не имеет смысла раскрывать службу, нуждающуюся в транзакциях, с помощью привязки, которая их не поддерживает.

Раскрытие службы клиентам, работающим на платформах .NET и Java, – это лишь один пример использования нескольких привязок. Другой – обращение к службе из Web-браузера и из приложения для Windows, написанного на одном из .NET-совместимых языков. В листинге 4.36 приведен пример раскрытия службы с помощью нескольких привязок.

Листинг 4.36. Конфигурация службы с несколькими привязками

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockQuoteService">
            <endpoint   binding="wsHttpBinding"
                        contract="EssentialWCF.IStockQuoteService"
                        address=”http://localhost/wshttpendpoint” />
            <endpoint   binding="netTcpBinding"
                        contract="EssentialWCF. IStockQuoteService"
                        address=”net.tcp://localhost/nettcpendpoint” />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

Резюме

Архитектура каналов позволяет построить унифицированную модель разработки распределенных приложений. Службы можно создавать независимо от транспортных протоколов и кодировок, а, стало быть, они могут поддерживать различные виды коммуникации. Привязки – это заранее сконфигурированные стеки каналов для коммуникаций определенного вида. В комплекте с WCF поставляются девять готовых привязок.

Ниже сформулированы основные принципы работы с каналами и привязками в WCF:

- привязка `netTcpBinding` предназначена для межмашинных коммуникаций между .NET-приложениями;
- привязка `netNamedPipeBinding` предназначена для локальных коммуникаций между .NET-приложениями – как между несколькими процессами, так и внутри одного процесса (междоменных и внутридоменных);
- привязка `basicHttpBinding` служит для поддержки унаследованных Web-служб, основанных на спецификации WS-I Basic Profile 1.1. Как правило, она используется для потребления ASMX-служб, созданных в ASP.NET, но годится и для предоставления служб клиентам на платформе .NET 2.0, не нуждающимся в спецификациях WS-*;
- привязки `ws2007HttpBinding` и `wsHttpBinding` применяются для создания Web-служб, поддерживающих спецификации WS-*. При создания новых служб в WCF рекомендуется применять привязку `ws2007HttpBinding`, поскольку она поддерживает новейшие стандарты в области безопасности, надежной доставки сообщений и транзакционности;
- есть еще три привязки, основанные на Web-службах: `wsDualHttpBinding`, `wsFederationHttpBinding` и `ws2007FederationHttpBinding`. Пользуйтесь ими, если возникает потребность в дуплексном обмене сообщениями по протоколу HTTP или интегрированной безопасности соответственно. Привязка `ws2007FederationHttpBinding` входит в состав .NET 3.5 и дополнительно поддерживает спецификацию WSS SAML Token Profile 1.1;
- привязка `netMsmqBinding` применяется для разработки несвязанных приложений с помощью технологии Microsoft Message Queue (MSMQ);
- привязка `msmqIntegrationBinding` служит для интеграции с существующими приложениями на базе MSMQ;
- стек каналов в WCF можно конструировать самостоятельно, что позволяет создавать заказные привязки. С их помощью можно организовать коммуникацию, не поддерживаемую ни одной из готовых привязок;
- WCF поддерживает раскрытие служб с помощью нескольких привязок. Это позволяет обеспечить оптимальную работу служб с разнородными клиентами;
- пользуйтесь готовыми привязками, если они отвечают вашим требованиям; в противном случае класс `CustomBinding` поможет создать заказную привязку.

Глава 5. Поведения

Поведения – это классы, которые влияют на работу WCF на этапе выполнения. Они вызываются, когда инициализируется исполняющая среда WCF на стороне клиента или сервера, а также в процессе передачи сообщений между ними. Поскольку поведения работают в критически важные моменты, их можно использовать для реализации многих встроенных в WCF функций. Кроме того, они представляют собой важный механизм расширения WCF.

Например, класс `ServiceHost` помимо передачи сообщений нужной операции отвечает за создание экземпляров и аспекты сервера, связанные с параллелизмом. Когда служба получает сообщение и передает его тому или иному методу классу, должен ли `ServiceHost` каждый раз создавать новый экземпляр этого класса или может повторно использовать ранее созданные экземпляры? А при вызове метода следует ли включать его в состав транзакции? Оба эти аспекта задаются с помощью поведений и учитываются в ходе инициализации.

Существует три основных типа поведений. *Поведения службы* работают на уровне службы и имеют доступ ко всем оконечным точкам. Они управляют такими вещами, как создание экземпляров и транзакции. Поведения службы могут также использоваться для авторизации и аудита. Область действия *поведений оконечной точки* ограничена одной оконечной точкой службы. Они хорошо приспособлены для инспектирования входящих и исходящих сообщений и выполнения над ними тех или иных действий. *Поведения операции* функционируют на уровне одной операции и подходят для сериализации, управления потоком транзакций и обработки параметров операции. Дополнительно в WCF определены *поведения обратного вызова*, которые работают аналогично поведениям службы, но контролируют оконечные точки, созданные на стороне клиента в процессе дуплексной коммуникации.

Чтобы разобраться в том, как используются поведения, будет полезно рассмотреть процедуру инициализации исполняющей среды. На стороне клиента этим занимается класс `ChannelFactory`, а на стороне сервера – класс `ServiceHost`. Оба класса выполняют сходные функции:

1. Получить на входе тип .NET и прочитать информацию из его атрибутов.
2. Загрузить конфигурацию из файла `app.config` или `web.config`. Класс `ChannelFactory` на стороне клиента интересуется прежде всего информацией о привязках, а класс `ServiceHost` на стороне сервера – контрактом и информацией о привязках.
3. Инициализировать структуру данных `ServiceDescription` для исполняющей среды.

4. Начать обмен данными. Класс `ChannelFactory` на стороне клиента использует канал для соединения со службой, а класс `ServiceHost` на стороне сервера открывает канал и прослушивает его в ожидании сообщений.

Информация о поведениях, считываемая на шаге 1, представлена в виде атрибутов, например, `[ServiceBehavior (TransactionTimeout="00:00:30")]`. На шаге 2 информация о поведениях определена в конфигурации, например, в виде элемента `<transactionTimeout="00:00:30">` в файле `app.config`. На шаге 3 классы `ChannelFactory` и `ServiceHost` инициализируют исполняющую среду WCF и отвечают за включение в нее поведений, обнаруженных на шагах 1 и 2. Кроме того, на шаге 3 можно добавить поведения и вручную, например: `Endpoint.Behaviors.Add(new MyBehavior())`.

Поведения могут работать не только на стадии инициализации, но и сразу после приема и перед отправкой сообщения. На стороне клиента поведения могут выполнять три функции:

- **Инспекция параметров**. Осмотреть и/или изменить данные в .NET-представлении перед тем, как они преобразуются в формат XML.
- **Форматирование сообщения**. Осмотреть и/или изменить данные в процессе преобразования из типа .NET в XML.
- **Инспекция сообщения**. Осмотреть и/или изменить данные в XML-представлении перед тем, как преобразовать его в типы .NET.

На стороне сервера поведения позволяют реализовать два дополнительных сценария:

- **Выбор операции**. На уровне службы можно проинспектировать входящее сообщение и решить, какую операцию вызывать.
- **Вызов операции**. На уровне операции вызвать тот или иной метод класса.

На рис. 5.1 изображен поток управления между различными элементами поведений, которые вызываются в процессе передачи сообщений от клиента к серверу и обратно. Когда клиентское приложение обращается к операции `GetPrices(…)`, вызываются инспектор параметров (Parameter Inspector) и форматер сообщений (Message Formatter), которым передаются параметры в формате .NET. Далее, все еще на стороне клиента, вызывается инспектор сообщений (Message Inspector), которому передается XML-сообщение. Когда сообщение прибывает в канал на стороне службы, вызываются инспектор сообщений и селектор операций (Operation Selector), которым поступившее сообщение передается для анализа и решения о том, какой операции его передать. Затем вызывается форматер сообщений, который преобразует сообщение в тип .NET, и инспектор параметров, которому сообщение передается уже в .NET-представлении. Наконец, вызывается активатор операций (Operation Invoker), который вызывает нужный метод класса-получателя, выполняя по ходу инициализацию и очистку.

Как видно из рис. 5.1, существует много мест, в которых поведения могут исследовать и изменить поток сообщений. Они вполне могут повлиять на общую производительность службы.

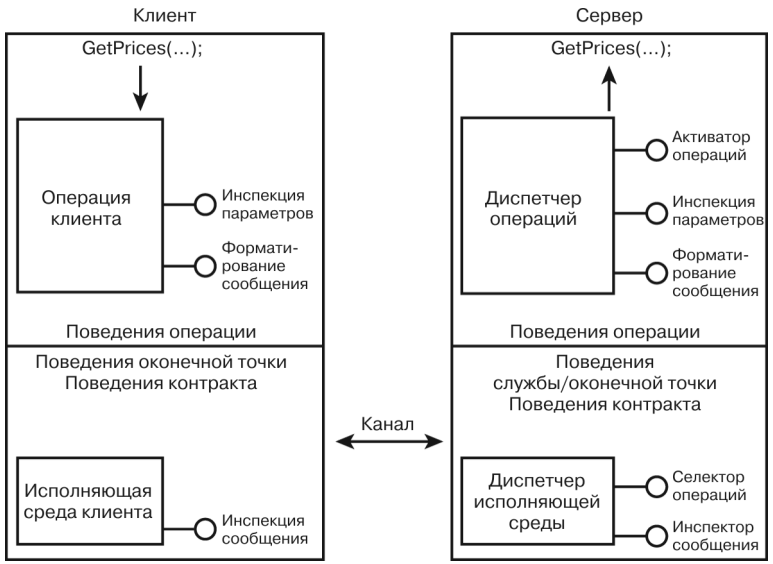

Рис. 5.1. Элементы поведений

Параллелизм и создание экземпляров (поведение службы)

Степень параллелизма – это мера оценки количества задач, которые могут выполняться одновременно; измеряется она в задачах (запросах, работах, транзакциях и т.п.). *Время выполнения* – это оценка того, сколько времени нужно для полного завершения задачи; оно измеряется в единицах времени (миллисекундах, секундах и т.д.). *Пропускная способность* – это оценка того, сколько задач завершается в заданный промежуток времени; измеряется оно в задачах в единицу времени (запросы/сек, транзакции/мин и т.д.). Пропускная способность – это функция от степени параллелизма и времени выполнения.

Повысить пропускную способность можно двумя способами: либо уменьшить время выполнения, либо увеличить степень параллелизма. Уменьшить время выполнения одной задачи можно либо путем изменения внутреннего алгоритма, либо за счет добавления аппаратных ресурсов, тут WCF мало чем может помочь. Степень параллелизма можно увеличить, выполняя больше задач одновременно. Для управления параллелизмом в WCF имеется два поведения: `InstanceContextMode` и `ConcurrencyMode`.

Поведение службы `InstanceContextMode` применяется для управления созданием экземпляров и может принимать одно из трех значений:

- **Single**. Один экземпляр класса службы обрабатывает все входящие запросы. Тем самым реализуется паттерн Singleton (синглет, одиночка);

Таблица 5.1. Сочетание поведений InstanceContextMode и ConcurrencyMode

	InstanceContextMode.Single	InstanceContextMode.PerCall	InstanceContextMode.PerSession
ConcurrencyMode.Single (по умолчанию)	Синглет – для обработки запросов создается только один экземпляр и только один поток. Пока текущий запрос обрабатывается, все последующие ставятся в очередь и обрабатываются в порядке FIFO (первым пришел, первым обслужен)	При каждом обращении создается новый экземпляр. Режим параллелизма не имеет значения, так как в каждом экземпляре собственный поток выполнения	На каждый сеанс связи с клиентом создается один экземпляр и для обработки всех запросов в этом сеансе используется один поток. Если клиент делает несколько асинхронных запросов, они ставятся в очередь и обрабатываются в порядке FIFO
ConcurrencyMode.Reentrant	Синглет – для обработки запросов создается только один экземпляр и только один поток. Пока текущий запрос обрабатывается, все последующие ставятся в очередь и обрабатываются в порядке FIFO (первым пришел, первым обслужен)	При каждом обращении создается новый экземпляр. Режим параллелизма не имеет значения, так как в каждом экземпляре собственный поток выполнения	На каждый сеанс связи с клиентом создается один экземпляр и для обработки всех запросов в этом сеансе используется один поток. Если клиент делает несколько асинхронных запросов, они ставятся в очередь и обрабатываются в порядке FIFO. Единственный поток может покинуть метод, сделать что-то еще, а потом вернуться. Такое возможно в случае применения техники асинхронного кодирования на стороне сервера
ConcurrencyMode.Multiple	Создается один экземпляр, но с ним могут параллельно работать несколько потоков. Члены класса должны быть защищены с помощью примитивов синхронизации, поскольку к ним возможны одновременные обращения из разных потоков	При каждом обращении создается новый экземпляр. Режим параллелизма не имеет значения, так как в каждом экземпляре собственный поток выполнения	На каждый сеанс связи с клиентом создается один экземпляр, но с ним могут параллельно работать несколько потоков. Если клиент делает несколько асинхронных вызовов, все они обрабатываются параллельно. Члены класса должны быть защищены с помощью примитивов синхронизации, поскольку к ним возможны одновременные обращения из разных потоков

- **PerCall**. На каждый входящий запрос создается отдельный экземпляр класса службы.
- **PerSession**. Один экземпляр класса службы создается на каждый сеанс связи с клиентом. При использовании безсеансовых каналов, все вызовы службы ведут себя как в случае `PerCall`, даже если для поведения `InstanceContextMode` установлено значение `PerSession`.

Подразумеваемое по умолчанию значение `InstanceContextMode.PerSession` говорит WCF о том, что нужно создавать новый экземпляр класса службы (на самом деле, прокси-класс) для каждого пользователя и переходить в режим `PerCall`, если используется безсеансовая привязка.

Поведение службы `ConcurrencyMode` применяется для управления параллелизмом внутри одного экземпляра службы. Принимаемое по умолчанию значение `ConcurrencyMode.Single` означает, что WCF будет запускать только один поток *в каждом экземпляре* класса службы. Всего есть три возможных значения:

- **Single**. В каждый момент времени к классу службы может обращаться только один поток. Это самый безопасный режим, поскольку операции службы могут не заботиться о безопасности относительно потоков.
- **Reentrant**. В каждый момент времени к классу службы может обращаться только один поток, но этот поток может покидать класс, а затем снова возвращаться в него.
- **Multiple**. К классу службы одновременно могут обращаться несколько потоков. В этом случае класс должен быть написан с учетом безопасности относительно потоков.

Совместно поведения `InstanceContextMode` и `ConcurrencyMode` позволяют настраивать стратегию порождения экземпляров и степень параллелизма для достижений нужных показателей производительности.

Параллелизм и создание экземпляров по умолчанию для безсеансовых привязок

В листинге 5.1 приведена служба, для которой не заданы поведения, описывающие параллелизм и создание экземпляров, то есть WCF примет значения по умолчанию: `ConcurrencyMode.Single` и `InstanceContextMode.PerSession`. Если такие настройки действуют для безсеансовой привязки, например `basicHttpBinding`, то WCF создает новый экземпляр службы для каждого входящего запроса и исполнять его код в одном потоке. В данном примере служба ждет пять секунд и потом возвращает управление.

Листинг 5.1. Служба с принятыми по умолчанию поведениями, описывающими параллелизм и создание экземпляров

```
[ServiceContract]
public interface IStockService
{
```

```
   [OperationContract]
   double GetPrice(string ticker);
}

public class StockService : IStockService
{
   StockService()
   {
      Console.WriteLine("{0}:
                    В потоке создан новый экземпляр StockService",
                    System.DateTime.Now);
   }
   public double GetPrice(string ticker)
   {
      Console.WriteLine("{0}: GetPrice вызван в потоке {1}",
                    System.DateTime.Now,
                    Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(5000);
      return 94.85;
   }
}
```

В листинге 5.2 показан код клиента, вызывающего метод `GetPrice` три раза. Все три вызова делаются асинхронно, после чего клиент дожидается возврата всех результатов и завершается.

Листинг 5.2. Клиент, асинхронно обращающийся к службе

```
class Program
{
   static int c = 0;
   static void Main(string[] args)
   {
      StockServiceClient proxy = new StockServiceClient();
      for (int i=0; i<3; i++)
      {
         Console.WriteLine("{0}: Вызывается GetPrice",
                                 System.DateTime.Now);
         proxy.BeginGetPrice("MSFT", GetPriceCallback, proxy);
         Thread.Sleep(100); // чтобы было проще разобраться в ответных
сообщениях
         Interlocked.Increment(ref c);
      }
      while (c > 0) // ждать получения всех ответов
      {
         Thread.Sleep(100);
      }
   }

   static void GetPriceCallback(IAsyncResult ar)
   {
      double price =
                ((StockServiceClient)ar.AsyncState).EndGetPrice(ar);
      Console.WriteLine("{0}: Цена:{1}", System.DateTime.Now,
                               price);
      Interlocked.Decrement(ref c);
   }
}
```


На рис. 5.2 показано, что выводит клиент (слева) и служба (справа). Видно, что клиент одновременно отправляет все три запроса, а результаты возвращаются спустя пять секунд. Распечатка вывода службы показывает, что для каждого запроса клиента был создан новый экземпляр класса службы, и каждый запрос обрабатывался в отдельном потоке. Поскольку привязка `basicHttpBinding` не поддерживает сеансов, принятое по умолчанию поведение `PerSession` ведет себя в этом примере как `PerCall`. Поведение `InstanceContextMode.PerSession` говорит WCF, что нужно создавать новый экземпляр для каждого запроса, а поведение `ConcurrencyMode.Single` – что в каждом экземпляре должен работать только один поток.

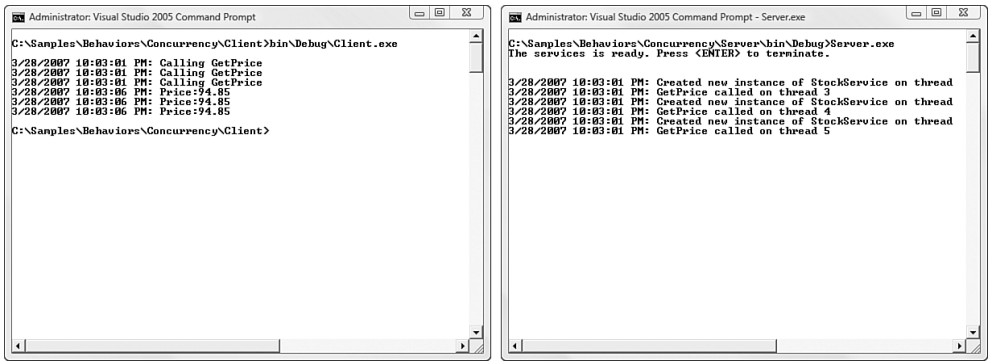

Рис. 5.2. Вывод клиента и службы в случае принимаемых по умолчанию поведений `ConcurrencyMode` и `InstanceContextMode` для безсеансовой привязки

Многопоточность в одном экземпляре

Принимаемое по умолчанию значение поведения `InstanceContextMode` говорит WCF о том, что нужно создавать новый экземпляр службы для каждого запроса. Но во многих случаях такой подход не оптимален. Например, если процедура инициализации службы занимает много времени (скажем, конструктор загружает данные из базы или строит большую структуру данных), было бы неэффективно создавать для каждого запроса новый экземпляр. Чтобы создать единственный экземпляр службы, разделяемый всеми параллельными потоками, следует использовать режим `InstanceContextMode.Single` в сочетании с `ConcurrencyMode.Multiple`. Поведение `InstanceContextMode.Single` означает, что нужно создать только один экземпляр, а `ConcurrencyMode.Multiple` – что к этому экземпляру могут одновременно обращаться несколько потоков. Это заметно улучшает масштабируемость, однако в коде службы необходимо применять синхронизацию для защиты локальных переменных.

В листинге 5.3 приведен код службы, в которой используются поведения `InstanceContextMode.Single` и `ConcurrencyMode.Multiple`. Обратите внимание, что атрибут `ServiceBehavior` задан для класса, а не интерфейса. Связано это с тем, что данный атрибут модифицирует поведение самой службы, а не ее контракта.

Листинг 5.3. Служба, в которой используются поведения `InstanceContextMode.Single` и `ConcurrencyMode.Multiple`

```
[ServiceContract]
public interface IStockService
{
   [OperationContract]
   double GetPrice(string ticker);
}

public class StockService : IStockService
{
   [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                    ConcurrencyMode = ConcurrencyMode.Multiple)]
   StockService()
   {
      Console.WriteLine("{0}:
               В потоке создан новый экземпляр StockService",
               System.DateTime.Now);
   }
   public double GetPrice(string ticker)
   {
      Console.WriteLine("{0}: GetPrice вызван в потоке {1}",
               System.DateTime.Now,
               Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(5000);
      return 94.85;
   }
}
```

На рис. 5.3 показано, что выводит клиент (слева) и служба (справа). Как и в предыдущем примере, клиент отправляет все три запроса одновременно, а результаты возвращаются спустя пять секунд. А распечатка вывода службы показывает, что был создан только один экземпляр класса службы, но при этом, как и раньше, каждый запрос клиента обрабатывался в отдельном потоке. Поведение `InstanceContextMode.Single` заставило WCF порождать единственный экземпляр класса, а поведение `ConcurrencyMode.Multiple` разрешило нескольким потокам одновременно обращаться к этому экземпляру.

Реализация синглета

Бывают случаи, когда должен существовать только один экземпляр службы, и этот экземпляр должен быть однопоточным. Все задачи должны выполняться строго поочередно (первым пришел, первым обслужен) без какого бы то ни было параллелизма. Хотя пропускная способность при этом заметно снижается, но ста-

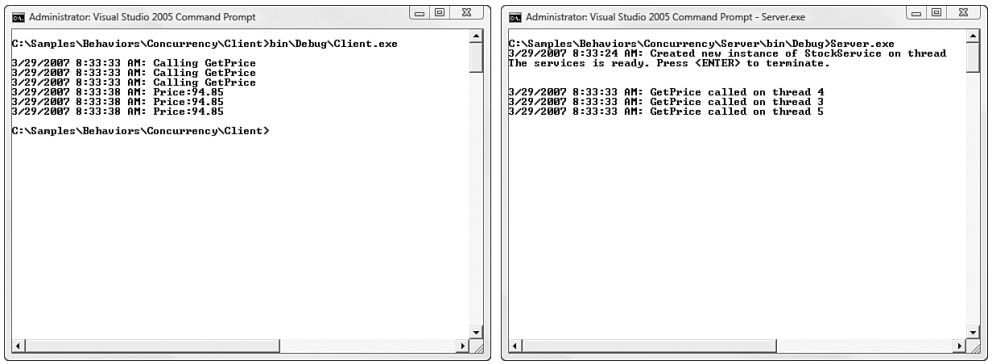

Рис. 5.3. Вывод клиента и службы в случае поведений `ConcurrencyMode.Multiple` и `InstanceContextMode.Single`

новятся возможными сценарии, когда все клиенты должны разделять некое общее состояние, а адекватных механизмов блокировки не существует.

Чтобы создать единственный однопоточный экземпляр службы, необходимо задать поведение `InstanceContextMode.Single` в сочетании с `ConcurrencyMode.Single`. В режиме `InstanceContextMode.Single` создается только один экземпляр, а в режиме `ConcurrencyMode.Single` WCF будет исполнять этот экземпляр в единственном потоке. В результате все запросы будут обрабатываться строго в порядке поступления (FIFO).

В листинге 5.4 приведен код службы, в которой используются поведения `InstanceContextMode.Single` и `ConcurrencyMode.Single`.

Листинг 5.4. Служба, в которой используются поведения `InstanceContextMode.Single` и `ConcurrencyMode.Single`

```
[ServiceContract]
public interface IStockService
{
   [OperationContract]
   double GetPrice(string ticker);
}

public class StockService : IStockService
{
   [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                    ConcurrencyMode = ConcurrencyMode.Single)]
   StockService()
   {
      Console.WriteLine("{0}:
               В потоке создан новый экземпляр StockService",
               System.DateTime.Now);
   }
   public double GetPrice(string ticker)
   {
      Console.WriteLine("{0}: GetPrice вызван в потоке {1}",
```

```
                System.DateTime.Now,
                Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(5000);
        return 94.85;
    }
}
```

На рис. 5.4 показано, что выводит клиент (слева) и служба (справа). Клиент отправляет три асинхронных запроса, но служба обрабатывает только один запрос за пять секунд. Из распечатки видно, что был создан только один класс службы. Поскольку задано поведение `InstanceContextMode.Single`, WCF порождает единственный экземпляр класса, а поведение `ConcurrencyMode.Single` разрешает обращаться к этому экземпляру только одному потоку в каждый момент времени. Отметим, что `ConcurrencyMode.Single` не ограничивает общее количество потоков в службе; требуется лишь, чтобы к каждому экземпляру обращался только один поток.

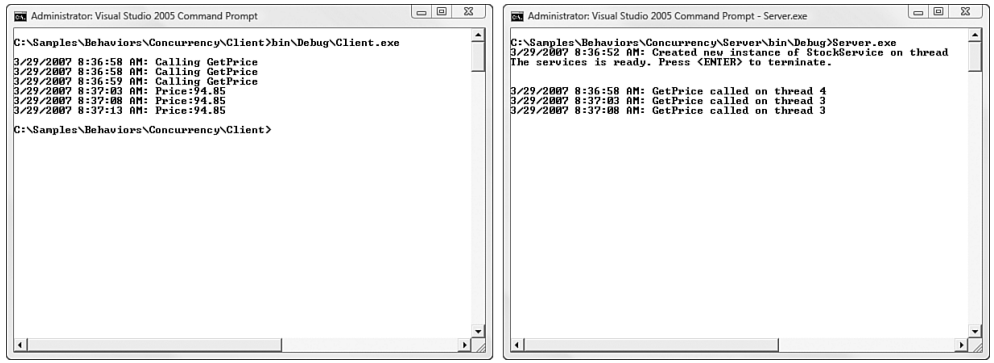

Рис. 5.4. Вывод клиента и службы в случае поведений `ConcurrencyMode.Single` и `InstanceContextMode.Single`

Сеансовые экземпляры

Сеансы широко используются для запоминания состояния каждого пользователя в распределенных приложениях. При работе с Web-сайтами или Web-приложениями принято сохранять состояние в сеансах. В этих случаях между сеансами и пользователями имеется взаимно однозначное соответствие. Аналогичную идею WCF поддерживает для служб. С помощью поведения `InstanceContextMode.PerSession` можно заставить WCF создавать экземпляр службы для каждого сеанса.

Для реализации сеансовых экземпляров службы нужно сделать две вещи: разрешить создание сеансов на уровне контракта и на уровне службы.

На уровне контракта сеансы активируются с помощью поведения `SessionMode`, задаваемого в контракте о службе. Оно может принимать значения `Allowed` (разрешено), `NotAllowed` (не разрешено) и `Required` (обязательно). Хотя сеансы задаются на уровне контракта, фактически они реализуются каналом, который определяется элементами привязки. Поэтому при запуске службы поведение проверяет совместимость контракта и канала. Например, если канал требует наличия сеансов, а используемая привязка (скажем, `basicHttpBinding`) их не поддерживает, то в момент запуска службы поведение контракта возбудит исключение.

> **Сеансы экземпляра – не то же самое, что надежные сеансы.** Сеансовые экземпляры службы не следует путать с другой функцией WCF – надежными сеансами (Reliable Sessions). Она реализует спецификацию WS-RM и применяется для надежной упорядоченной доставки сообщений между оконечными точками с помощью промежуточных посредников. Эта функция не имеет ничего общего с поведениями, управляющими параллелизмом и созданием объектов.

На уровне службы сеансы активируются путем задания поведения `InstanceContextMode.PerSession`. В результате WCF будет создавать экземпляр класса службы для каждого сеанса подключения к ней.

В листинге 5.5 приведен код службы с поведением `InstanceContextMode.PerSession`, в которой создается один экземпляр для каждого сеанса. Помимо возврата цена акции, эта служба также подсчитывает, сколько раз она вызывалась. Поскольку для поведения `InstanceContextMode` задано значение `PerSession`, клиент видит, сколько раз служба вызывалась в данном сеансе, а не общее число обращений к ней. Если бы было задано поведение `InstanceContextMode.Single`, то клиент видел бы общее число обращений за все время работы службы, а в случае поведения `InstanceContextMode.PerCall` счетчик вызовов всегда был бы равен 1.

Отметим, что для синхронизации доступа к переменной класса `n_calls` используется блокировка. Это необходимо, потому что в случае поведения `ConcurrencyMode.Multiple` обращаться к сеансовому экземпляру могут сразу несколько потоков.

Листинг 5.5. Создание сеансового экземпляра

```
[DataContract]
class StockPrice
{
   [DataMember]
   public double price;
   [DataMember]
   public int calls;
}

[ServiceContract(SessionMode = SessionMode.Required)]
interface IStockService
{
   [OperationContract]
   StockPrice GetPrice(string ticker);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession,
```

```
                        ConcurrencyMode = ConcurrencyMode.Multiple)]
class StockService : IStockService
{
   System.Object lockThis = new System.Object();
   private int n_Calls = 0;
   StockService()
   {
      Console.WriteLine("{0}:
               В потоке создан новый экземпляр StockService",
               System.DateTime.Now);
   }
   public StockPrice GetPrice(string ticker)
   {
      StockPrice p = new StockPrice();
      Console.WriteLine("{0}: GetPrice вызван в потоке {1}",
               System.DateTime.Now,
               Thread.CurrentThread.ManagedThreadId);
      p.price = 94.85;
      lock (lockThis)
      {
         p.calls = ++n_Calls;
      }
      Thread.Sleep(5000);
      return (p);
   }
}
```

На рис. 5.5 показано, что выводит клиент (слева) и служба (справа). В левой части два окна, потому что одновременно запущено два клиента. Каждый клиент синхронно вызывает метод `GetPrice` три раза. Из распечатки работы службы видно, что было создано два экземпляра – по одному на каждый сеанс. Отметим, что каждый клиент видит, сколько запросов отправил именно он, но не общее число запросов. Это объясняется тем, что счетчик `n_Calls` хранится в сеансовом экземпляре служб и инициализируется нулем. Если изменить поведение `InstanceContextMode` на `PerCall`, то каждый клиент всегда будет видеть значение 1, а если на `Single` – то счетчик будет возрастать от 1 до 6, поскольку учитываются вызовы от обоих клиентов.

Управление количеством одновременно работающих экземпляров

По умолчанию для обслуживания входящих запросов WCF запускает столько экземпляров, сколько возможно. Если в контракте о службе явно не задано поведение, описывающее создание экземпляров и параллелизм, то будет создаваться экземпляр службы для каждого входящего запроса, и для их обслуживания будут выделяться отдельные потоки. В целом это неплохо с точки зрения производительности и масштабируемости, поскольку пропускная способность растет при наращивании аппаратных возможностей.

Но бывают ситуации, когда пропускную способность желательно ограничить. Для ограничения степени параллелизма и количества создаваемых экземпляров

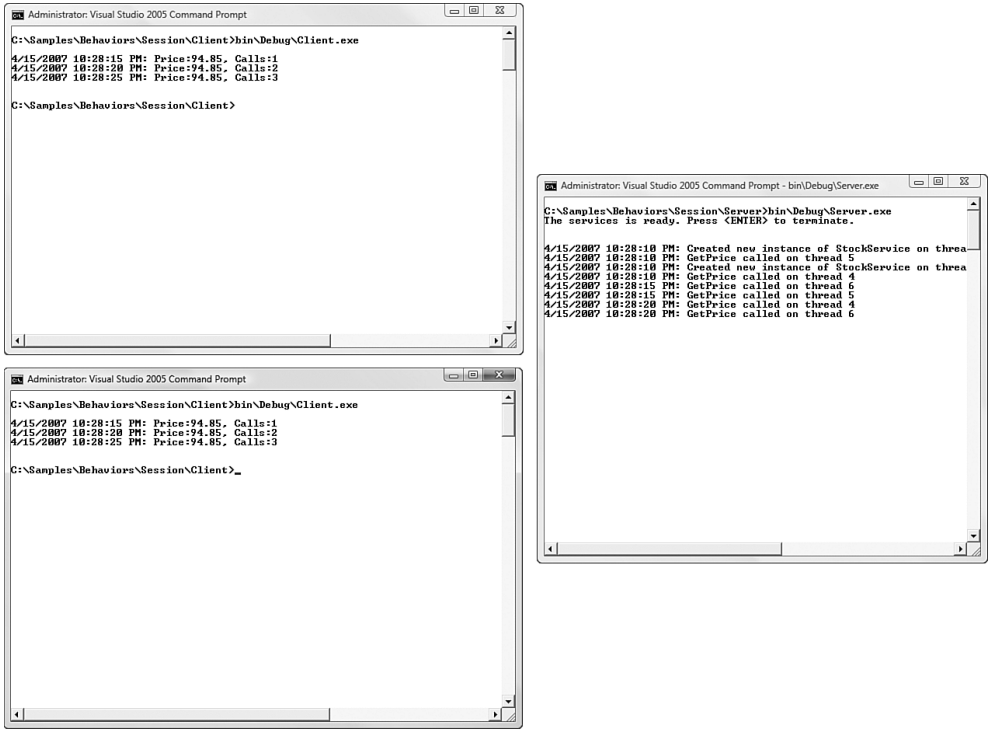

Рис. 5.5. Вывод службы, распознающей сеансы

имеются три параметра. Они задаются в элементе `serviceThrottling` в секции конфигурационного файла, относящейся к поведениям.

Поведение `maxConcurrentInstances` управляет количеством создаваемых экземпляров службы. Этот параметр полезен, если для поведения `ConcurrencyMode` задано значение `PerCall` или `PerSession`, поскольку в этих случаях экземпляры создаются по мере необходимости. Определив максимальное количество, вы ограничите число экземпляров службы, одновременно находящихся в памяти. По достижении предела новые экземпляры перестают создаваться до тех пор, пока не будет уничтожен или доступен для повторного использования один из старых.

В листинге 5.6 приведен код службы, для которой поведения `ConcurrencyMode` и `InstancingMode` явно не заданы и, следовательно, по умолчанию применяются значения `Single` и `PerSession` соответственно. Для обслуживания всех поступивших запросов этой службе требуется 20 секунд.

Листинг 5.6. Служба, в которой поведения ConcurrencyMode и InstancingMode выбраны по умолчанию

```
[ServiceContract]
public interface IStockService
{
```

```
   [OperationContract]
   double GetPrice(string ticker);
}

public class StockService : IStockService
{
   StockService()
   {
      Console.WriteLine("{0}:
               В потоке создан новый экземпляр StockService",
               System.DateTime.Now);
   }
   public double GetPrice(string ticker)
   {
      Console.WriteLine("{0}: GetPrice вызван в потоке {1}",
               System.DateTime.Now,
               Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(20000);
      return 94.85;
   }
}
```

В листинге 5.7 приведен код клиента, который десять раз асинхронно вызывает эту службу.

Листинг 5.7. Клиент, асинхронно вызывающий службу десять раз

```
class Program
{
   static int c = 0;
   static void Main(string[] args)
   {
      Console.WriteLine();
      ServiceReference.StockServiceClient p;
      for (int i = 0; i < 10; i++)
      {
         Console.WriteLine("{0}: вызывается GetPrice",
                                 System.DateTime.Now);
         p = new ServiceReference.StockServiceClient();
         p.BeginGetPrice("MSFT", GetPriceCallback, p);
         Interlocked.Increment(ref c);
      }
      while (c > 0) // дождаться получения всех ответов
      {
         Thread.Sleep(100);
      }
   }

   static void GetPriceCallback(IAsyncResult ar)
   {
      try
      {
         double price = ((ServiceReference.StockServiceClient)
                              ar.AsyncState).EndGetPrice(ar);
         Console.WriteLine("{0}: Цена:{1}",
```



```
                                    System.DateTime.Now, price);
         ((ServiceReference.StockServiceClient)ar.AsyncState).Close();
         Interlocked.Decrement(ref c);
      }
      catch (Exception ex)
      {
         Console.WriteLine(ex.InnerException.Message);
      }
   }
}
```

В листинге 5.8 показан файл `app.config` для этой службы. Параметр `maxConcurrentInstances` равен пяти, то есть служба не будет создавать более пяти экземпляров.

Листинг 5.8. Ограничение степени параллелизма с помощью параметра maxConcurrentInstances

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>

      <services>
         <service name="EssentialWCF.StockService"
                        behaviorConfiguration="throttling">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EssentialWCF"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                           binding="basicHttpBinding"
                           contract="EssentialWCF.StockService" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="throttling">
               <serviceThrottling maxConcurrentInstances="5"/>
            </behavior>
         </serviceBehaviors>
      </behaviors>

   </system.serviceModel>
</configuration>
```

На рис. 5.6 показано, что выводит клиент (слева) и служба (справа). Обратите внимание, что сразу после запуска клиент отправляет десять запросов. Спустя 20 секунд прибывают первые пять ответов, а еще через 20 секунд – пять оставшихся. Из распечатки работы службы видно, что первые пять экземпляров создаются немедленно по получении запросов от клиента, но последующие пять – только после того, как созданные ранее завершат работу.

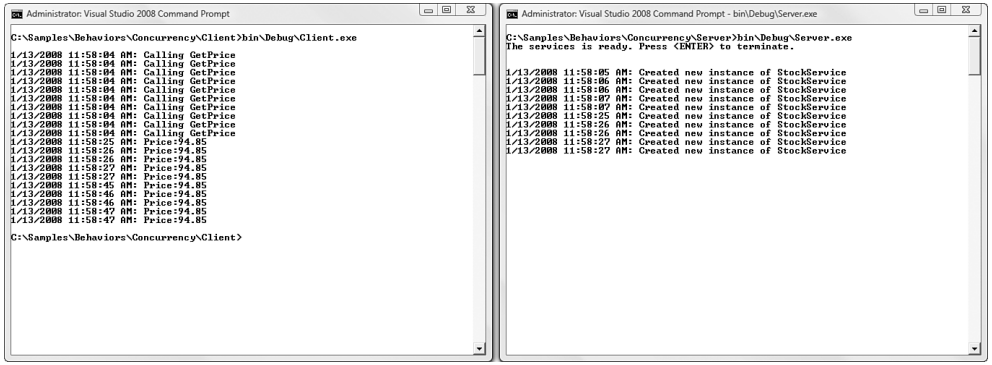

Рис. 5.6. Вывод службы и клиента в ситуации, когда ограничено число одновременно работающих экземпляров

Управление количеством одновременных вызовов

Если `InstancingMode` равно `Single`, то WCF создает единственный экземпляр службы вне зависимости от количества клиентских запросов. Если `ConcurrencyMode` равно `Multiple`, то WCF создает по одному потоку на каждый запрос (до достижения системного предела) для параллельного выполнения методов службы. Чтобы ограничить число потоков, можно воспользоваться поведением `maxConcurrentCalls`, которое задает максимальное число одновременно обслуживаемых запросов.

В листинге 5.9 приведен код службы с поведениями `InstanceContextMode.Single` и `ConcurrencyMode.Multiple`. Для завершения работы ей требуется 20 секунд.

Листинг 5.9. Служба, в которой используются поведения InstanceContextMode.Single и ConcurrencyMode.Multiple

```
[ServiceContract]
public interface IStockService
{
   [OperationContract]
   double GetPrice(string ticker);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                 ConcurrencyMode = ConcurrencyMode.Multiple)]
public class StockService : IStockService
{
   StockService()
   {
      Console.WriteLine("{0}:
               В потоке создан новый экземпляр StockService",
               System.DateTime.Now);
   }
   public double GetPrice(string ticker)
   {
      Console.WriteLine("{0}: GetPrice вызван в потоке {1}",
               System.DateTime.Now,
               Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(20000);
      return 94.85;
   }
}
```


В листинге 5.10 показан файл `app.config` для этой службы. Параметр `maxConcurrentCalls` равен пяти, следовательно, одновременно может обрабатываться не более пяти вызовов.

Листинг 5.10. Ограничение степени параллелизма с помощью параметра maxConcurrentCalls

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>

      <services>
         <service name="EssentialWCF.StockService"
                           behaviorConfiguration="throttling">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EssentialWCF"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                           binding="basicHttpBinding"
                           contract="EssentialWCF.StockService" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="throttling">
               <serviceThrottling maxConcurrentCalls="5"/>
            </behavior>
         </serviceBehaviors>
      </behaviors>

   </system.serviceModel>
</configuration>
```

На рис. 5.7 показано, что выводит клиент (слева) и служба (справа). Обратите внимание, что сразу после запуска клиент отправляет десять запросов. Спустя 20 секунд прибывают первые пять ответов, а еще через 20 секунд – пять оставшихся. Из распечатки работы службы видно, что был создан только один экземпляр, а первые пять обращений к `GetPrice` начали обрабатываться немедленно – каждый в своем потоке. По завершении обработки потоки используются повторно и обрабатываются следующие пять обращений.

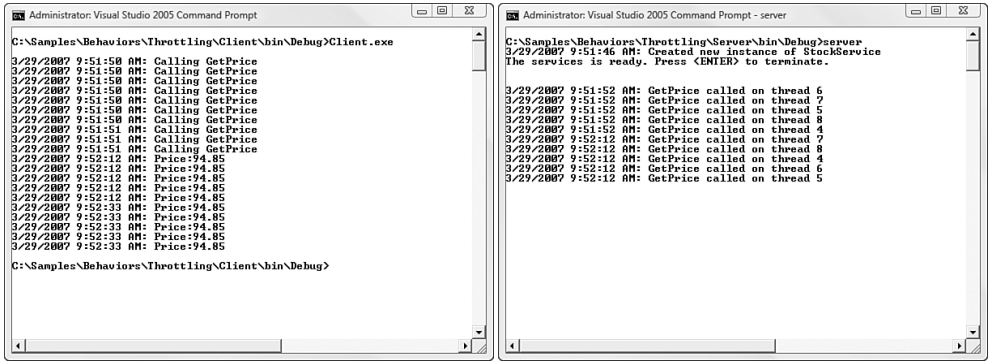

Рис. 5.7. Вывод службы и клиента в ситуации, когда ограничено число одновременно обрабатываемых вызовов

Управление количеством одновременных сеансов

Если `InstancingMode` равно `PerSession`, то WCF создает отдельный экземпляр для каждого сеанса соединения со службой. Для управления числом сеансов можно воспользоваться поведением `maxConcurrentSessions`. По достижении максимума следующий клиент, пытающийся создать сеанс, будет ждать закрытия какого-либо из уже существующих сеансов. Это полезно, когда нужно ограничить число пользователей (клиентов или серверов), соединяющихся со службой.

В листинге 5.11 приведен код службы с поведениями `InstanceContextMode.PerSession` и `ConcurrencyMode.Multiple`. Для завершения работы ей требуется 20 секунд.

Листинг 5.11. Служба, в которой используются поведения InstanceContextMode.PerSession и ConcurrencyMode.Multiple

```
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IStockService
{
   [OperationContract]
   double GetPrice(string ticker);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession,
                 ConcurrencyMode = ConcurrencyMode.Multiple)]
public class StockService : IStockService
{
   StockService()
   {
      Console.WriteLine("{0}: Создан новый экземпляр: {1}",
            System.DateTime.Now);
            OperationContext.Current.SessionId);
   }
   public double GetPrice(string ticker)
   {
```



```
      Console.WriteLine("{0}: GetPrice вызван в потоке {1}",
            System.DateTime.Now,
            Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(20000);
      return 94.85;
   }
}
```

В листинге 5.12 показан файл `app.config` для это службы. Параметр `maxConcurrentSessions` равен пяти, следовательно, одновременно может быть создано не более пяти клиентских сеансов. Отметим, что мы указали привязку `wsHttpBinding`, а не `basicHttpBinding`, так как последняя не поддерживает сеансов.

Листинг 5.12. Ограничение степени параллелизма с помощью параметра maxConcurrentSessions

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>

      <services>
         <service name="EssentialWCF.StockService"
                  behaviorConfiguration="throttling">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EssentialWCF"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                  binding="wsHttpBinding"
                  contract="EssentialWCF.StockService" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="throttling">
               <serviceThrottling maxConcurrentSessions="5"/>
            </behavior>
         </serviceBehaviors>
      </behaviors>

   </system.serviceModel>
</configuration>
```

На рис. 5.8 показано, что выводит клиент (слева) и служба (справа). Сразу после запуска клиент отправляет десять запросов. Спустя 20 секунд прибывают первые пять ответов, а еще через 20 секунд – пять оставшихся. Из распечатки работы службы видно, что создано пять сеансов, и пять обращений к `GetPrice` начали обрабатываться немедленно. Когда обработка этих пяти вызовов завершится и клиент закроет соединение, могут быть созданы следующие пять сеансов.

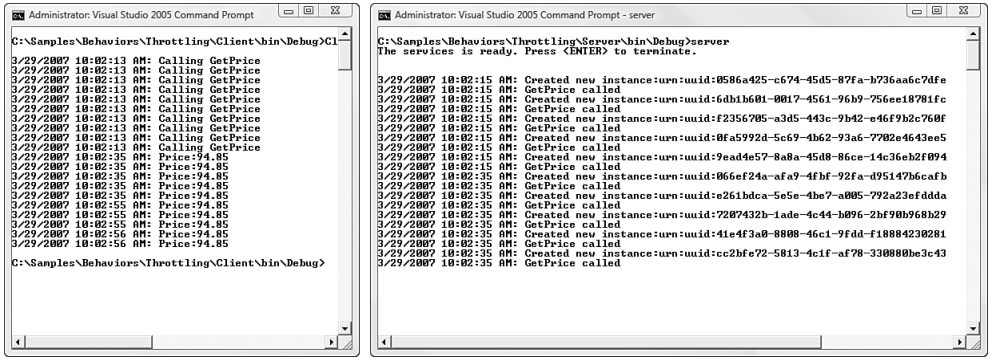

Рис. 5.8. Вывод службы и клиента в ситуации, когда ограничено число одновременно открытых сеансов

Экспорт и публикация метаданных (поведение службы)

АПК службы – ее адреса, привязки и контракты – представлены с помощью метаданных, чтобы потенциальные клиенты знали, куда, как и что отправлять. Все это вместе называется *метаданными службы*. Именно поведение службы в части метаданных – первое, с чем сталкивается большинство разработчиков, так как оно присутствует в конфигурационных файлах, которые Visual Studio 2008 генерирует при создании WCF-проекта. Для открытия доступа к метаданным это поведение работает в паре с оконечной точкой метаданных.

Чтобы сделать метаданные полезными для клиентов, необходимы два шага: экспортировать их в формате, который клиент может прочитать, и опубликовать там, где клиент сможет их найти. По умолчанию форматом экспорта является WSDL, поэтому клиент, понимающий этот формат, который описан в стандарте, будет знать, как взаимодействовать со службой. WCF публикует метаданные по протоколу WS-MetadataExchange, работающему поверх любого поддерживаемого транспорта, или возвращает их в ответ на запрос HTTP GET. Оба шага – экспорт и публикация метаданных – реализуются поведением службы `ServiceMetadataBehavior`.

Служба раскрывает свои метаданные через оконечную точку Metadata Exchange (MEX). Она, как и любая другая оконечная точка WCF, имеет адрес, привязку и контракт и может быть добавлена к службе с помощью конфигурационного файла или программно.

Оконечная точка MEX должна в своем контракте раскрывать интерфейс `IMetadataExchange`, который определен в пространстве имен `System.ServiceModel.Description` и содержит пять методов для инспекции метаданных службы и раскрытия их в WSDL-документе. Для оконечных точек MEX имеется ряд системных привязок, в частности, `mexHttpBinding`, `mexHttpsBinding`, `mexNamedPipeBinding` и `mexTcpBinding`. Адрес оконечной точки может быть как

абсолютным, так и относительным и составляется в соответствии с обычными правилами адресации оконечных точек.

В листинге 5.13 показан конфигурационный файл, в котором метаданные определяются и раскрываются с помощью поведения `serviceMetadata`. Для этого поведения задан атрибут `httpGetEnabled="True"`, который означает, что WCF должна отвечать не только на запросы по протоколу WS-MEX, но и на запросы HTTP GET.

В контракт о службе включена оконечная точка, которая раскрывает интерфейс `ImetadataExchange`. Для нее указана относительная адресация по протоколу HTTP, поэтому абсолютный адрес равен http://localhost:8000/EssentialWCF/mex. Используется привязка `mexHttpBinding`, которая создает привязку `wsHttpBinding` без обеспечения безопасности.

Листинг 5.13. Конфигурационный файл, разрешающий публикацию метаданных с помощью элемента serviceMetadata

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>

      <services>
      <service name="EssentialWCF.StockService"
                     behaviorConfiguration="myBehavior">
         <host>
            <baseAddresses>
               <add baseAddress="http://localhost:8000/EssentialWCF"/>
            </baseAddresses>
         </host>
         <endpoint address=""
                        binding="basicHttpBinding"
                        contract="EssentialWCF.StockService" />
         <endpoint address="mex"
                        binding="mexHttpBinding"
                        contract="IMetadataExchange" />
      </service>
   </services>

   <behaviors>
      <serviceBehaviors>
         <behavior name=" myBehavior ">
            <serviceMetadata httpGetEnabled="True"/>
         </behavior>
      </serviceBehaviors>
   </behaviors>

   </system.serviceModel>
</configuration>
```

В листинге 5.14 приведен код авторазмещаемой службы, которая раскрывает свои метаданные. Функционально код эквивалентен конфигурации в листинге 5.13.

Листинг 5.14. Авторазмещаемая служба, разрешающая публикацию метаданных с помощью класса ServiceMetadataBehavior

```
[ServiceContract]
public interface IStockService
{
   [OperationContract]
   double GetPrice(string ticker);
}

public class StockService : IStockService
{
   public double GetPrice(string ticker)
   {
      return 94.85;
   }
}

public class service
{
   public static void Main()
   {
      ServiceHost serviceHost = new ServiceHost(typeof(StockService),
            new Uri("http://localhost:8000/EssentialWCF"));
      serviceHost.AddServiceEndpoint(
            typeof(IStockService),
            new BasicHttpBinding(),
            "");

      ServiceMetadataBehavior behavior = new ServiceMetadataBehavior();
      behavior.HttpGetEnabled = true;
      serviceHost.Description.Behaviors.Add(behavior);
      serviceHost.AddServiceEndpoint(
            typeof(IMetadataExchange),
            MetadataExchangeBindings.CreateMexHttpBinding(),
            "mex");

      serviceHost.Open();

      // Теперь к службе можно обратиться.
      Console.WriteLine("Служба готова. Для завершения нажмите <ENTER>.\n");
      Console.ReadLine();

      serviceHost.Close();
   }
}
```

Реализация транзакций (поведение операции)

Говоря о транзакциях, обычно имеют в виду один из двух сценариев. Многошаговые бизнес-процессы занимают много времени – минуты, дни и даже месяцы. Поток работ в таких процессах может затрагивать несколько организаций и физических лиц. Короткие транзакции – это бизнес-операции, которые как правило завершаются в течение нескольких секунд и почти не имеют внешних зависимостей. Хотя для того и для другого определен четкий интерфейс и поток работ детерминирован, принципиально это совершенно разные вещи. WCF поддерживает короткие транзакции, опираясь на инфраструктуру транзакций в .NET и Windows, когда все происходит на платформе Windows, и прибегая к транзакциям, определенным в стандартах WS-*, в случае кросс-платформенных операций.

В многошаговых бизнес-процессах обычно присутствуют как автоматизированные, так и ручные операции. Некоторые выполняются очень быстро (например, размещение заказа), другие растягиваются на месяцы (например, возврат переплаты). Если многошаговый бизнес-процесс (скажем, планирование командировки) прекращается, не дойдя до конца, то все уже выполненные шаги (например, бронирование билетов на самолет) необходимо откатить, выполнив компенсаторные шаги (отказ от брони). Такие транзакции лучше всего поддерживать с помощью брокера сообщений или такой «корпоративной шины данных», как BizTalk Server.

Короткие транзакции инкапсулируют дискретные бизнес-функции. Обычно они завершаются за несколько секунд. Бизнес-функции могут быть высокоуровневыми (например, «открыть новый счет»), цель которых – агрегирование или обновление информации из нескольких источников. Но бывают и низкоуровневые бизнес-функции («изменить адрес заказчика»), которые обновляют только один источник. В любом случае обновления, включенные в транзакцию, должны выполняться атомарно – либо все, либо ни одного, – чтобы не пострадала целостность данных. Если в ходе какого-то обновления возникает ошибка, то служба должна откатить все уже совершенные изменения, оставив данные точно в том виде, в котором они были перед началом транзакции.

Обычно такое поведение характеризуют как ACID-транзакцию. На эту тему существует обширная литература, но, если коротко, то ACID-транзакции обладают следующими свойствами:

- **Атомарность (Atomic)**. Либо все обновления, включенные в транзакцию, выполнены успешно, либо вообще не выполнены. Частичные обновления не допускаются. Например, если при переводе денег с одного банковского счета на другой дебетование выполнено успешно, а кредитование завершилось с ошибкой, то операция дебетования откатывается, чтобы деньги не появлялись из ничего и не уходили в никуда.
- **Непротиворечивость (Consistent)**. После завершения операции все данные оказываются корректны с точки зрения бизнес-правил. Например, в случае перевода денег исходный и конечный счета должны быть допустимы, иначе транзакция отменяется.
- **Изолированность (Isolated)**. Во время выполнения операции промежуточные результаты обновления не видны вне самой транзакции. Например, при переводе денег никакие другие клиенты не должны видеть частично обновленных балансов.
- **Долговечность (Durable)**. После того как транзакция зафиксирована, данные должны быть сохранены так, чтобы не терялись при последующих сбоях в системе.

Транзакционные операции внутри службы

Все операции транзакционных служб завершаются успешно или неудачно как неделимое целое. Инициировавшая их сторона вправе предполагать, что результат будет непротиворечив, как бы ни завершилась операция. На рис. 5.9 изображен псевдокод такого поведения. Клиент открывает соединение со службой и вызывает ее метод `Transfer`, который выполняет дебетование одного счета, кредитование другого и помечает, что транзакция завершилась. Для обеспечения семантики транзакции сам клиент ничего не делает.

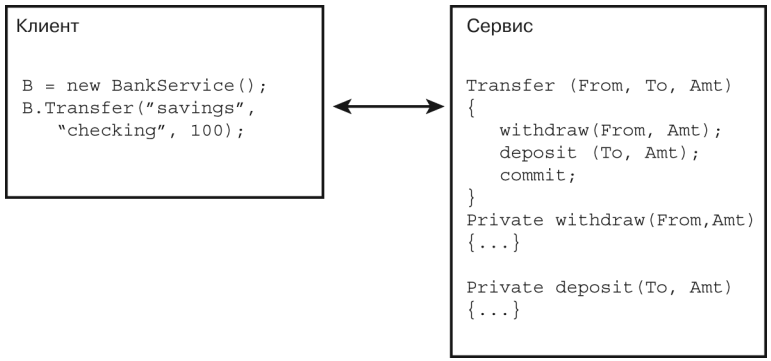

Рис. 5.9. Операция, оформленная в виде ACID-транзакции

Для реализации такого поведения в WCF операция службы должна быть помечена как транзакционная с помощью атрибута `[OperationBehavior (TransactionScopeRequired=true)]`. В результате перед тем, как передать управление вызванному методу, WCF создаст новую транзакцию и включит в нее поток, в котором выполняется операция. Если в ходе выполнения операции возникнет ошибка, все частичные обновления транзакционных ресурсов будут откачены.

Если в атрибуте задано значение `TransactionScopeRequired=false` (это умолчание), то операция выполняется без заведения транзакции и, следовательно, не поддерживает свойства ACID. Если такая операция обновит одну таблицу, а при обновлении другой возникнет ошибка, то первая таблица останется обновленной, то есть свойство атомарности нарушается.

Вы можете уведомить о завершении операции явно или неявно. Если задано поведение `[OperationBehavior(TransactionAutoComplete=true)]`, то операция неявно считается завершенной, если она не сообщила об ошибке. Если же произошла ошибка, то операция считается незавершенной и частичные обновления откатываются. Вместо этого можно указать поведение `[OperationBehavior (TransactionAutoComplete =false)]` и перед возвратом управления явно вызвать метод `OperationContext.Current.SetTransactionComplete()`. В последнем случае необходимо использовать в коммуникационном канале сеансовый

элемент привязки и в контракте о службе сообщить, что она поддерживает сеансы, с помощью атрибута `[ServiceContract(SessionMode=SessionMode.Allowed)]`.

В листинге 5.15 показан код службы `BankService`, которая раскрывает две операции. Первая – `GetBalance` – не является транзакционной. Она просто читает баланс счета из базы данных и возвращает его в качестве результата. Чтобы обозначить ненужность транзакции, задан атрибут `OperationBehavior [TransactionScopeRequired=false]`. Вторая операция – `Transfer` – транзакционная, поэтому для нее задано поведение `TransactionScopeRequired=true`. Она вызывает два внутренних метода `Withdraw` и `Deposit`, каждый из которых обновляет базу данных с помощью класса `DBAccess`. Операция `Transfer` неявно помечает транзакцию как завершенную, поскольку для нее задан атрибут `TransactionAutoComplete=true`. Если ни одна из операций `Withdraw` и `Deposit` не возбуждает исключения, то считается, что произведенные ими изменения выполнены полностью.

В состав службы `BankService` входит также внутренний класс `DBAccess`, который выполняет все операции доступа к базе данных. Обратите внимание, что в его конструкторе открывается соединение с базой данных. Когда объект `DBAccess` покинет область видимости и при этом не останется невыполненных запросов или активных транзакций, сборщик мусора закроет соединение. Попытка принудительно закрыть соединение в деструкторе приведет к ошибке, поскольку в момент, когда объект выходит из области видимости, еще могут оставаться активные транзакции.

Листинг 5.15. Транзакционная операция

```
[ServiceContract]
public interface IBankService
{
   [OperationContract]
   double GetBalance(string AccountName);

   [OperationContract]
   void Transfer(string From, string To, double amount);
}

public class BankService : IBankService
{
   [OperationBehavior (TransactionScopeRequired=false)]
   public double GetBalance (string AccountName)
   {
      DBAccess dbAccess = new DBAccess();
      double amount = dbAccess.GetBalance(AccountName);
      dbAccess.Audit(AccountName, "Query", amount);
      return amount;
   }

   [OperationBehavior(TransactionScopeRequired = true,
                                    TransactionAutoComplete = true)]
   public void Transfer(string From, string To, double amount)
```

```
   {
      try
      {
         Withdraw(From, amount);
         Deposit(To, amount);
      }
      catch (Exception ex)
      {
         throw ex;
      }
   }

   private void Withdraw(string AccountName, double amount)
   {
      DBAccess dbAccess = new DBAccess();
      dbAccess.Withdraw(AccountName, amount);
      dbAccess.Audit(AccountName, "Withdraw", amount);
   }
   private void Deposit(string AccountName, double amount)
   {
      DBAccess dbAccess = new DBAccess();
      dbAccess.Deposit(AccountName, amount);
      dbAccess.Audit(AccountName, "Deposit", amount);
   }
}

class DBAccess
{
   private SqlConnection conn;
   public DBAccess()
   {
      string cs = ConfigurationManager.
                     ConnectionStrings["sampleDB"].ConnectionString;
      conn = new SqlConnection(cs);
      conn.Open();
   }

   public void Deposit(string AccountName, double amount)
   {
      string sql = string.Format("Deposit {0}, {1}, "{2}"",
                                 AccountName, amount.ToString(),
                                 System.DateTime.Now.ToString());
      SqlCommand cmd = new SqlCommand(sql, conn);
      cmd.ExecuteNonQuery();
   }
   public void Withdraw(string AccountName, double amount)
   {
      string sql = string.Format("Withdraw {0}, {1}, "{2}"",
                                 AccountName, amount.ToString(),
                                 System.DateTime.Now.ToString());
      SqlCommand cmd = new SqlCommand(sql, conn);
      cmd.ExecuteNonQuery();
   }
   public double GetBalance(string AccountName)
   {
```



```
      SqlCommand cmd = new SqlCommand("GetBalance " +
                                      AccountName, conn);
      SqlDataReader reader = cmd.ExecuteReader();
      reader.Read();
      double amount = System.Convert.ToDouble(reader["Balance"].ToString());
      reader.Close();
      return amount;
   }
   public void Audit(string AccountName, string Action, double amount)
   {
      Transaction txn = Transaction.Current;
      if (txn != null)
         Console.WriteLine("{0} | {1} Audit:{2}",
            txn.TransactionInformation.DistributedIdentifier,
            txn.TransactionInformation.LocalIdentifier,Action);
      else
         Console.WriteLine("<no transaction> Audit:{0}",
                                      Action);
         string sql = string.Format("Audit {0}, {1}, {2}, "{3}"" ,
                                      AccountName, Action, amount.ToString(),
                                      System.DateTime.Now.ToString());
         SqlCommand cmd = new SqlCommand(sql, conn);
         cmd.ExecuteNonQuery();
   }

}
```

Код клиента для этого примера показан в листинге 5.16. Клиент ничего не знает о транзакции внутри службы.

Листинг 5.16. Клиент, вызывающий транзакционную службу

```
ServiceReference.BankServiceClient proxy = new
              ServiceReference.BankServiceClient();

Console.WriteLine("{0}: До — сберегательный счет:{1}, счет до востребования {2}",
                           System.DateTime.Now,
                           proxy.GetBalance("savings"),
                           proxy.GetBalance("checking"));

proxy.Transfer("savings", "checking", 100);

Console.WriteLine("{0}: После — сберегательный счет:{1}, счет до востребо-
вания {2}",
                           System.DateTime.Now,
                           proxy.GetBalance("savings"),
                           proxy.GetBalance("checking"));
proxy.Close();
```

Поскольку каждый из двух внутренних методов `Withdraw` и `Deposit` создает новый объект класса `DBAccess`, то открываются два независимых соединения с базой данных. Когда `Withdraw` открывает первое соединение внутри транзакции, эта транзакция оказывается локальной, а не распределенной. Но в момент открытия второго соединения транзакция преобразуется в распределенную, что-

бы можно было координировать действия, выполняемые в обоих соединениях. Метод `DBAccess.Audit` распечатывает идентификаторы локальной (`LocalIdentifier`) и распределенной (`DistributedIdentier`) транзакции, как показано на рис. 5.10. Обратите внимание, что метод `Withdraw` выполняется без распределенной транзакции, поскольку в этот момент открыто только одно соединение с базой данных. Но при выполнении `Deposit` транзакция уже стала распределенной, так как в области ее действия появилось второе соединение. Эскалация производится автоматически, и это может иметь резко негативные последствия для производительности.

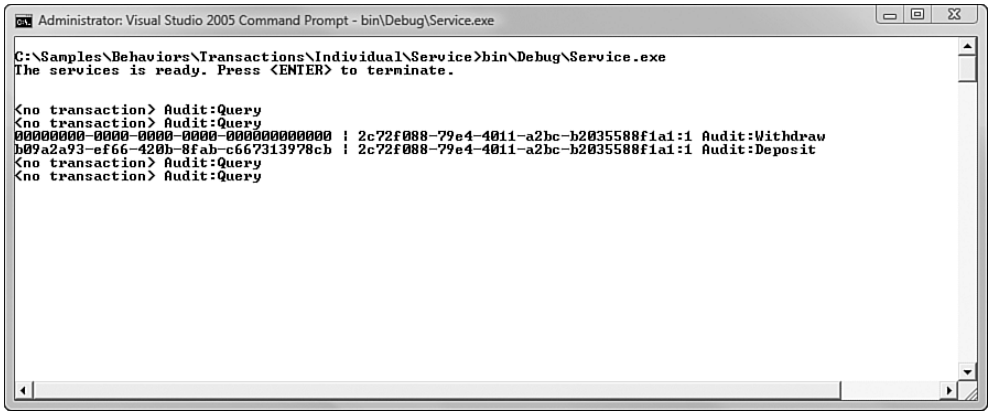

Рис. 5.10. Распечатка, в которой показаны идентификаторы локальной и распределенной транзакций

В листинге 5.17 приведен оптимизированный код, в котором операция перевода денег сама создает объект `DBAccess`, открывая при этом соединение, а затем передает этот объект методам `Withdraw` и `Deposit`, так что в результате мы имеем всего одно соединение.

Листинг 5.17. Транзакционная операция, оптимизированная, чтобы избежать появления распределенной транзакции

```
[OperationBehavior(TransactionScopeRequired = true,
                   TransactionAutoComplete = true)]
private void Transfer(string From, string To, double amount)
{
   DBAccess dbAccess = new DBAccess();
   Withdraw(From, amount, dbAccess);
   Deposit(To, amount, dbAccess);
}

private void Withdraw(string AccountName, double amount,
                      DBAccess dbAccess)
{
   dbAccess.Withdraw(AccountName, amount);
   dbAccess.Audit(AccountName, "Withdraw", amount);
}
private void Deposit(string AccountName, double amount,
                     DBAccess dbAccess)
{
   dbAccess.Deposit(AccountName, amount);
   dbAccess.Audit(AccountName, "Deposit", amount);
}
```


На рис. 5.11 показано, что выводит оптимизированная служба. Теперь идентификатор распределенной транзакции остается равным 0, то есть распределенная транзакция не создавалась.

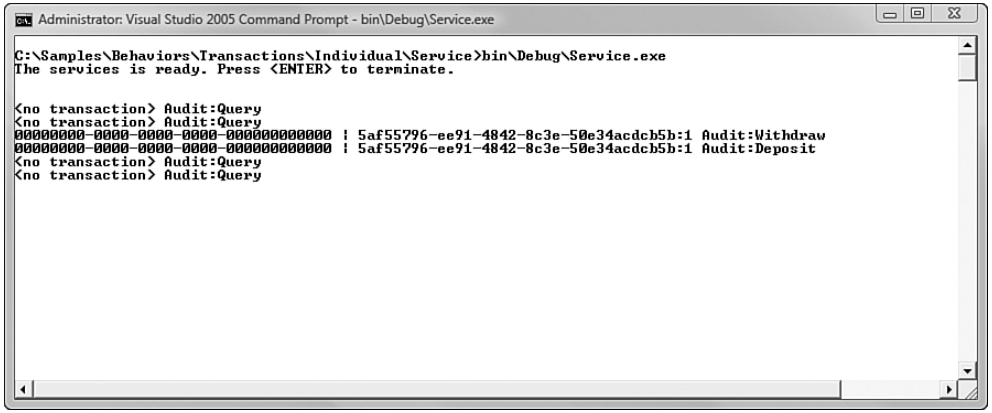

Рис. 5.11. Распечатка работы оптимизированной транзакционной службы

Поток транзакций, пересекающий границы операций

В распределенных системах транзакции иногда пересекают границы служб. Например, если одна служба управляет информацией о заказчиках, другая – информацией о заказах, а пользователь хочет разместить заказ и отгрузить товар по новому адресу, то система вынуждена будет вызывать операции обеих служб. Коль скоро транзакция успешно завершилась, пользователь вправе ожидать, что обе части были корректно обновлены. Если инфраструктура поддерживает атомарный транзакционный протокол, то службы можно объединять в подобную агрегированную транзакцию. Спецификация WS-AT (Web Service Atomic Transactions) описывает инфраструктуру разделения информации между участвующими службами, которая необходима для реализации семантики двухфазной фиксации транзакций. В WCF переход транзакции через границы служб называется *потоком транзакций*. Чтобы обеспечить семантику потока транзакций, необходимо выполнить пять шагов:

- **(Контракт о службе)** `SessionMode.Required`. В контракте о службе необходимо затребовать сеансы, поскольку именно таким способом координ-

натор (обычно клиент) и службы-участницы получают общий доступ к инормации.

- **(Поведение операции)** `TransactionScopeRequired=true`. В поведении операции необходимо затребовать транзакционный контекст. Это означает, что транзакция будет создана, если ее еще не существует.
- **(Контракт об операции)** `TransactionFlowOption.Allowed`. В контрактах об операциях нужно разрешить включение информации о потоке транзакций в заголовок сообщений.
- **(Определение привязки)** `TransactionFlow=true`. В привязке должен быть разрешен поток транзакций, чтобы канал мог помещать информацию о транзакциях в заголовок SOAP. Отметим также, что привязка должна поддерживать сеансы; это умеет делать `wsHttpBinding`, но не `basicHttpBinding`.
- **(Клиент)** `TransactionScope`. Сторона, инициировавшая транзакцию, обычно клиент, должна обращаться к операциям службы в транзакционном контексте. Кроме того, для фиксации изменений она должна вызывать метод `TransactionScope.Close()`.

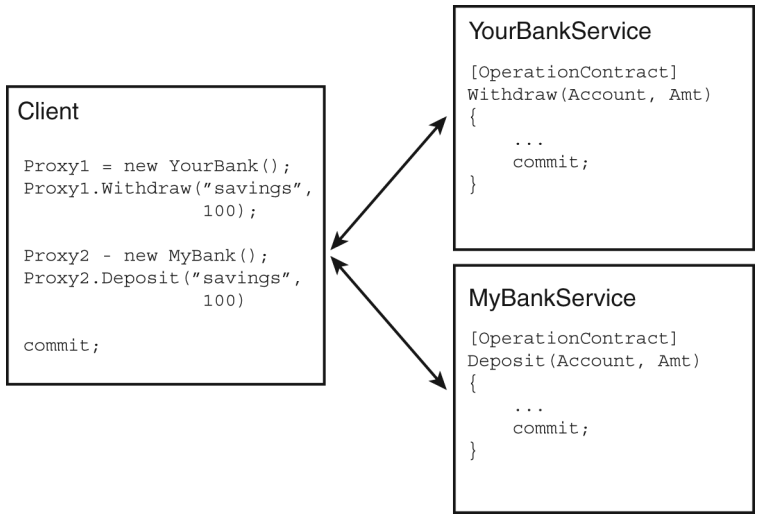

Рис. 5.12. Транзакция, пересекающая границы служб

В документации по .NET 3.5, относящейся к атрибуту `TransactionScopeRequired`, приведена таблица 5.2, в которой описываются отношения между этими элементами. Ниже она воспроизводится для удобства читателя.

В листинге 5.18 демонстрируется использование этих элементов. Код аналогичен представленному в листинге 5.15, но, если раньше гарантировалась транзакционная целостность только одной операции службы (`Transfer`), то теперь наличие атрибута `TransactionFlowOption` гарантирует то же самое сразу для

Таблица 5.2

TransactionScopeRequired	Привязка допускает поток транзакций	Вызывающая сторона обеспечивает поток транзакций	Результат
False	False	Нет	Метод выполняется вне транзакции
True	False	Нет	Метод создает новую транзакцию и выполняется в ее контексте
True или False	False	Да	В заголовке SOAP-сообщения возвращается информация об ошибке
False	True	Да	Метод выполняется вне транзакции
True	True	Да	Метод выполняется в потоке транзакций

нескольких служб. Отметим несколько моментов. Во-первых, в контракте `ServiceContract` помечено, что он требует открытия сеансов. Чтобы удовлетворить этому требованию, мы должны использовать сеансовый протокол, например, `wsHttpBinding` или `netTcpBinding`. Во-вторых, для иллюстрации мы установили поведение `TransactionAutoComplete` в false, а в последней строчке метода вызываем `SetTransactionComplete`. Если в ходе выполнения мы не дойдем до вызова `SetTransactionComplete`, транзакция будет автоматически откачена. В-третьих, для каждого контракта об операции задан атрибут `TransactionFlowOption.Allowed`, позволяющий транзакции пересекать границы вызовов.

Листинг 5.18. Транзакция, пересекающая границы операций

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IBankService
{
   [OperationContract]
   double GetBalance(string AccountName);

   [OperationContract]
   void Transfer(string From, string To, double amount);
}

public class BankService : IBankService
{
```

```
[OperationBehavior(TransactionScopeRequired = false)]
public double GetBalance(string AccountName)
{
      DBAccess dbAccess = new DBAccess();
      double amount = dbAccess.GetBalance(AccountName);
      dbAccess.Audit(AccountName, "Query", amount);
      return amount;
}

[OperationBehavior(TransactionScopeRequired = true,
                                TransactionAutoComplete = true)]
public void Transfer(string From, string To, double amount)
   {
      try
         {
            Withdraw(From, amount);
            Deposit(To, amount);
         }
         catch (Exception ex)
         {
            throw ex;
         }
   }

[OperationBehavior(TransactionScopeRequired = true,
                                TransactionAutoComplete = false)]
[TransactionFlow(TransactionFlowOption.Allowed)]
private void Deposit(string AccountName, double amount)
   {
      DBAccess dbAccess = new DBAccess();
      dbAccess.Deposit(AccountName, amount);
      dbAccess.Audit(AccountName, "Deposit", amount);
      OperationContext.Current.SetTransactionComplete();
   }

[OperationBehavior(TransactionScopeRequired = true,
                                TransactionAutoComplete = false)]
[TransactionFlow(TransactionFlowOption.Allowed)]
private void Withdraw(string AccountName, double amount)
   {
      DBAccess dbAccess = new DBAccess();
      dbAccess.Withdraw(AccountName, amount);
      dbAccess.Audit(AccountName, "Withdraw", amount);
      OperationContext.Current.SetTransactionComplete();
   }
}
```

В листинге 5.19 приведен конфигурационный файл. Отметим использование привязки `wsHttpBinding`, поддерживающей сеансы. Это необходимо, потому что в коде для контракта о службе задано поведение `SessionMode.Required`. Также обратите внимание на атрибут `transactionFlow="true"` в секции конфигурации привязок.

Листинг 5.19. Разрешение потока транзакций в конфигурационном файле

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```



```
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.BankService">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EssentialWCF"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                        binding="wsHttpBinding"
                        bindingConfiguration="transactions"
                        contract="EssentialWCF.IBankService" />
         </service>
      </services>

      <bindings>
         <wsHttpBinding>
            <binding name="transactions"
                  transactionFlow="true" >
            </binding>
         </wsHttpBinding>
      </bindings>
   </system.serviceModel>
</configuration>
```

В листинге 5.20 показан код клиента, который агрегирует две службы в единую транзакцию. Создаются три прокси, два из которых указывают на одну службу, а третий – на другую. Для `proxy1` вызываются две операции `Query` и одна операция `Withdraw`, после чего вызывается операция `Deposit` для `proxy2`. Если внутри этих операций не возникнет ошибок, то каждая из них вызовет `SetTransactionComplete()`. Когда обе операции вернут управление, клиент вызовет `scope.Complete()`, чтобы завершить транзакцию. Только после того, как все участники транзакции вызовут метод `SetTransactionComplete()`, транзакция будет зафиксирована, в противном случае откачена. И наконец вызываются еще две операции `Query` для `proxy3`, дабы удостовериться, что изменения пережили транзакцию.

Листинг 5.20. Координация распределенной транзакции со стороны клиента

```
using (TransactionScope scope = new
               TransactionScope(TransactionScopeOption.RequiresNew))
{
   localhost1.BankServiceClient proxy1 = new
localhost1.BankServiceClient();
   localhost2.BankServiceClient proxy2 = new
localhost2.BankServiceClient();
   Console.WriteLine("{0}: До — сберегательный счет:{1}, счет до востребования {2}",
                              System.DateTime.Now,
                              proxy1.GetBalance("savings"),
                              proxy2.GetBalance("checking"));

   proxy1.Withdraw("savings", 100);
```

```
   proxy2.Deposit("checking", 100);
   scope.Complete();

   proxy1.Close();
   proxy2.Close();
}
localhost1.BankServiceClient proxy3 = new localhost1.BankServiceClient();
Console.WriteLine("{0}: До — сберегательный счет:{1}, счет до востребования {2}",
                  System.DateTime.Now,
                  Proxy3.GetBalance("savings"),
                  Proxy3.GetBalance("checking"));
```

На рис. 5.13 показан вывод клиента и обеих служб. Информация, печатаемая клиентом, расположена слева; это балансы сберегательного счета и счета до востребования до и после перевода денег. Информация, печатаемая службами, расположена справа. К верхней службе обращаются Proxy1 и Proxy3, к нижней – Proxy2. Верхняя служба выполняет две операции `Query`, операцию `Withdraw` и еще две операции `Query`. Нижняя служба выполняет одну операцию `Deposit`. Отметим, что обе службы печатают один и тот же идентификатор распределенной транзакции, то есть входят в одну и ту же транзакцию.

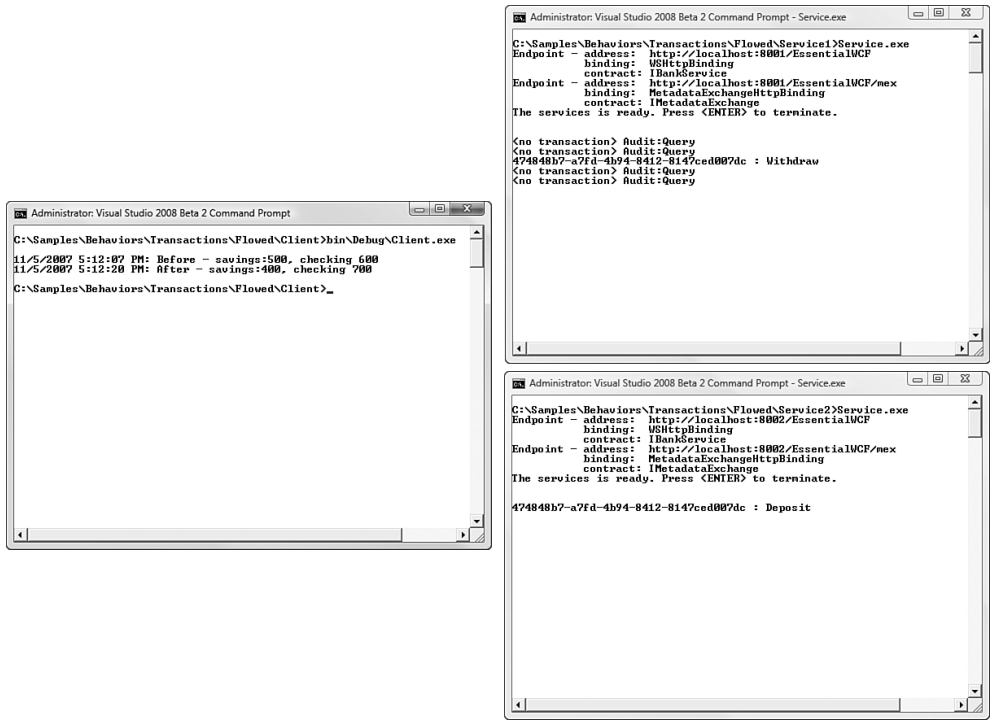

Рис. 5.13. Вывод двух транзакционных служб, скоординированных для работы в составе одной транзакции

Выбор транзакционного протокола — OleTx или WS-AT

Менеджеры транзакций отвечают за координацию работы нескольких сторон и фиксацию ее результатов с помощью *менеджеров ресурсов*. Менеджеры ресурсов отвечают за надежное постоянное хранение. В зависимости от того, какие используются ресурсы и границы каких приложений пересекаются, WCF задействует для реализации транзакций один из трех менеджеров транзакций. Облегченный менеджер транзакций (Lightweight Transaction Manager – LTM) служит для управления ресурсами в одном домене приложения. Ядерный менеджер транзакций (Kernel Transaction Manager – KTM), имеющийся только в ОС Vista и Windows Server 2008, применяется для управления транзакционной файловой системой и транзакционным реестром. Координатор распределенных транзакций (Distributed Transaction Coordinator – DTC) используется для управления транзакциями, которые пересекают границы доменов приложений, процессов и компьютеров. Приложение не контролирует выбор менеджера транзакций; WCF самостоятельно выберет оптимальный и, если необходимо, выполнит эскалацию.

Для передачи семантики перехода через границы домена приложения, процесса или компьютера можно использовать один из двух протоколов. OleTx – это двоичный протокол, применяемый только на платформе Windows. Он является «родным» для DTC и идеален для коммуникаций во внутренней сети. Протокол Web Services Atomic Transactions, или WS-AT, основан на стандартах и тоже позволяет пересекать границы процессов или компьютеров. Но, в отличие от OleTx, протокол WS-AT не зависит от транспортного механизма и может быть реализован поверх TCP, HTTP и других сетевых протоколов. Хотя приложение не участвует в выборе менеджера ресурсов, оно может определить транзакционный протокол.

На самом деле, транзакционный протокол можно задавать только для некоторых привязок: поддерживающих сеансы (это необходимо для транзакций), двусторонний обмен (необходимо для потока транзакций) и не привязанных к стеку WS-* (иначе будет гарантированно использован протокол WS-AT). Таким образом, остаются только привязки `netTcpBinding` и `netNamedPipeBinding`. Для них транзакционный протокол можно задать в коде или в конфигурационном файле. В листинге 5.21 показан конфигурационный файл, в котором используется привязка к TCP в сочетании с протоколом WS-AT; это дает возможность поддержать поток транзакций между стандартными (WS-AT), быстрыми (двоичными) и безопасными (TCP) службами через Интернет.

Листинг 5.21. Задание протокола WS-AT в привязке

```
<bindings>
   <netTcpBinding>
      <binding name = "wsat"
                  transactionFlow = "true"
                  transactionProtocol = "WSAtomicTransactionOctober2004"
      />
   </netTcpBinding>
</bindings>
```

Поведения транзакционных служб

Два поведения, определенных на уровне операций – `TransactionScopeRequired` и `TransactionAutoComplete`, – описаны выше. На уровне службы есть еще два поведения: `TransactionIsolationLevel` и `TransactionTimeout`.

Атрибут `TransactionIsolationLevel` относится к уровню изолированности транзакции, то есть к букве I в аббревиатуре ACID. Он определяет, в какой мере транзакция изолирована от среды, в которой выполняется. Существует несколько уровней изолированности. Уровень *`Serializable`*, принимаемый по умолчанию, обеспечивает наивысшую изолированность и запрещает другим процессам обновлять данные, пока транзакция не завершится. Например, если транзакция содержит предложение `select count(*)`, то, пока количество строк не будет подсчитано, ни один процесс не сможет ни вставить, ни удалить данные. Уровень *`ReadUncommitted`* – самый низкий, он позволяет другим процессам читать и изменять данные, измененные в ходе выполнения транзакции, еще до того, как она завершилась. На практике лучше оставить принимаемое по умолчанию значение *`Serializable`* и стараться избегать транзакций, блокирующих больше данных, чем необходимо, например предложений типа `select (*) from order`.

Уровень изолированности транзакций `TransactionIsolationLevel` должен быть совместим с контекстом `TransactionScope`, который определил клиент. Если ни то, ни другое не задано, по умолчанию принимается уровень `IsolationLevel.Serializable`. В листинге 5.22 показаны настройки на стороне клиента и службы; в обоих случаях задан уровень `ReadUncommitted`.

Время, отведенное для завершения транзакции, тоже можно контролировать. Задавать его можно на стороне клиента или службы, но для разных целей. На стороне клиента этот параметр ограничивает время работы транзакционной операции, инициированной пользователем. На стороне сервера ограничение может задать системный администратор, чтобы никакая транзакция не потребляла слишком много ресурсов.

В листинге 5.22 для атрибута `TransactionScopeOption.Timeout` на стороне клиента задано значение 30 секунд, то есть инициированная пользователем транзакция будет прервана, если продлится дольше. Кроме того, в листинге 5.22 в атрибуте `ServiceBehavior` задано значение `TransactionTimeout`, равное 1 минуте; любая транзакция, продолжающаяся более 1 минуты, будет автоматически прервана.

Листинг 5.22. Задание уровня изолированности и таймаута транзакций

```
//
// Код клиента
//
TransactionOptions opt = new TransactionOptions();
opt.IsolationLevel = IsolationLevel.ReadUncommitted;
opt.Timeout = new System.TimeSpan(0, 0, 30);
using (TransactionScope scope = new
        TransactionScope (TransactionScopeOption.RequiresNew), opt))
{
   localhost.BankServiceClient proxy = new localhost.BankServiceClient();
   proxy.Transfer("savings", "checking", 100);
   scope.Complete();
   proxy.Close();
}

//
// Код службы
//
[ServiceContract(SessionMode=SessionMode.Required)]
[ServiceBehavior (TransactionIsolationLevel =
                      IsolationLevel.ReadUncommitted,
                  TransactionTimeout="00:01:00"))]
public class BankService
{
   [OperationContract]
   [OperationBehavior(TransactionScopeRequired = true,
                              TransactionAutoComplete = true)]
   [TransactionFlow(TransactionFlowOption.Allowed)]
   private void Transfer(string From, string To, double amount)
   {
      ...
   }
}
```


Реализация заказных поведений

Заказные поведения позволяют вставлять код в стратегически важные места на этапе инициализации исполняющей среды, а также в конвейер обработки сообщений. Поведения можно добавлять в код, манипулируя описанием службы с помощью атрибутов, или в конфигурационный файл. Во всех случаях программа может предпринять вспомогательные действия, например, поискать информацию в каталоге или вывести что-то в протокол для аудита.

На рис. 5.14 изображены интерфейсы, предлагаемые для создания заказных поведений на стороне клиента.

На рис. 5.15 изображены интерфейсы, предлагаемые для создания и вставки заказных поведений на стороне сервера.

Реализация заказного поведения состоит из трех шагов.

Шаг 1. Создать класс, который реализует интерфейс `Inspector`, `Selector`, `Formatter` или `Invoker`. Обычно именно в этом классе и реализуется содержательная часть поведения. Например, для протоколирования всех входящих сообщений (это делается автоматически, если воспользоваться классами из пространства имен `System.Diagnostics`, как описано в главе 9) можно реализовать интерфейс `IDispatchMessageInspector` и поместить нужный код в метод `AfterReceiveRequest`. А если вы хотите, чтобы некий код выполнялся непосредственно до и сразу после вызова операции, то реализуйте интерфейс `IOperationInvoker` и поместите код в метод `Invoke`. Однако, если ваша цель – сделать что-то с исполняющей средой (скажем, проверить или изменить привязки стека каналов), а не воздействовать на конвейер обработки сообщений (инспекти-

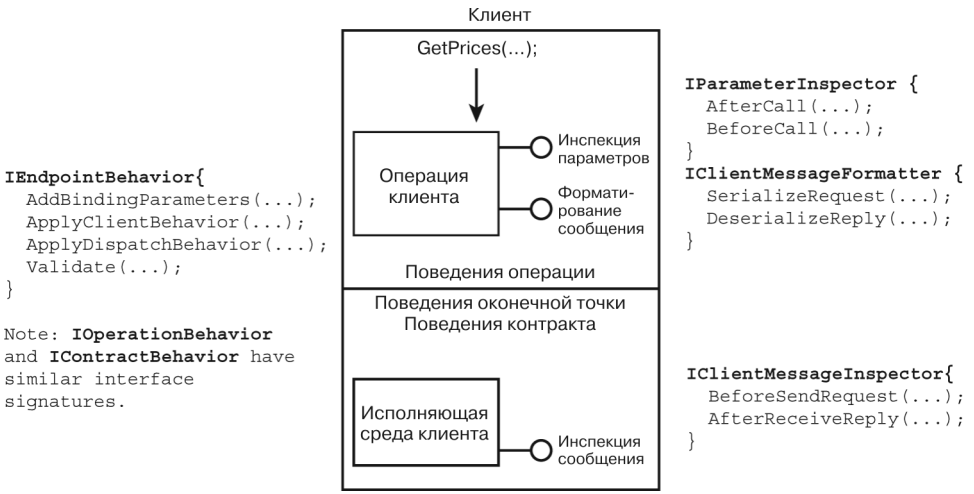

Рис. 5.14. Интерфейсы для создания заказных поведений на стороне клиента

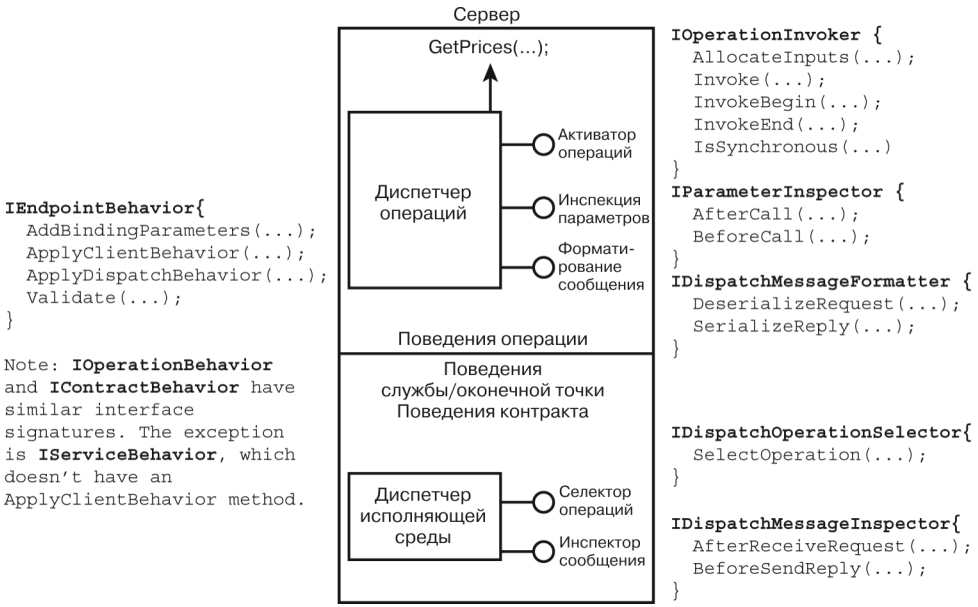

Рис. 5.15. Интерфейсы для создания заказных поведений на стороне сервера

ровать сообщения или параметры либо выбирать или вызывать операцию), то этот шаг можно пропустить.

Шаг 2. Создать класс, который реализует один из интерфейсов поведения: `IServiceBehavior`, `IEndpointBehavior`, `IOperationBehavior` или `IContract-Behavior`. С помощью метода `ApplyClientBehavior` или `ApplyDispatch-Behavior` добавьте класс, созданный на предыдущем шаге, в список поведений. Опять же, если вы намереваетесь оперировать исполняющей средой, а не конвейером обработки сообщений, можете просто вставить свой код в методы `Validate` или `AddBindingParameter` этого класса, а не регистрировать поведение клиента или диспетчера для последующего выполнения.

Шаг 3. Сконфигурировать клиент или службу, так чтобы они использовали созданное поведение. Это можно сделать в коде, в конфигурационном файле или с помощью атрибутов:

- добавить поведение за счет манипуляций с классом `ServiceDescription`. Если вы пользуетесь авторазмещением, то класс поведения можно включить в список поведений службы (если добавляется `IServiceBehavior`) или в список поведений оконечной точки либо контракта (если добавляется `IEndpointBehavior`, `IContractBehavior` или `IOperationBehavior`). Это наиболее автономный, но и самый неудобный для сопровождения механизм, поскольку при любом изменении поведения код службы придется перекомпилировать;
- воспользоваться атрибутами для добавления поведения в исполняющую среду клиента или сервера, для чего требуется реализовать интерфейс `Attribute`. Это позволяет разработчикам применять атрибуты при определении служб, оконечных точек или операций в своем коде;
- добавить поведение в исполняющую среду клиента или сервера с помощью конфигурационного файла. Для этого потребуется выполнить два дополнительных шага. Во-первых, нужно создать класс, который реализует интерфейс `BehaviorExtensionElement`, и определить в этом классе конфигурационные элементы с атрибутом `[ConfigurationProperty]`. Следует реализовать методы `CreateBehavior` и `BehaviorType` этого интерфейса, которые создают и возвращают ваш новый класс `BehaviorExtension`. Далее в тот конфигурационный файл на стороне клиента или службы, где вы собираетесь использовать поведение, нужно добавить секцию `<behaviorExtensions>`, в которую поместить ссылку на полностью квалифицированный тип, и затем уже применять новое поведение на уровне службы или оконечной точки.

После того как все три шага будут выполнены, поведение готово к использованию. Оно будет вызываться автоматически на этапе инициализации исполняющей среды клиентом или службой, а также при отправке или получении каждого сообщения. В оставшейся части этого раздела демонстрируются различные поведения.

Реализация инспектора сообщений для поведения оконечной точки

В листинге 5.23 реализовано поведение протоколирования, которое распечатывает каждое сообщение, получаемое и отправляемое оконечной точкой. Это

инспектор сообщений, вызываемый из поведения оконечной точки. Здесь же показано, как поведение добавляется в описание авторазмещаемой службы.

Реализация заказного поведения для трассировки. Если вам нужен инспектор сообщений для диагностических целей, то лучше воспользоваться приемами трассировки, описанными в главе 10 «Обработка исключений».

Класс `myMessageInspector` реализует интерфейс `IDispatchMessageInspector`. В методах `BeforeSendRequest` и `AfterReceiveReply` он выводит сообщение на консоль. Класс `myEndPointBehavior` реализует интерфейс `IEndpointBehavior`. В методе `AddDispatchBehavior` он добавляет класс `myMessageInspector` в список инспекторов сообщений, чтобы он вызывался для каждого сообщения. Наконец, главная программа добавляет класс `myEndpointBehavior` в список поведений для всех оконечных точек. Отметим, что, поскольку в этой службе объявлена оконечная точка MEX, то `myEndpointBehavior` распечатывает заодно адресованные ей запросы и ответы на них.

Листинг 5.23. Инспектор сообщений в поведении оконечной точки службы

```
using System;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.IO;

namespace EssentialWCF
{
   public class myEndpointBehavior : IEndpointBehavior
   {
      public void AddBindingParameters
                  (ServiceEndpoint endpoint,
                   System.ServiceModel.Channels.BindingParameterCollection
                        bindingParameters)
      {
      }

      public void ApplyClientBehavior
                  (ServiceEndpoint endpoint,
                   ClientRuntime clientRuntime)
      {
      }
      public void ApplyDispatchBehavior
                  (ServiceEndpoint endpoint,
                   EndpointDispatcher endpointDispatcher)
      {
         endpointDispatcher.DispatchRuntime.MessageInspectors.Add(
                  new myMessageInspector());
      }
      public void Validate(ServiceEndpoint endpoint)
      {
      }
   }
   public class myMessageInspector : IDispatchMessageInspector
   {
      public object AfterReceiveRequest
                  (ref System.ServiceModel.Channels.Message request,
                  IClientChannel channel,
                  InstanceContext instanceContext)
      {
         Console.WriteLine(request.ToString());
         return request;
      }

      public void BeforeSendReply
                  (ref System.ServiceModel.Channels.Message reply,
                  object correlationState)
      {
         Console.WriteLine(reply.ToString());
      }
   }

   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      double GetPrice(string ticker);
   }

   public class StockService :IStockService
   {
      public double GetPrice(string ticker)
      {
         return 94.85;
      }
   }
   public class service
   {
      public static void Main()
      {
         ServiceHost serviceHost = new ServiceHost(typeof(StockService));

         foreach (ServiceEndpoint endpoint in
                        serviceHost.Description.Endpoints)
               endpoint.Behaviors.Add(new myEndpointBehavior());

         serviceHost.Open();

         // Теперь можно обращаться к службе.
         Console.WriteLine("Службы готовы к работе\n\n");
         Console.ReadLine();

         // Вызвать Close для ServiceHostBase, чтобы остановить службу.
         serviceHost.Close();

      }
   }
}
```

Раскрытие инспектора параметров для поведения операции службы в виде атрибута

В листинге 5.24 реализовано поведение для проверки параметров путем сопоставления с регулярными выражениями. Оно применимо к любой операции и позволяет разработчику определить регулярное выражение и сообщение об ошибке, которое возвращается, если параметр имеет недопустимое значение.

В коде иллюстрируется инспектор параметров, вызываемый из поведения операции, и способ реализации поведения в виде атрибута. Также показано, как поведение операции добавляется в описание службы путем задания атрибута.

Класс `myParameterInpsector` реализует интерфейс `IParameterInspector`. В этом классе есть два локальных свойства `_pattern` и `_message`, применяемые для проверки параметров в методе `BeforeCall`. В этом методе значение параметра сопоставляется с образцом, заданным в виде регулярного выражения. Если значение не соответствует образцу, возбуждается исключение.

Класс `myOperationBehavior` реализует интерфейсы `IEndpointBehavior` и `Attribute`. В методе `AddDispatchBehavior` он добавляет класс `myParameterInspector` в список инспекторов параметров, которые вызываются для каждой операции. Наконец, определение операции `GetPrice` снабжается атрибутом `myOperationBehavior`, с помощью которого параметры проверяются на этапе выполнения.

Листинг 5.24. Раскрытие инспектора параметров для поведения операции службы в виде атрибута

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.IO;

namespace EssentialWCF
{
   [AttributeUsage(AttributeTargets.Method)]
   public class myOperationBehavior : Attribute, IOperationBehavior
   {
      public string pattern;
      public string message;

      public void AddBindingParameters
                  (OperationDescription operationDescription,
                   BindingParameterCollection bindingParameters)
      {
      }

      public void ApplyClientBehavior
```



```
                  (OperationDescription operationDescription,
                   ClientOperation clientOperation)
      {
      }

      public void ApplyDispatchBehavior
                  (OperationDescription operationDescription,
                   DispatchOperation dispatchOperation)
      {
         dispatchOperation.ParameterInspectors.Add(
                  new myParameterInspector(this.pattern, this.message));
      }

      public void Validate(OperationDescription operationDescription)
      {
      }
   }

   public class myParameterInspector : IParameterInspector
   {
      string _pattern;
      string _message;
      public myParameterInspector(string pattern, string message)
      {
         _pattern = pattern;
         _message = message;
      }

      public void AfterCall(string operationName,
                  object[] outputs,
                  object returnValue, object
                  correlationState)
      {
      }

      public object BeforeCall(string operationName, object[] inputs)
      {
         foreach (object input in inputs)
         {
            if ((input != null) && (input.GetType() == typeof(string)))
            {
               Regex regex = new Regex(_pattern);
               if (regex.IsMatch((string)input))
                     throw new FaultException(string.Format(
                              "значение параметра вне диапазона:{0}, {1}",
                              (string) input, _message));
            }
         }
         return null;
      }
   }

   [ServiceContract]
   public interface IStockService
   {
```

```
      [OperationContract]
      double GetPrice(string ticker);
   }

   public class StockService : IStockService
   {
      [myOperationBehavior(pattern="[^a-zA-Z]",
                    message="Допустимы только буквы")]
      public double GetPrice(string ticker)
      {
         if (ticker == "MSFT") return 94.85;
         else return 0.0;
      }
   }
}
```

Задание поведения службы в конфигурационном файле

В листинге 5.25 реализовано поведение для проверки наличия лицензионного ключа, выполняемое на этапе конфигурирования службы. Если ключ отсутствует или неправильный, служба не запустится. На этом примере демонстрируется поведение оконечной точки, проверяющее конфигурационную информацию на этапе инициализации исполняющей среды. Также показано расширение поведения, которое вызывается на том же этапе, и то, как расширение добавляет поведение в исполняющую среду службы. В результате заказное поведение включается в конфигурационный файл (`app.config` или `web.config`) и добавляется в исполняющую среду, чтобы информацию можно было проверить при запуске службы.

Класс `myServiceBehavior` реализует интерфейс `IServiceBehavior` и содержит два локальных свойства: `_EvaluationKey` и `_EvaluationType`. Метод `Validate` класса `myEndpointBehavior` сравнивает эти свойства с предопределенными значениями.

Класс `myBehaviorExtensionElement` реализует интерфейс `IBehaviorExtensionElement`. В нем определены два свойства с атрибутом `[ConfigurationProperty]`, которые могут присутствовать в конфигурационном файле. Методы `BehaviorType` и `CreateBehavior` переопределены так, чтобы вернуть тип заказного поведения – `myServiceBehavior` – и создать объект этого типа на стадии инициализации исполняющей среды. Конструктор класса `myServiceBehavior` принимает два аргумента, по одному для каждого свойства, чтобы в дальнейшем можно было выполнить проверку.

Листинг 5.25. Поведение оконечной точки, задаваемое в конфигурационном файле

```
using System;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.ServiceModel.Configuration;
```



```
using System.Configuration;

namespace EssentialWCF
{
   public class myServiceBehavior : IServiceBehavior
   {
      string _EvaluationKey;
      string _EvaluationType;
      public myEndpointBehavior(string EvaluationKey, string EvaluationType)
      {
         _EvaluationKey = EvaluationKey;
         _EvaluationType = EvaluationType;
      }

      public void AddBindingParameters
                          (ServiceDescription serviceDescription,
                           ServiceHostBase serviceHostBase,
                           System.Collections.ObjectModel.
                                   Collection<ServiceEndpoint> endpoints,
                           BindingParameterCollection bindingParameters)
      {
      }

      public void ApplyClientBehavior(ServiceEndpoint endpoint,
                          ClientRuntime clientRuntime)
      {
      }

      public void ApplyDispatchBehavior(ServiceEndpoint endpoint,
                          EndpointDispatcher endpointDispatcher)
      {
      }

      public void Validate(ServiceDescription serviceDescription,
                          ServiceHostBase serviceHostBase)
      {
         if ((_EvaluationType == "Enterprise") &
               (_EvaluationKey != "SuperSecretEvaluationKey"))
               throw new Exception
                     (String.Format("Неправильный ключ апробирования.
                                 Type:{0}",_EvaluationType));
      }
   }

   public class myBehaviorExtensionElement : BehaviorExtensionElement
   {
      [ConfigurationProperty("EvaluationKey", DefaultValue = "",
                                 IsRequired = true)]
      public string EvaluationKey
      {
         get { return (string)base["EvaluationKey"]; }
         set { base["EvaluationKey"] = value; }
      }
      [ConfigurationProperty("EvaluationType",
                                 DefaultValue = "Enterprise",
```

```
                                IsRequired = false)]
    public string EvaluationType
    {
       get { return (string)base["EvaluationType"]; }
       set { base["EvaluationType"] = value; }
    }

    public override Type BehaviorType
    {
       get { return typeof(myServiceBehavior); }
    }

    protected override object CreateBehavior()
    {
       return new myServiceBehavior(EvaluationKey, EvaluationType);
    }

 }
 [ServiceContract]
 public interface IStockService
 {
    [OperationContract]
    double GetPrice(string ticker);
 }

 public class StockService : IStockService
 {
    public double GetPrice(string ticker)
    {
       if (ticker == "MSFT") return 94.85;
       else return 0.0;
    }
 }
}
```

В листинге 5.26 показан конфигурационный файл для этой службы. В него добавлен элемент `<behaviorExtension>`, ссылающийся на реализацию расширения. Отметим, что тип полностью квалифицирован, то есть в его описание включено имя типа и информация о сборке (название, номер версии, культура и открытый ключ). В данном примере сборка и библиотека (DLL), в которой находятся расширения, называется Server.

Листинг 5.26. Конфигурационный файл, в котором задано поведение оконечной точки

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>

      <extensions>
         <behaviorExtensions>
            <add name ="FreeTrial"
                    Type = "EssentialWCF.myBehaviorExtensionElement, Server,
                    Version=1.0.0.0, Culture=neutral,
                    PublicKeyToken=null"/>
         </behaviorExtensions>
      </extensions>

      <services>
         <service name="EssentialWCF.StockService"
                       behaviorConfiguration="customBehavior">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EssentialWCF"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                          binding="basicHttpBinding"
                          contract="EssentialWCF.StockService" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="customBehavior">
               <FreeTrial EvaluationKey="SuperSecretEvaluationKey"
                                 EvaluationType="Enterprise"/>
            </behavior>
         </serviceBehaviors>
      </behaviors>

   </system.serviceModel>
</configuration>
```


Поведения, касающиеся безопасности

Существует несколько важных поведений, относящихся к безопасности. Подробно о них рассказано в главе 8, но уже сейчас важно понимать, какую роль они играют в качестве поведений.

`ServiceCredentials` – поведение службы, применяемое для задания ее верительных грамот. Этот класс полезен для доступа к информации о безопасности на стороне клиента, например, к параметрам безопасности Windows или клиентским сертификатам. Он реализован в виде поведения, чтобы можно было инспектировать входящие сообщения на предмет информации, относящейся к безопасности.

Атрибут `Impersonation` – это поведение операции, позволяющее олицетворить клиента на стороне службы. Если олицетворение (impersonation) разрешено или требуется, то переданные по каналу верительные грамоты клиента используются исполняющей средой WCF, чтобы притвориться данным клиентом на время выполнения операции службы.

Поведение `ServiceAuthorization`, работающее на уровне службы или операции, используется, чтобы авторизовать клиента для доступа к службе в целом или к отдельной операции. С его помощью вы можете задать менеджер авторизации `AuthorizationManager`, который отвечает за рассмотрение «просьб» пользователя и решает, дать ли ему доступ.

Поведение `ServiceSecurityAudit` задается в секции `<serviceBehaviors>` конфигурационного файла и определяет, какую информацию следует автоматически записывать в протокол при каждом запросе к службе. Если это поведение не задано, то аудит не производится вовсе.

Резюме

Поведения – это одна из важнейших точек расширения в архитектуре WCF. Они используются на этапе инициализации исполняющей среды клиента и службы, а также при каждом вызове службы. Когда инициализируется исполняющая среда, поведения ищутся путем анализа типов, переданных в виде атрибутов `ClientChannel` или `ServiceHost`, а затем в конфигурационных файлах. Поведения можно также определить в коде и добавить в объект `ServiceDescription` перед открытием `ServiceHost`.

С помощью поведений реализуются инспекторы – классы, инспектирующие переданные им объекты. Существуют инспекторы, работающие на уровне сообщений, и инспекторы, анализирующие параметры операций. На уровне операций поведения привлекаются к выбору методу, который надлежит вызвать для обработки входящего сообщения SOAP, и далее на стадии вызова этого метода.

Поведения управляют созданием экземпляров службы и параллелизмом работы на уровне службы и отдельных операций. Создание экземпляров контролирует параметр `InstanceContextMode`, который может принимать значения `Single`, `PerCall` или `PerSession`. Параллелизмом управляет параметр `ConcurrencyMode`, принимающий значения `Single`, `Multiple` или `Reentrant` (реентерабельный код в одном потоке). Совместно эти параметры определяют уровень параллелизма службы: от синглета на одном конце спектра до создания нового экземпляра для каждого обращения – на другом. Поведения службы позволяют также управлять пропускной способностью за счет ограничения числа одновременных обращений, экземпляров или сеансов.

Одним из важных поведений службы является `serviceMetaData`. Оно раскрывает систему типов WCF и модель `ServiceDescription` в виде WSDL-документа, чтобы клиенты знали, куда, как и что отправлять службе. В общем случае поведение оконечной точки `serviceMetaData` пользуется протоколом WS-MEX, оставляя возможность работать с клиентами на разных платформах и возвращать метаданные в различных форматах. Лишь в том случае, когда служба включает это поведение, она будет отдавать WSDL-документ. Этим WCF отличается от привычных многим разработчикам ASMX-служб, которые раскрывают WSDL-описание по умолчанию.

В WCF с помощью поведений реализованы короткие ACID-транзакции. Транзакции могут работать строго внутри службы или пересекать ее границы. Транзакции реализованы на уровне поведений операции, хотя для корректной работы должны удовлетворяться некоторые условия, формулируемые на уровне оконечных точек и службы в целом. Так, для передачи транзакционного контекста от клиента к службе оконечные точки должны пользоваться протоколом, поддерживающим сеансы, например `wsHttpBinding`. Сеансы необходимо разрешить на уровне службы, чтобы поток транзакций мог пересекать ее границы. Поведения операций позволяют разработчикам задавать уровень изолированности транзакций, а администраторам – величины таймаутов.

WCF поддерживает три менеджера транзакций: для локальных транзакций внутри одного процесса, для ресурсов Vista и координатор распределенных транзакций (Distributed Transaction Coordinator). Для коммуникации через границы службы можно использовать как протокол транзакций на базе механизма RPC, работающий только на платформе Windows, так и стандартный протокол WS-AT.

Разработчики могут самостоятельно создавать заказные поведения на уровне службы, оконечной точки, контракта или операции. С их помощью можно инспектировать и модифицировать входящие и исходящие сообщения на стороне службы или клиента. Они же позволяют разработчику или системному администратору контролировать или модифицировать исполняющую среду WCF на этапе запуска службы.

Глава 6. Сериализация и кодирование

В главе 2 «Контракты» мы встречались с сериализацией, когда обсуждали, как класс `DataContract` преобразует типы CLR в стандартный формат XML для передачи между клиентом и службой. Однако есть немало ситуаций, когда сериализации с помощью `DataContract` недостаточно. В частности, это относится к случаям, когда существующий тип CLR не поддерживает сериализацию по контракту о данных, к унаследованным Web-службам, к интероперабельности, к переносу кода с других платформ (например, .NET Remoting) и к *формированию данных* (data shaping). Формирование данных – это процедура управления сериализацией типа .NET в формат XML, применяемая в целях оптимизации. Во всех этих случаях важно знать, как воспользоваться средствами сериализации, которые предоставляют WCF и .NET Framework.

Кодирование – еще одна важная тема, неразрывно связанная с сериализацией. В WCF проводится различие между сериализацией объектов и преобразованием сообщений в массив байтов, который можно послать с помощью транспортного протокола.

Сравнение сериализации и кодирования

Между сериализацией и кодированием в WCF имеются существенные различия. Смысл этих терминов в WCF несколько отличается от того, что принято понимать под ними в других технологиях распределенных вычислений (например, Web-службы в ASP.NET и COM).

Словом *сериализация* часто описывают процедуру преобразования графа объектов в массив байтов. Это очень полезно для представления состояния объекта. Сериализации позволяет сохранить состояние объекта в базе данных, скопировать объект в буфер обмена или передать его по сети другому приложению. Однако к WCF стандартное определение сериализации неприменимо. В WCF сериализацией называется процедура преобразования графа объектов в информационный набор XML (XML Infoset). Это определение не ново, для Web-служб ASP.NET был принят такой же подход. XML Infoset – это модель данных, используемая в WCF для внутреннего представления сообщения. Класс `System.ServiceModel.Channels.Message` представляет сообщение в виде информационного набора. XML Infoset определяет модель данных для представления XML-документа. Кроме того, это базовая абстракция, из которой выводится спецификация XML. На рис. 6.1 показано соотношение между языком XML и информационным набором XML.

Рис. 6.1. Информационный набор XML

Ключевое различие между XML и информационным набором XML состоит в том, что XML Infoset не определяет никакого конкретного формата. Если в стандарте XML используется текстовый формат, что в спецификации XML Infoset такого ограничения нет. Работать на уровне XML Infoset зачастую удобнее, чем напрямую с XML. Например, в WCF сообщения можно представлять в разных форматах при условии, что все они основаны на спецификации XML Infoset. В их число входит и текстовый формат, определенный в XML 1.1, и ряд других, например, двоичный формат XML. Это позволяет WCF работать с XML, оставляя возможность использовать тот формат, который оптимален с точки зрения интероперабельности и производительности.

Термином *кодирование* описывается процедура преобразования WCF-сообщения в массив байтов. Делается это для того, чтобы передать сообщение по транспортному протоколу. В WCF поддерживается пять форматов кодирования: двоичный, текстовый, механизм оптимизации передачи сообщения (Message Transmission Optimization Mechanism – MTOM), JavaScript-нотация объектов (JavaScript Object Notation – JSON) и Plain-Old-XML (POX). Какой из них использовать, зависит от потребностей приложения. Например, для достижения максимальной производительности обмена между приложениями .NET можно выбрать кодировщик `BinaryMessageEncoder`, для интероперабельности с Web-службами на базе стандартов WS-* – `TextMessageEncoder` или `MtomMessageEncoder`, а для работы с Web-приложениями на базе технологии AJAX – `JsonMessageEncoder`. Кодировщики – это один из механизмов расширения WCF, поэтому можно добавлять новые кодировщики, если имеющихся недостаточно для решения конкретной задачи.

В этой главе мы рассмотрим, как сериализация и кодирование используются в WCF для передачи сообщений по сети. Мы изучим различные виды сериализации и кодирования и покажем, когда лучше применять тот или иной способ.

Сравнение вариантов сериализации, имеющихся в WCF

В WCF есть много способов сериализации объектов. Решение о том, какому отдать предпочтение, зависит от многих факторов, в частности от того, хотите ли вы обобществлять типы или контракты, поддерживать существующие типы .NET, сохранять ссылки и т.д.

Класс DataContractSerializer

По умолчанию в WCF для сериализации применяется класс `DataContractSerializer`, который находится в пространстве имен `System.Runtime.Serialization`. Он поддерживает обобществление контрактов, основанных на XSD-схемах, и отображает типы CLR на типы, определенные в спецификации XSD. Это означает, что XSD – общая схема, используемая для обмена данными между двумя приложениями, например, написанными на .NET-языке и Java. Примером может служить работа со строками.

Отметим, что между клиентом и сервером не передаются никакие другие типы, кроме XSD. Так, на рис. 6.2 в коммуникации не участвует ни один из типов `System.String` или `java.lang.String`. Это позволяет любой стороне отображать XSD-типы на типы, определенные в той платформе, на которой она работает. Такой подход хорошо работает для примитивных типов. Составные типы трактуются как расширения примитивных. Так каким же образом описать отображение произвольного типа .NET CLR на XSD-схему с помощью класса `DataContractSerializer`?

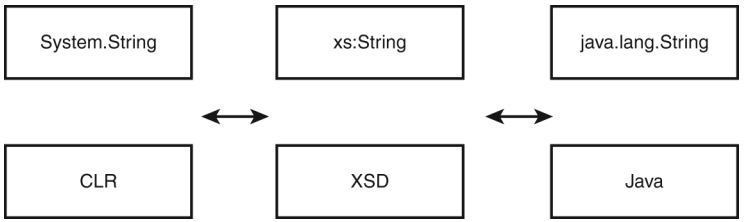

Рис. 6.2. Типы XSD

В главе 2 было сказано, что атрибут `[DataContract]` позволяет пометить тип как сериализуемый. Члены и свойства типа, помеченные атрибутом `[DataMember]`, становятся частью контракта о данных. Требуется, чтобы разработчик явно указывал в контракте способ сериализации типа. Напротив, класс `XmlSerializer` не требует явного задания. В листинге 6.1 приведен составной тип `Employee`, для которого используется класс `DataContractSerializer`. На его примере мы изучим схему и сериализованное представление, создаваемое классом `DataContractSerializer`. Это ляжет в основу сравнения с другими механизмами сериализации, имеющимися в WCF.

Листинг 6.1. Использование DataContractSerializer для класса Employee

```
using System.Runtime.Serialization;

[DataContract]
public class Employee
{
   private int employeeID;
   private string firstName;
   private string lastName;

   public Employee(int employeeID, string firstName, string lastName)
   {
      this.employeeID = employeeID;
      this.firstName = firstName;
      this.lastName = lastName;
   }

   [DataMember]
   public int EmployeeID
   {
      get { return employeeID; }
      set { employeeID = value; }
   }

   [DataMember]
   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }

   [DataMember]
   public string LastName
   {
      get { return lastName; }
      set { lastName = value; }
   }
}
```


В листинге 6.2 показано представление класса `Employee` в виде XSD-схемы.

Листинг 6.2. XSD-схема класса Employee

```
<xs:schema xmlns:tns=http://schemas.datacontract.org/2004/07/
➥elementFormDefault="qualified" targetNamespace=
➥"http://schemas.datacontract.org/2004/07/"
➥xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:complexType name="Employee">
      <xs:sequence>
         <xs:element minOccurs="0" name="EmployeeID" type="xs:int" />
         <xs:element minOccurs="0" name="FirstName" nillable="true"
                           type="xs:string" /
         <xs:element minOccurs="0" name="LastName" nillable="true"
                           type="xs:string" />
      </xs:sequence>
   </xs:complexType>
   <xs:element name="Employee" nillable="true" type="tns:Employee" />
</xs:schema>
```

В листинге 6.3 показано, как экспортируется эта схема.

Листинг 6.3. Экспорт XSD-схемы

```
using System.IO;
using System.Runtime.Serialization;
```

```
using System.Xml.Schema;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         XsdDataContractExporter xsdexp = new XsdDataContractExporter();
         xsdexp.Options = new ExportOptions();
         xsdexp.Export(typeof(Employee));

         // Вывести экспортрованную схему в файл
         using (FileStream fs = new FileStream("sample.xsd", FileMode.Create))
         {
               foreach (XmlSchema sch in xsdexp.Schemas.Schemas())
                     sch.Write(fs);
         }
      }
   }
}
```

Последнее, что нам понадобится для сравнения с другими механизмами, – это сериализация экземпляра класса `Employee` с помощью `DataContractSerializer`. В листинге 6.4 показано, как это делается.

Листинг 6.4. Сериализация с использованием класса DataContractSerializer

```
using System.IO;
using System.Runtime.Serialization;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         Employee e = new Employee(101, "John", "Doe");
         FileStream writer = new FileStream("sample.xml", FileMode.Create);
         DataContractSerializer ser = new DataContractSerializer(typeof(Employee));
         ser.WriteObject(writer, e);
         writer.Close();
      }
   }
}
```

Сериализованное представление `Employee`, полученное с помощью класса `DataContractSerializer`, показано в листинге 6.5.

Листинг 6.5. Класс Employee, сериализованный с помощью DataContractSerializer

```
<Employee   xmlns="http://schemas.datacontract.org/2004/07/"
               xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
   <EmployeeID>101</EmployeeID>
   <FirstName>John</FirstName>
   <LastName>Doe</LastName>
</Employee>
```


Класс NetDataContractSerializer

Альтернативный механизм сериализации в WCF, который позволяет обобществлять типы, – это класс `NetDataContractSerializer`, находящийся в пространстве имен `System.Runtime.Serialization`. Его можно применять, когда типы на сторонах клиента и сервера должны быть идентичны. Чтобы обеспечить идентичность, `NetDataContractSerializer` включает дополнительную информацию, необходимую для воссоздания типа CLR и сохранения ссылок. Это единственное отличие `NetDataContractSerializer` от `DataContractSerializer`.

Обобществление информации о типе идет вразрез с принципом, согласно которому общими являются только контракты. Поэтому `NetDataContractSerializer` не предназначен для создания служб, объединяющих различные приложения; его следует использовать только в пределах одного приложения. Именно по этой причине класс `NetDataContractSerializer` не вошел в состав WCF и воспользоваться им можно, только написав дополнительный код. О том, как активировать это средство, речь пойдет в разделе «Обобществление типов с помощью NetDataContractSerializer».

Посмотрим, как конкретный объект класса `Employee` сериализуется с помощью `DataContractSerializer` и `NetDataContractSerializer`. В листинге 6.4 уже была продемонстрирована сериализация с помощью `DataContractSerializer`, а в листинге 6.6 показано, как то же самое делается с применением `NetDataContractSerializer`. Отметим, что конструктору `NetDataContractSerializer` не нужно передавать тип, так как он распознает тип класса `Employee` во время выполнения.

Листинг 6.6. Сериализация с использованием класса NetDataContractSerializer

```
using System.IO;
using System.Runtime.Serialization;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         Employee e = new Employee(101, "John", "Doe");
         FileStream writer = new FileStream("sample.xml", FileMode.Create);
         NetDataContractSerializer ser = new NetDataContractSerializer();
         ser.WriteObject(writer, e);
         writer.Close();
      }
   }
}
```

В листинге 6.7 показан результат сериализации класса `Employee`. Обратите внимание, что `NetDataContractSerializer` включил имя сборки (`Assembly`) и типа (`Type`). Эту дополнительную информацию можно использовать для десериализации XML-представления в указанный тип. Следовательно, клиент и сервер будут работать с одним и тем же типом. Еще одно отличие – это атрибут `z:Id`, задаваемый для различных элементов. Он имеет отношение к ссылочным типам и к сохранению ссылок во время десериализации. О том, как сохраняются ссылки, мы будем говорить в разделе «Сохранение ссылок и циклических ссылок» ниже.

Листинг 6.7. Класс Employee, сериализованный с помощью NetDataContractSerializer

```
<Employee z:Id="1" z:Type="Employee" z:Assembly="DataContract,
➥Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
➥xmlns="http://schemas.datacontract.org/2004/07/"
➥xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
➥xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
   <EmployeeID>101</EmployeeID>
   <FirstName z:Id="2">John</FirstName>
   <LastName z:Id="3">Doe</LastName>
</Employee>
```

Класс XmlSerializer

Третий способ сериализации в WCF основан на использовании класса `XmlSerializer`. Этот механизм встроен в каркас .NET 2.0. К числу его достоинств можно отнести поддержку существующих типов .NET, совместимость с Web-службами ASP.NET и способность к формированию результата вывода в формате XML.

WCF поддерживает `XmlSerializer`, поэтому может работать с существующими типами, тогда как `DataContractSerializer` разработан специально для поддержки новых типов. Поддерживать существующие типы бывает необходимо, когда приходится работать с унаследованными приложениями или компонентами третьих фирм, для которых у вас нет исходного кода, поэтому переработать программу с учетом сериализации по контракту о данных и перекомпилировать ее невозможно. Класс `XmlSerializer` применяется для сериализации также и в Web-службах, разработанных в ASP.NET. Следовательно, он помогает конвертировать такие службы в WCF. Наконец, `XmlSerializer` обеспечивает максимальный контроль над преобразованием объекта в формат XML и может применяться в ситуациях, когда класса `DataContractSerializer` недостаточно для формирования нужного результата.

При работе с `XmlSerializer` возможно три подхода. Первый – согласиться с сериализацией, применяемой по умолчанию. Класс `XmlSerializer` требует, чтобы в сериализуемом типе был открытый конструктор, и сериализует все открытые поля и открытые свойства, допускающие чтение и запись. Предполагается, что для воссоздания класса нужно будет вызвать конструктор по умолчанию, а затем задать значения полей и свойств. Этот простой подход работает только, если вы

специально проектировали класс с учетом данного метода сериализации. Кроме того, чтобы сериализовать внутренние члены класса, вам придется раскрыть их для всего мира. Второй подход основан на применении атрибутов `[XmlElement]` и `[XmlAttribute]`, которыми помечаются открытые поля и открытые свойства, допускающие чтение и запись. Эти и другие атрибуты для управления результатом преобразования в XML находятся в пространстве имен `System.Xml.Serialization`. Подход на основе атрибутов позволяюе контролировать представление открытых полей и свойств в формате XML, но и он подвержен ограничениям на способ сериализации и требует раскрытия внутренних структур данных. Третий подход предполагает использование интерфейса `IXmlSerializable` и позволяет полностью перехватить контроль над сериализацией.

Мы еще будем говорить о нестандартной сериализации с помощью класса `XmlSerializer` ниже в этой главе, а пока рассмотрим простейший пример. В листинге 6.8 показан класс `Employee`, спроектированный для сериализации с помощью `XmlSerializer` без поддержки со стороны `DataContractSerializer` (атрибуты `[DataContract]` и `[DataMember]` удалены). Для этого необходим конструктор по умолчанию.

Листинг 6.8. Класс Employee, разработанный с учетом XmlSerializer

```
public class Employee
{
   private int employeeID;
   private string firstName;
   private string lastName;

   public Employee()
   {
   }

   public Employee(int employeeID, string firstName, string lastName)
   {
      this.employeeID = employeeID;
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public int EmployeeID
   {
      get { return employeeID; }
      set { employeeID = value; }
   }

   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }

   public string LastName
   {
```

```
      get { return lastName; }
      set { lastName = value; }
   }
}
```

В листинге 6.9 показано, как объект класса `Employee` сериализуется с помощью `XmlSerializer`.

Листинг 6.9. Сериализация с использованием класса XmlSerializer

```
using System.IO;
using System.Xml.Serialization;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         Employee e = new Employee(101, "John", "Doe");
         FileStream writer = new FileStream("sample.xml", FileMode.Create);
         XmlSerializer ser = new XmlSerializer((typeof(Employee)));
         ser.Serialize(writer, e);
         writer.Close();
      }
   }
}
```

В листинге 6.10 показан результат сериализации с помощью `XmlSerializer`. Он очень похож на то, что получилось в результате применения `DataContractSerializer`. Основное различие заключается в том, что не показано. `XmlSerializer` поддерживает гораздо меньше типов, чем `DataContractSerializer`, зато обеспечивает больший контроль над результирующим XML-представлением.

Листинг 6.10. Класс Employee, сериализованный с помощью XmlSerializer

```
<?xml version="1.0"?>
<Employee  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <EmployeeID>101</EmployeeID>
   <FirstName>John</FirstName>
   <LastName>Doe</LastName>
</Employee>
```

Показанный в листинге 6.8 класс `Employee` нельзя сериализовать без помощи `XmlSerializer`. Чтобы сообщить WCF о необходимости применить `XmlSerializer`, можно снабдить контракт о службе, контракт об операции или саму службу атрибутом `[XmlSerializerFormat]`. В листинге 6.11 мы пометили этим атрибутом контракт о службе. Как правило, атрибут `[XmlSerializerFormat]` задается для раскрытия любого контракта, в котором используется класс `XmlSerializer`. Увидев этот атрибут, Visual Studio и утилита `svcutil.exe` сгенерируют соответствующие прокси-классы. Если же он отсутствует, что для генерации подходящих прокси-классов `svcutil.exe` следует запускать с флагом `/Serializer:XmlSerializer`.

Атрибут `DataContractFormat`**.** Существует также аналогичный атрибут `[DataContractFormat]`, говорящий о необходимости использовать класс `DataContractSerializer`. Но, поскольку WCF использует `DataContractSerializer` по умолчанию, явно указывать этот атрибут не имеет смысла.

Листинг 6.11. Использование атрибута XmlSerializerFormat

```
using System.Collections.Generic;
using System.ServiceModel;

namespace EssentialWCF
{
   [ServiceContract]
   [XmlSerializerFormat]
   public interface IEmployeeInformation
   {
      [OperationContract]
      List<Employee> GetEmployees();
   }
}
```

Класс DataContractJsonSerializer

Класс `DataContractJsonSerializer` поддерживает формат сериализации JavaScript Object Notation (JSON) и включен в версию .NET Framework 3.5. Этот способ сериализации удобен для вызова служб из Web-приложения с помощью JavaScript, особенно для приложений, написанных с использованием ASP.NET AJAX и Silverlight. Класс `DataContractJsonSerializer` применяется, когда задано поведение `WebScriptEnablingBehavior`. Или когда поведение `WebHttpBehavior` сконфигурировано для кодирования в формате JSON. Эти поведения оконечной точки говорят WCF о необходимости поддержать службы, соответствующие спецификации REST/POX. Дополнительную информацию об этих атрибутах см. в главе 13. А пока мы посмотрим, как можно воспользоваться классом `DataContractJsonSerializer` напрямую, и сравним результат с рассмотренными выше механизмами. В листинге 6.12 показано, как сериализовать объект `Employee` с помощью `DataContractJsonSerializer`.

Листинг 6.12. Сериализация с помощью класса DataContractJsonSerializer

```
using System.IO;
using System.Runtime.Serialization.Json;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         Employee e = new Employee(101, "John", "Doe");
         FileStream writer = new FileStream("sample.xml", FileMode.Create);
```

```
            DataContractJsonSerializer ser = new
               DataContractJsonSerializer(typeof(Employee));
            ser.WriteObject(writer, e);
            writer.Close();
        }
    }
}
```

Класс `DataContractJsonSerializer` следует тем же правилам, что `DataContractSerializer`, но выводит данные в формате JSON, а не XML Сериализованное с его помощью представление объекта `Employee` приведено в листинге 6.13. Результат оказывается гораздо более компактным и простым для восприятия, чем в случае `DataContactSerializer`, `NetDataContractSerializer` или `XmlSerializer`.

Листинг 6.13. Класс Employee, сериализованный с помощью DataContractJsonSerializer

```
{"EmployeeID":101,
 "FirstName":"John",
 "LastName":"Doe"}
```

Выбор сериализатора

Решить, какой из классов `DataContractSerializer`, `NetDataContractSerializer`, `XmlSerializer` или `DataContractJsonSerializer` выбрать, часто бывает легко. Класс `DataContractSerializer` следует применять по умолчанию, так как именно он реализует «родной» для WCF механизм сериализации. Но, если требуется поддержать существующие типы, с которыми `DataContractSerializer` не справляется, то следующим кандидатом будет `XmlSerializer`. Хотя класс `NetDataContractSerializer` тоже интересен, но прямой поддержки для него нет, поэтому придется писать код самостоятельно. Несмотря на некоторые достоинства, мы не рекомендуем пользоваться классом `NetDataContractSerializer`, так как он требует, чтобы типы на стороне клиента и сервера были идентичны. Наконец, класс `DataContractJsonSerializer` применяется главным образом в службах, которые вызываются из Web-приложений, написанных с помощью технологии AJAX. Если вы планируете вести разработку на платформе ASP.NET AJAX или создавать обогащенные Интернет-приложения (Rich Internet Applications – RIA) с помощью Silverlight, то стоит обратить внимание на JSON-сериализацию, которую обеспечивает класс `DataContractJsonSerializer`. Хотя этот формат характерен, прежде всего, для Web-приложений, написанных частично на языке JavaScript, но популярность его велика, поэтому в других контекстах он тоже встречается. Использовать ли его в таких случаях – вопрос личного вкуса. Наконец, WCF предоставляет ряд точек расширения, позволяющих в частности полностью подменить механизм сериализации.

Сохранение ссылок и циклических ссылок

С сериализацией связаны два важных вопроса: сохранение ссылок и *циклических ссылок*. Оба решаются единым механизмом *сохранения ссылок*. Это может оказаться очень существенным для уменьшения объема данных в сериализованном представлении, а также для обобществления информации о типе между клиентом и сервером.

Механизм сохранения ссылок позволяет ссылаться на одни и те же данные, не дублируя их. Это часто бывает необходимо при работе с такими структурами данных, как списки, массивы и хэш-таблицы, где одни и те же данные могут встречаться несколько раз. Описываемый механизм позволяет сериализовать каждый элемент данных, когда он встречается впервые, а впоследствии ссылаться на сериализованные данные. В результате, если один и тот же элемент встречается многократно, то размер результата сериализации заметно сокращается.

Циклические ссылки возникают, когда один объект ссылается на другой, а тот в свою очередь ссылается на первый объект. Примером может служить отношение родитель-потомок, когда объект-потомок хранит ссылку на родителя. Такие ситуации встречаются в объектно-ориентированном программировании сплошь и рядом. Сериализация объектов с циклическими ссылками без механизма сохранения ссылок вообще невозможна, поскольку при этом возникали бы бесконечные циклы. А так мы можем сохранить ссылку на элемент данных, а не сериализовывать его снова и снова.

Класс `DataContractSerializer` по умолчанию не включает сохранение ссылок. Классы `NetDataContractSerializer` и `XmlSerializer` сохраняют ссылки всегда. Если вы собираетесь обобществлять информацию о типе между клиентом и сервером, пользуйтесь одним из этих механизмов сериализации. В противном случае можете воспользоваться нестандартным атрибутом, чтобы поддержать сохранение ссылок в сериализаторе `DataContractSerializer`.

Сохранение ссылок при использовании интерфейса `IXmlSerializable`. Если вы собираетесь организовать нестандартную сериализацию с помощью интерфейса `IXmlSerializable`, то программировать сохранение ссылок придется самостоятельно.

Рассмотрим пример, приведенный в листинге 6.14. Сначала создается список типа `List<Employee>`. Затем в него добавляются несколько объектов типа `Employee`.

Листинг 6.14. Необходимость сохранения ссылок

```
using System.Collections.Generic;
using System.ServiceModel;

[ServiceContract]
```

```
public interface IEmployeeInformation
{
   [OperationContract]
   Employee[] GetEmployeesOfTheMonth();
}

[ServiceContract]
public class EmployeeInformation
{
   public EmployeeInformation()
   {
   }

   public Employee[] GetEmployeesOfTheMonthForLastSixMonths()
   {
      List<Employee> list = new List<Employee>(6);
      Employee Employee1 = new Employee(1, "John", "Doe");
      Employee Employee2 = new Employee(2, "Jane", "Doe");
      Employee Employee3 = new Employee(3, "John", "Smith");

      list.Add(Employee1);
      list.Add(Employee2);
      list.Add(Employee3);
      list.Add(Employee1);
      list.Add(Employee2);
      list.Add(Employee3);

      return list.ToArray();
   }
}
```

По умолчанию `DataContractSerializer` сериализует каждую ссылку, создавая отдельную копию данных. В листинге 6.15 видно, что каждый из объектов `Employee1`, `Employee2` и `Employee3` встречается дважды.

Листинг 6.15. Список, сериализованный без сохранения ссылок

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
➥xmlns:a="http://www.w3.org/2005/08/addressing">
   <s:Header>    <a:Actions:mustUnderstand="1">
➥http://tempuri.org/IEmployeeInformation/
➥GetEmployeesOfTheMonthForLastSixMonthsResponse</a:Action>
   <a:RelatesTo>urn:uuid:0a35e6da-1dad-47f6-bb95-bfcdf3fe856c</a:RelatesTo>
   </s:Header>
   <s:Body>
      <GetEmployeesOfTheMonthForLastSixMonthsResponse xmlns=
➥"http://tempuri.org/">
         <GetEmployeesOfTheMonthForLastSixMonthsResult
➥xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
            <Employee xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>1</EmployeeID>
               <FirstName>John</FirstName>
               <LastName>Doe</LastName>
            </Employee>
            <Employee xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>2</EmployeeID>
               <FirstName>Jane</FirstName>
               <LastName>Doe</LastName>
            </Employee>
            <Employee xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>3</EmployeeID>
               <FirstName>John</FirstName>
               <LastName>Smith</LastName>
            </Employee>
            <Employee xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>1</EmployeeID>
               <FirstName>John</FirstName>
               <LastName>Doe</LastName>
            </Employee>
            <Employee xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>2</EmployeeID>
               <FirstName>Jane</FirstName>
               <LastName>Doe</LastName>
            </Employee>
            <Employee xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>3</EmployeeID>
               <FirstName>John</FirstName>
               <LastName>Smith</LastName>
            </Employee>
         </GetEmployeesOfTheMonthForLastSixMonthsResult>
      </GetEmployeesOfTheMonthForLastSixMonthsResponse>
   </s:Body>
</s:Envelope>
```


Чтобы сохранять ссылки, применим заказное поведение для создания экземпляра `DataContractSerializer`, передав конструктору параметр `preserveObjectReferences`, равный `true`. Поведение – это механизм расширения WCF, позволяющий модифицировать работу исполняющей среды; подробнее см. главу 5. В данном случае мы изменяем принимаемое по умолчанию поведение класса `DataContractSerializer`, чтобы организовать сохранение ссылок. В листинге 6.16 приведена реализация этого поведения.

Листинг 6.15. Реализация заказного поведения, включающего механизм сохранения ссылок

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.Xml;

namespace EssentialWCF
{
   public class ReferencePreservingDataContractFormatAttribute :
                     Attribute, IOperationBehavior
   {
```

```
      public void AddBindingParameters(
                           OperationDescription description,
                           BindingParameterCollection parameters)
      {
      }

      public void ApplyClientBehavior(
                           OperationDescription description,
                           ClientOperation proxy)
      {
         IOperationBehavior innerBehavior = new
➥ReferencePreservingDataContractSerializerOperationBehavior(description);
         innerBehavior.ApplyClientBehavior(description, proxy);
      }

      public void ApplyDispatchBehavior(
                           OperationDescription description,
                           DispatchOperation dispatch)
      {
         IOperationBehavior innerBehavior = new
➥ReferencePreservingDataContractSerializerOperationBehavior(description);
         innerBehavior.ApplyDispatchBehavior(description, dispatch);
      }

      public void Validate(OperationDescription description)
      {
      }
   }

   public class
➥ReferencePreservingDataContractSerializerOperationBehavior :
                     DataContractSerializerOperationBehavior
   {
      public ReferencePreservingDataContractSerializerOperationBehavior(
               OperationDescription operationDescription)
         : base(operationDescription)
      {
      }

      public override XmlObjectSerializer CreateSerializer(Type type,
               string name, string ns, IList<Type> knownTypes)
      {
         return CreateDataContractSerializer(type, name, ns, knownTypes);
      }

      private static XmlObjectSerializer
         CreateDataContractSerializer(Type type, string name,
                        string ns, IList<Type> knownTypes)
      {
         return CreateDataContractSerializer(type, name, ns, knownTypes);
      }

      public override XmlObjectSerializer CreateSerializer(Type type,
                     XmlDictionaryString name, XmlDictionaryString ns,
                     IList<Type> knownTypes)
```



```
      {
         return new DataContractSerializer(type, name, ns,
                        knownTypes, 0x7FFF, false, true, null);
      }
   }
}
```

Результат применения этого атрибута к нашему контракту об операции показан в листинге 6.17. Видно, что объекты `Employee1`, `Employee2` и `Employee3` встречаются только по одному разу, но теперь они помечены атрибутом `z:Id`, выступающим в роли идентификатора ссылки. Дополнительные ссылки на те же объекты содержат атрибут `z:Ref` с тем же значением, что у `z:Id`.

Листинг 6.17. Список, сериализованный с сохранением ссылок

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
➥xmlns:a="http://www.w3.org/2005/08/addressing">
   <s:Header>
      <a:Action s:mustUnderstand="1">http://tempuri.org/IEmployeeInformation/
➥GetEmployeesOfTheMonthForLastSixMonthsResponse</a:Action>
      <a:RelatesTo>urn:uuid:9331e9f4-9991-447e-812d-db1b129bfb25</a:RelatesTo>
   </s:Header>
   <s:Body>
      <GetEmployeesOfTheMonthForLastSixMonthsResponse
➥xmlns="http://tempuri.org/">
         <GetEmployeesOfTheMonthForLastSixMonthsResult z:Id="1" z:Size="6"
➥xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
➥xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
            <Employee z:Id="2" xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>1</EmployeeID>
               <FirstName z:Id="3">John</FirstName>
               <LastName z:Id="4">Doe</LastName>
            </Employee>
            <Employee z:Id="5"
                  xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>2</EmployeeID>
               <FirstName z:Id="6">Jane</FirstName>
               <LastName z:Id="7">Doe</LastName>
            </Employee>
            <Employee z:Id="8"
                  xmlns="http://schemas.datacontract.org/2004/07/">
               <EmployeeID>3</EmployeeID>
               <FirstName z:Id="9">John</FirstName>
               <LastName z:Id="10">Smith</LastName>
            </Employee>
            <Employee z:Ref="2" i:nil="true"
                  xmlns="http://schemas.datacontract.org/2004/07/"/>
            <Employee z:Ref="5" i:nil="true"
                  xmlns="http://schemas.datacontract.org/2004/07/"/>
            <Employee z:Ref="8" i:nil="true"
                  xmlns="http://schemas.datacontract.org/2004/07/"/>
         </GetEmployeesOfTheMonthForLastSixMonthsResult>
      </GetEmployeesOfTheMonthForLastSixMonthsResponse>
   </s:Body>
</s:Envelope>
```

Обобществление типов с помощью класса NetDataContractSerializer

По умолчанию в WCF для сериализации применяется класс `DataContractSerializer`. Именно его авторы WCF рекомендуют большинству разработчиков, так как при этом обобществляются контракты, а не типы, а этот принцип лежит в основе сервисно-ориентированных архитектур. Однако, если вам нужно воспроизвести тип максимально достоверно, обобществив информацию о нем между клиентом и службой, и это не влечет негативных последствий для проекта, то можно применить для сериализации класс `NetDataContractSerializer`. Выше в разделе «Сравнение вариантов сериализации, имеющихся в WCF» мы уже упоминали, что по существу это тот же класс, что `DataContractSerializer`, но с поддержкой обобществления информации о типе и сохранения ссылок.

Хотя WCF поддерживает класс `NetDataContractSerializer`, но никакого атрибута, позволяющего использовать его для сериализации контракта данных, не существует. Сделано это специально, чтобы разработчики не относились к обобществлению типов с легким сердцем. Чтобы воспользоваться классом `NetDataContractSerializer`, необходимо написать заказное поведение (листинг 6.18) и применить его к контрактам об операциях (листинг 6.19).

Листинг 6.18. Использование атрибута NetDataContractFormatAttribute

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.Xml;

namespace EssentialWCF
{
   public class NetDataContractFormatAttribute :
➥Attribute, IOperationBehavior
   {
      public void AddBindingParameters(OperationDescription
➥description, BindingParameterCollection parameters)
      {
      }

      public void ApplyClientBehavior(OperationDescription
➥description, ClientOperation proxy)
      {
         ReplaceDataContractSerializerOperationBehavior(description);
      }

      public void ApplyDispatchBehavior(OperationDescription
➥description, DispatchOperation dispatch)
      {
         ReplaceDataContractSerializerOperationBehavior(description);
      }

      public void Validate(OperationDescription description)
      {
      }

      private static void ReplaceDataContractSerializerOperationBehavior
➥(OperationDescription description)
      {
         DataContractSerializerOperationBehavior dcs =
➥description.Behaviors.Find<DataContractSerializerOperationBehavior>();

         if (dcs != null)
               description.Behaviors.Remove(dcs);

         description.Behaviors.Add(new
               NetDataContractSerializerOperationBehavior(description));
      }

      public class NetDataContractSerializerOperationBehavior :
➥DataContractSerializerOperationBehavior
      {
         private static NetDataContractSerializer serializer = new
➥NetDataContractSerializer();

         public NetDataContractSerializerOperationBehavior(
➥OperationDescription operationDescription) : base(operationDescription) { }

         public override XmlObjectSerializer CreateSerializer(Type
➥type, string name, string ns, IList<Type> knownTypes)
         {
            return NetDataContractSerializerOperationBehavior.serializer;
         }

         public override XmlObjectSerializer CreateSerializer(Type type,
➥XmlDictionaryString name, XmlDictionaryString ns, IList<Type> knownTypes)
         {
            return NetDataContractSerializerOperationBehavior.serializer;
         }
      }
   }
}
```


Чтобы использовать `NetDataContractSerializer` в качестве сериализатора, снабдите операции атрибутом `[NetDataContractFormat]`, как показано в листинге 6.19.

Листинг 6.19. Сериализация с помощью атрибута NetDataContract

```
using System.Collections.Generic;
using System.ServiceModel;

namespace EssentialWCF
{
   [ServiceContract]
```

```
   public interface IEmployeeInformation
   {
      [OperationContract]
      [NetDataContractFormat]
      List<Employee> GetEmployees();
   }
}
```

Коль скоро к операции применен этот атрибут, WCF воспользуется классом `NetDataContractSerializer`. Если посмотреть, что выводится в сеть, то окажется, что в XML-документ включена информация о типе и дополнительная информация, необходимая для сохранения ссылок (см. листинг 6.20).

Листинг 6.20. Результат сериализации с помощью атрибута NetDataContract

```
<s:Envelope    xmlns:s="http://www.w3.org/2003/05/soap-envelope"
➥xmlns:a="http://www.w3.org/2005/08/addressing">
   <s:Header>
      <a:Action s:mustUnderstand="1">http://tempuri.org/IEmployeeInformation/
➥GetEmployeesResponse</a:Action>
      <a:RelatesTo>urn:uuid:12c35e16-52ed-4b81-a4d4-2cabd9f2c7c2</a:RelatesTo>
   </s:Header>
   <s:Body>
      <GetEmployeesResponse xmlns="http://tempuri.org/">
         <ArrayOfEmployee z:Id="1" z:Type="System.Collections.Generic.List'1
➥[[Employee, App_Code.5cwz4hgr, Version=0.0.0.0, Culture=neutral,
➥PublicKeyToken=null]]" z:Assembly="0"
➥xmlns="http://schemas.datacontract.org/2004/07/"
➥xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
➥xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
            <_items z:Id="2" z:Size="4">
               <Employee z:Id="3">
                  <EmployeeID>1</EmployeeID>
                  <FirstName z:Id="4">John</FirstName>
                  <LastName z:Id="5">Doe</LastName>
               </Employee>
               <Employee z:Id="6">
                  <EmployeeID>2</EmployeeID>
                  <FirstName z:Id="7">Jane</FirstName>
                  <LastName z:Id="8">Doe</LastName>
               </Employee>
               <Employee z:Id="9">
                  <EmployeeID>3</EmployeeID>
                  <FirstName z:Id="10">John</FirstName>
                  <LastName z:Id="11">Smith</LastName>
               </Employee>
               <Employee i:nil="true"/>
            </_items>
            <_size>3</_size>
            <_version>3</_version>
         </ArrayOfEmployee>
      </GetEmployeesResponse>
   </s:Body>
</s:Envelope>
```


Обратимая сериализация с применением интерфейса IExtensibleDataObject

Контроль над версиями контрактов о данных – важный аспект сервисно-ориентированных архитектур, изменяющихся с течением времени. Вполне возможно появление новых версий служб с обновленными контрактами о данных, в которые включена новая информация. Вместо того чтобы перекомпилировать все старые клиенты и службы, пользовавшиеся прежними версиями контрактов о данных, хотелось бы, чтобы они просто игнорировали новые данные. Именно так и ведет себя класс `DataContractSerializer`. Обнаружив новые данные, он их отбросит. Но если принятые данные позже возвращаются клиенту, то такое игнорирование будет означать потерю данных. Предположим, например, что новый клиент посылает старой службе данные, которые сохраняются в базе и впоследствии извлекаются оттуда. Если в этой ситуации клиент посылает серверу дополнительные данные, то обратно он их не получит. Именно эту проблему и призван решить интерфейс `IExtensibleDataObject`. Он позволяет работать с внешними данными, не известными контракту о данных. Делается это путем сохранения данных, не распознанных на этапе десериализации, в экземпляре класса `ExtensibleDataObject`.

По умолчанию класс `DataContractSerializer` игнорирует все неожиданные данные, если только в контракте о данных не реализован интерфейс `IExtensibleDataObject`. Ниже приведен пример двух контрактов о данных для класса `Employee`. В первом (листинг 6.21) есть три поля: `FirstName`, `LastName` и `EmployeeID`. Второй (листинг 6.22) – обновленная версия того же контракта с дополнительным полем `SSN`.

Не забывайте реализовывать интерфейс `IExtensibleDataObject`**.** Можно разделять контракты о данных и без генерации прокси-классов с помощью svcutil.exe или Add Service Reference. Для этого нужно добавить ссылку в сборку, содержащую контракты о данных. В таком случае убедитесь, что контракты о данных реализуют интерфейс `IExtensibleDataObject`, иначе они не будут поддерживать обратимую сериализацию.

Листинг 6.21. Исходный контракт Employee

```
using System.Runtime.Serialization;

[DataContract]
public class Employee
{
   private int employeeID;
   private string firstName;
   private string lastName;

   public Employee(int employeeID, string firstName, string lastName)
   {
      this.employeeID = employeeID;
      this.firstName = firstName;
```

```
      this.lastName = lastName;
   }

   [DataMember]
   public int EmployeeID
   {
      get { return employeeID; }
      set { employeeID = value; }
   }

   [DataMember]
   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }

   [DataMember]
   public string LastName
   {
      get { return lastName; }
      set { lastName = value; }
   }
}
```

В листинге 6.22 приведена новая версия контракта `Employee`, в которую добавлено поле `SSN` (номер социального страхования).

Листинг 6.22. Новый контракт Employee

```
using System.Runtime.Serialization;

[DataContract]
public class Employee
{
   private int employeeID;
   private string firstName;
   private string lastName;
   private string ssn;

   public Employee()
   {
   }

   public Employee(int employeeID, string firstName, string lastName, string ssn)
   {
      this.employeeID = employeeID;
      this.firstName = firstName;
      this.lastName = lastName;
      this.ssn = ssn;
   }

   [DataMember]
   public int EmployeeID
   {
```



```
      get { return employeeID; }
      set { employeeID = value; }
   }

   [DataMember]
   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }

   [DataMember]
   public string LastName
   {
      get { return lastName; }
      set { lastName = value; }
   }

   [DataMember]
   public string SSN
   {
      get { return ssn; }
      set { ssn = value; }
   }
}
```

Контракты в листингах 6.21 и 6.22 различны. Можно ожидать, что сервер, пользующийся исходным контрактом о данных, не станет общаться с клиентом, посылающим данные согласно новому контракту. Но на самом деле все работает. Объясняется это тем, что в новом контракте есть все поля, описанные в старом, а, значит, вся информация, необходимая серверу. Дополнительные данные сервер просто игнорирует. В этом можно убедиться, воспользовавшись службой `UpdateEmployee`, представленной в листинге 6.23. Она принимает экземпляр `Employee`, что-то с ним делает, а затем возвращает тот же самый экземпляр клиенту.

Листинг 6.23. Служба обновления данных о работнике

```
using System.ServiceModel;

namespace EssentialWCF
{
   [ServiceContract]
   public interface IEmployeeInformation
   {
      [OperationContract]
      Employee UpdateEmployee(Employee employee);
   }

   public class EmployeeInformation : IEmployeeInformation
   {
      public Employee UpdateEmployee(Employee emp)
      {
         // Сделаем вид, что выполняем какие-то действия...
```

```
         // Какие именно, для этого примера несущественно.

         // Возвращаем клиенту тот же самый экземпляр Employee
         return emp;
      }
   }
}
```

В листинге 6.24 приведен код соответствующего клиента.

Листинг 6.24. Клиент службы обновления данных о работнике

```
using System;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         Employee e = new Employee() { EmployeeID = 123456,
               FirstName = "John", LastName = "Doe",
               SSN = "000-00-0000" };
         Console.WriteLine("{0} {1}, {2}, {3}", new object[] {
                  e.FirstName,
                  e.LastName,
                  e.EmployeeID,
                  e.SSN });

         using (EmployeeInformationClient client =
                  new EmployeeInformationClient())
            e = client.UpdateEmployee(e);

         Console.WriteLine("{0} {1}, {2}, {3}", new object[] {
                  e.FirstName,
                  e.LastName,
                  e.EmployeeID,
                  e.SSN });

         Console.WriteLine("Для завершения нажмите [ENTER].");
         Console.ReadLine();
      }
   }
}
```

Возвращенный сервером результат не содержит поля `SSN`. Следовательно, из-за несовместимости версий мы не можем послать контракт данных серверу и получить его обратно неизмененным. Так как же все-таки модифицировать нашу службу, чтобы она принимала неизвестные данные и возвращала их, как положено? К счастью, WCF предоставляет такую возможность. Мы можем изменить контракт о данных на сервере, допустив возможность появления дополнительных данных, о которых ему ничего не известно. Для этого следует реализовать в контракте данных интерфейс `IExtensibleDataObject`, причем по умолчанию так и делается, если клиентский прокси-класс генерируется с помощью `svcutil.exe`

или Add Service Reference. В листинге 6.25 показан первоначальный контракт `Employee` с поддержкой интерфейса `IExtensibleDataObject`.

Листинг 6.25. Первоначальный контракт Employee с поддержкой интерфейса IExtensibleDataObject

```
using System.Runtime.Serialization;

[DataContract]
public class Employee : IExtensibleDataObject
{
   private ExtensionDataObject extensionData;

   private int employeeID;
   private string firstName;
   private string lastName;

   public Employee()
   {
   }

   public Employee(int employeeID, string firstName, string lastName)
   {
      this.employeeID = employeeID;
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public ExtensionDataObject ExtensionData
   {
      get { return extensionData; }
      set { extensionData = value; }
   }

   [DataMember]
   public int EmployeeID
   {
      get { return employeeID; }
      set { employeeID = value; }
   }

   [DataMember]
   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }

   [DataMember]
   public string LastName
   {
      get { return lastName; }
      set { lastName = value; }
   }
}
```

При таком изменении клиент получает от сервера поле SSN. Учитывая, что такое поведение считается ожидаемым в сервисно-ориентированной архитектуре, мы рекомендуем реализовывать интерфейс IExtensibleDataObject во всех контрактах о данных.

Сериализация типов с помощью суррогатов

Иногда нужно сериализовать тип, который либо не является сериализуемым вовсе, либо способ его сериализации вас не устраивает. Примером может служить тип, полученный от внешнего поставщика, или компонент, исходный код которого утрачен. В следующем примере (листинг 6.26) показан несериализуемый класс Employee. Мы намеренно не включили конструктор по умолчанию, к тому же все его поля и свойства не допускают записи. Следовательно, ни один из рассмотренных выше способов сериализации к нему не применим. Чтобы все-таки справиться с задачей, нам потребуется суррогат, который сериализует этот класс от своего имени.

Листинг 6.26. Несериализуемый класс Employee

```
namespace EssentialWCF.Serialization.Surrogate
{
   public class Employee
   {
      private int employeeID;
      private string firstName;
      private string lastName;

      public Employee(int employeeID, string firstName,
                              string lastName)
      {
         this.employeeID = employeeID;
         this.firstName = firstName;
         this.lastName = lastName;
      }

      public int EmployeeID
      {
         get { return employeeID; }
      }

      public string FirstName
      {
         get { return firstName; }
      }

      public string LastName
      {
         get { return lastName; }
      }
   }
}
```


Для разработки суррогата потребуется выполнить два шага. Сначала определяется контракт о данных, который представляет сериализованный тип. Затем пишется суррогатный контракт о данных, реализующий интерфейс IDataContractSurrogate. Мы рассмотрим три его главных метода: GetDataContractType, GetDeserializedObject и GetObjectToSerialize. GetDataContractType возвращает сериализованный тип объекту DataContractSerializer, а GetDeserializedObject и GetObjectToSerialize выполняют десериализацию и сериализацию соответственно. Класс EmployeeSurrogate приведен в листинге 6.27.

Листинг 6.27. Класс EmployeeSurrogate

```
using System;
using System.CodeDom;
using System.Collections.ObjectModel;
using System.Runtime.Serialization;

namespace EssentialWCF.Serialization.Surrogate
{
   [DataContract]
   internal class EmployeeSurrogated
   {
      [DataMember]
      private int employeeID;
      [DataMember]
      private string firstName;
      [DataMember]
      private string lastName;

      public EmployeeSurrogated(int employeeID, string firstName,
                  string lastName)
      {
         this.employeeID = employeeID;
         this.firstName = firstName;
         this.lastName = lastName;
      }

      public int EmployeeID
      {
         get { return employeeID; }
      }

      public string FirstName
      {
         get { return firstName; }
      }

      public string LastName
      {
         get { return lastName; }
      }
   }

   public class EmployeeSurrogate : IDataContractSurrogate
```

```
{
   public object GetCustomDataToExport(Type clrType,
                Type dataContractType)
   {
      return null; // не реализован
   }

   public object GetCustomDataToExport(
               System.Reflection.MemberInfo memberInfo,
               Type dataContractType)
   {
      return null; // не реализован
   }

   public Type GetDataContractType(Type type)
   {
      if (typeof(Employee).IsAssignableFrom(type))
      {
         return typeof(EmployeeSurrogated);
      }
      return type;
   }

   public object GetDeserializedObject(object obj, Type targetType)
   {
      if (obj is EmployeeSurrogated)
      {
         EmployeeSurrogated oldEmployee = (EmployeeSurrogated)obj;
         Employee newEmployee =
                        new Employee(oldEmployee.EmployeeID,
                                 oldEmployee.FirstName,
                                 oldEmployee.LastName);
         return newEmployee;
      }
      return obj;
   }

   public void GetKnownCustomDataTypes(
               Collection<Type> customDataTypes)
   {
      throw new NotImplementedException();
   }

   public object GetObjectToSerialize(object obj, Type targetType)
   {
      if (obj is Employee)
      {
         Employee oldEmployee = (Employee)obj;
         EmployeeSurrogated newEmployee =
                        new EmployeeSurrogated(oldEmployee.EmployeeID,
                                 oldEmployee.FirstName,
                                 oldEmployee.LastName);
         return newEmployee;
      }
      return obj;
```



```
      }

      public Type GetReferencedTypeOnImport(string typeName,
                  string typeNamespace, object customData)
      {
         if (typeNamespace.Equals("http://schemas.datacontract.org/2004/07/
➥EmployeeSurrogated"))
         {
            if (typeName.Equals("EmployeeSurrogated"))
                  return typeof(Employee);
         }
         return null;
      }

      public CodeTypeDeclaration ProcessImportedType(CodeTypeDeclaration
➥typeDeclaration,
         CodeCompileUnit compileUnit)
      {
         return typeDeclaration;
      }
   }
}
```

Теперь соберем все вместе, известив `DataContractSerializer` о наличии суррогатного класса. Необходимо создать экземпляр `DataContractSerializer` и передать его конструктору объект типа `EmployeeSurrogate`, как показано в листинге 6.28.

Листинг 6.28. Использование класса `EmployeeSurrogate` совместно с `DataContractSerializer`

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Xml;

namespace EssentialWCF.Serialization.Surrogate
{
   class Program
   {
      static void TryToSerialize(Employee e)
      {
         DataContractSerializer dcs =
                  new DataContractSerializer(typeof(Employee));

         using (StringWriter sw = new StringWriter())
         {
            using (XmlWriter xw = XmlWriter.Create(sw))
            {
               try
               {
                  dcs.WriteObject(xw, e);
               }
               catch (InvalidDataContractException)
```

```
            {
               Console.WriteLine("Не могу сериализовать без суррогата!);
            }
         }
      }
   }

   static string SerializeUsingSurrogate(
               DataContractSerializer dcs, Employee e)
   {
      using (StringWriter sw = new StringWriter())
      {
         using (XmlWriter xw = XmlWriter.Create(sw))
         {
            dcs.WriteObject(xw, e);
            xw.Flush();
            return sw.ToString();
         }
      }
   }

   static Employee DeserializeUsingSurrogate(
               DataContractSerializer dcs, string employeeAsString)
   {
      using (StringReader tr = new StringReader(employeeAsString))
         using (XmlReader xr = XmlReader.Create(tr))
            return dcs.ReadObject(xr) as Employee;
   }

   static void Main(string[] args)
   {
      Employee e = new Employee(12345, "John", "Doe");

      TryToSerialize(e);

      DataContractSerializer dcs =
               new DataContractSerializer(typeof(Employee),
               null, int.MaxValue, false, false,
               new EmployeeSurrogate());

      string employeeAsString = SerializeUsingSurrogate(dcs, e);

      e = DeserializeUsingSurrogate(dcs, employeeAsString);

      Console.ReadLine();
   }
}
}
```

Потоковая отправка объемных данных

WCF поддерживает два режима обработки сообщений: *буферизованный* и *потоковый*. В буферизованном режиме, который принимается по умолчанию, сообщение целиком хранится в памяти до момента отправки или после получения.

В большинстве случаев этого режима достаточно, а иногда, например для надежной доставки или добавления/проверки цифровой подписи, даже необходимо. Однако буферизация объемных сообщений может истощить системные ресурсы и ограничить масштабируемость. Поэтому WCF поддерживает также потоковую обработку сообщений. В этом режиме данные отправляются и принимаются с помощью класса `System.IO.Stream`. Обычно потоковый режим включается путем настройки привязки или транспортного канала. В листинге 6.29 показано, как включить этот режим для настройки `netTcpBinding`, задав в конфигурационном файле атрибут `transferMode`, который может принимать значения `Buffer`, `Streamed`, `StreamedResponse` и `StreamedRequest`. Такой набор значений позволяет точно управлять потоком сообщений между клиентом и сервером.

Листинг 6.29. Включение потокового режима для привязки netTcpBinding

```
<system.serviceModel>
   <services>
      <service name="EssentialWCF.FileDownload">
         <endpoint address=""
                   binding="netTcpBinding"
                   bindingConfiguration="EnableStreamingOnNetTcp"
                   contract="EssentialWCF.IFileDownload" />
      </service>
   </services>
   <bindings>
      <netTcpBinding>
         <binding name="EnableStreamingOnNetTcp" transferMode="Streamed" />
      </netTcpBinding>
   </bindings>
</system.serviceModel>
```

Чтобы извлечь преимущества из потокового режима, необходимо в контракте об операции воспользоваться экземпляром класса `System.IO.Stream` или вернуть контракт о сообщении, в котором используется поток. В листинге 6.30 приведен пример контракта о службе загрузки файлов, которая возвращает `System.IO.Stream`.

Листинг 6.30. Контракт о службе FileDownload

```
using System.IO;
using System.ServiceModel;

namespace EssentialWCF
{
   [ServiceContract]
   public interface IFileDownload
   {
      [OperationContract]
      Stream GetFile(string fileName);
   }
}
```

Потоковый режим применим не во всех сценариях, где фигурируют большие объемы данных. Например, если необходима надежная доставка сообщений, цифровая подпись или возможность возобновления после сбоя, то потоки не годятся. В таких случаях лучше вручную разбивать данные на мелкие сообщения, из которых получатель самостоятельно собирает исходное. Подобный подход легко реализовать поверх WCF.

Использование класса XmlSerializer для нестандартной сериализации

Использование `DataContractSerializer` – предпочтительный механизм сериализации в WCF. Но иногда возникает желание выйти за пределы стандартной сериализации. Один из таких способов дает класс `XmlSerializer`. Есть много причин прибегнуть к его услугам: возможность реализовать нестандартную сериализацию, обобществление типов или поддержка унаследованных Web-служб. Как и `DataContractSerializer`, класс `XmlSerializer` является неотъемлемой частью WCF. В этом разделе мы рассмотрим этот класс и обсудим, как применить его для формирования результирующего XML-документа.

Класс `DataContractSerializer` всегда сериализует данные, применяя элементы, а не атрибуты XML. В листинге 6.31 показано, во что `DataContractSerializer` сериализует класс `Employee`.

Листинг 6.31. Результат сериализации объекта Employee с помощью класса DataContractSerializer

```
<Employee xmlns="http://schemas.datacontract.org/2004/07/TestSerialization"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
   <EmployeeID>101</EmployeeID>
   <FirstName>John</FirstName>
   <LastName>Doe</LastName>
</Employee>
```

Легко видеть, что контракт о данных можно было бы переписать с использованием атрибутов XML, например, так:

```
<Employee EmployeeID="101" FirstName="John" LastName="Doe" />
```

Но `DataContractSerializer` не умеет сериализовывать с помощью атрибутов. Единственное, что `DataContractSerializer` позволяет настраивать, – это имена элементов XML, задаваемые в атрибуте `[DataMember]`. Сериализатор `NetDataContractSerializer` по существу мало чем отличается от `DataContractSerializer`, разве что поддерживает обобществление информации о типе. Таким образом, `XmlSerializer` оказывается единственным инструментом, который позволяет вам полностью взять контроль над сериализацией в свои руки. В листинге 6.32 приведена схема класса `Employee` с использованием атрибутов XML.

Листинг 6.32. XSD-схема класса Employee

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://tempuri.org/XMLSchema.xsd"
elementFormDefault="qualified"
xmlns="http://tempuri.org/XMLSchema.xsd"
xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Employee">
      <xs:complexType>
         <xs:attribute name="EmployeeID" type="xs:int" />
         <xs:attribute name="FirstName" type="xs:string" />
         <xs:attribute name="LastName" type="xs:string" />
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Нестандартная сериализация с применением атрибутов

Использовать класс `XmlSerializer` для формирования результирующего XML-документа можно двумя способами. Самый прямолинейный подход состоит в том, чтобы применить предоставляемые каркасом .NET Framework готовые атрибуты, которые находятся в пространстве имен `System.Xml.Serialization`. По умолчанию `XmlSerializer` выводит открытые поля и свойства, допускающие чтение и запись, в виде элементов XML. Однако наличие атрибута `[XmlAttribute]` заставляет `XmlSerializer` представлять их в виде атрибутов. Кроме того, `XmlSerializer` по умолчанию сериализует все открытые поля и свойства, допускающие чтение и запись, для которых явно не задан атрибут `[XmlIgnore]`. Есть и другие атрибуты, например `[XmlElement]`, `[XmlRoot]`, `[XmlArray]` и `[XmlArrayItem]`, которые модифицируют алгоритм сериализации типов.

Нестандартная сериализация с применением интерфейса IXmlSerializable

Есть и другой подход к работе с `XmlSerializer` – воспользоваться интерфейсом `IXmlSerializable`. Обычно он применяется в более сложных случаях, когда необходим полный контроль над сериализацией. В интерфейсе `IXmlSerializable` определены три метода: `GetSchema`, `ReadXml` и `WriteXml`. Начиная с версии .NET 2.0, метод `GetSchema` объявлен устаревшим и заменен атрибутом `[XmlSchemaProvider]`. Методы `ReadXml` и `WriteXml` применяются соответственно для десериализации и сериализации класса. В листинге 6.33 приведен пример.

Листинг 6.33. Сериализация класса Employee в формат XML

```
using System.IO;
using System.Xml;
```

```
using System.Xml.Schema;
using System.Xml.Serialization;

[XmlSchemaProvider("MySchema")]
public class Employee : IXmlSerializable
{
   private const string ns = "http://essentialwcf/xmlserialization/";

   private int employeeID;
   private string firstName;
   private string lastName;

   public Employee()
   {
   }

   public Employee(int employeeID, string firstName, string lastName)
   {
      this.employeeID = employeeID;
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public int EmployeeID
   {
      get { return employeeID; }
      set { employeeID = value; }
   }

   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }

   public string LastName
   {
      get { return lastName; }
      set { lastName = value; }
   }

   public static XmlQualifiedName MySchema(XmlSchemaSet schemaSet)
   {
      XmlSchema shema = XmlSchema.Read(new StringReader(
            @"<xs:schema elementFormDefault=""qualified""" +
            @" xmlns:tns="" + namespace + """ +
            @" targetNamespace="" + namespace + """ +
            @" xmlns=""http://www.w3.org/2001/XMLSchema""" +
            @" xmlns:xs=""http://www.w3.org/2001/XMLSchema"">" +
            @" <xs:element name=""Employee"">" + @"<xs:complexType>" +
            @"<xs:attribute name=""EmployeeID"" type=""xs:int"" />" +
            @"<xs:attribute name=""FirstName"" type=""xs:string"" />" +
            @"<xs:attribute name=""LastName"" type=""xs:string"" />" +
            @"</xs:complexType>" +
            @" </xs:element>" +
            @"</xs:schema>"), null);
      schemaSet.XmlResolver = new XmlUrlResolver();
      schemaSet.Add(shema);

      return new XmlQualifiedName("Employee", ns);
   }

   public XmlSchema GetSchema()
   {
      return null;
   }

   public void ReadXml(XmlReader reader)
   {
      reader.ReadStartElement("Employee");
      reader.MoveToAttribute("ID");
      this.employeeID = reader.ReadContentAsInt();
      reader.MoveToAttribute("FirstName");
      this.firstName = reader.ReadContentAsString();
      reader.MoveToAttribute("LastName");
      this.LastName = reader.ReadContentAsString();
      reader.ReadEndElement();
   }

   public void WriteXml(XmlWriter writer)
   {
      writer.WriteStartElement("Employee");

      writer.WriteStartAttribute("ID");
      writer.WriteString(this.employeeID.ToString());
      writer.WriteEndAttribute();

      writer.WriteStartAttribute("FirstName");
      writer.WriteString(this.firstName);
      writer.WriteEndAttribute();

      writer.WriteStartAttribute("LastName");
      writer.WriteString(this.lastName);
      writer.WriteEndAttribute();

      writer.WriteEndElement();
   }
}
```


Класс `XmlSerializer` позволяет использовать XSD-схемы в качестве отправной точки для составления контрактов. Недостаток в том, что приходится писать довольно много кода.

Выбор кодировщика

В начале этой главы мы говорили о двух шагах подготовки объекта к передаче по сети. Первый шаг – сериализация, в ходе которой граф объекта преобразуется в информационный набор XML. Второй – кодирование, когда информационный набор XML трансформируется в массив байтов, передаваемых по сети. WCF под-

держивает три способа кодирования: текстовое, двоичное и MTOM. В этом разделе мы поговорим о том, в каких случаях использовать тот или иной кодировщик.

Текстовое и двоичное кодирование

До появления WCF у вас был широкий выбор технологий построения распределенных приложений и, в частности, .NET Remoting и Web-службы ASP.NET. Технология .NET Remoting прекрасно подходила для коммуникации между .NET-приложениями, поскольку оптимизировала передачу данных, применяя двоичное кодирование. Производительность оказывалась выше, чем для Web-служб ASP.NET, которым ради интероперабельности приходилось ограничиваться текстовым кодированием. WCF абстрагирует механизм кодирования и позволяет задавать в привязке любой из двух способов. Тем самым функциональность, предлагаемая WCF, заменяет и .NET Remoting, и Web-службы ASP.NET.

Вы не работаете с кодированием напрямую, а задаете нужный режим в привязке, с помощью которой раскрывается служба. В главе 4 были описаны как привязки, применяемые только для коммуникации между .NET-приложениями, так и интероперабельные привязки. К первому типу относится привязка `netTcpBinding`, в которой используется кодировщик `binaryMessageEncoding`; он обеспечивает наивысшую производительность, но ценой утраты интероперабельности. Напротив, в привязке `wsHttpBinding` используется кодировщик `textMessageEncoding`, поддерживающий интероперабельность в соответствии со спецификациями WS-*. В листинге 6.34 приведен пример заказной привязки с кодировщиком `textMessageEncoding`.

Листинг 6.34. Заказная привязка с кодировщиком textMessageEncoding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <customBinding>
            <binding name="Custom">
               <textMessageEncoding />
               <httpTransport />
            </binding>
         </customBinding>
      </bindings>
      <client>
         <endpoint
            address="http://localhost/SerializationExample/Service.svc"
            binding="customBinding"
            bindingConfiguration="Custom"
            contract="Serialization.localhost.IEmployeeInformation"
            name="IEmployeeInformation" />
      </client>
   </system.serviceModel>
</configuration>
```


В листинге 6.35 показана конфигурация заказной привязки с кодировщиком `binaryMessageEncoding`.

Листинг 6.35. Заказная привязка с кодировщиком binaryMessageEncoding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <customBinding>
            <binding name="Custom">
               <binaryMessageEncoding />
               <httpTransport />
            </binding>
         </customBinding>
      </bindings>
      <client>
         <endpoint
            address="http://localhost/SerializationExample/Service.svc"
            binding="customBinding"
            bindingConfiguration="Custom"
            contract="Serialization.localhost.IEmployeeInformation"
            name="IEmployeeInformation" />
      </client>
   </system.serviceModel>
</configuration>
```

Отправка двоичных данных в кодировке MTOM

Кодировщик `textMessageEncoding` преобразует сообщения в текстовый формат XML. С точки зрения интероперабельности это замечательно, но для передачи больших объемов двоичных данных неэффективно. Для этой цели в ситуациях, требующих интероперабельности, можно применять схему кодирования MTOM (Message Transmission Optimization Mechanism – механизм оптимизации передачи сообщений). Это стандартный способ передачи двоичных данных в виде приложений к сообщению SOAP. Иными словами, двоичные данные можно передавать прямо в сообщении SOAP без издержек, сопутствующих Base64-кодированию. Чтобы воспользоваться преимуществами MTOM, служба должна содержать в контракте об операции байтовый массив или объект `Stream`.

WCF поддерживает MTOM с помощью кодировщика `mtomMessageEncoding`. Как правило, этот кодировщик заказывается в привязке. В листинге 6.36 показано, как задать кодировщик MTOM в привязке `wsHttpBinding`.

Листинг 6.36. Привязка wsHttpBinding с кодировщиком mtomMessageEncoding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
```

```
        <wsHttpBinding>
           <binding name="MTOMBinding" messageEncoding="Mtom" />
        </wsHttpBinding>
      </bindings>
      <client>
         <endpoint
               address="http://localhost/SerializationExample/Service.svc"
               binding="wsHttpBinding"
               bindingConfiguration="MTOMBinding"
               contract="EssentialWCF.IEmployeePicture"
               name="IEmployeePicture" />
      </client>
   </system.serviceModel>
</configuration>
```

Знакомство с кодировщиком WebMessageEncoder

Кодировщик `WebMessageEncoder` включен в WCF, начиная с версии .NET Framework 3.5. Он поддерживает кодирование в форматах JSON и POX. Этот класс не предлагает какого-то особого вида кодирования, а просто агрегирует стили кодирования, распространенные в современных Web-приложениях. Включается `WebMessageEncoder`, если использовать одно из двух поведений оконечной точки: `WebHttpBehavior` или `WebScriptEnablingBehavior`.

Поведение `WebHttpBehavior` говорит классу `WebMessageEncoder`, что он должен пользоваться кодировщиком `TextMessageEncoder`. В результате создается объект типа `TextMessageEncoder`, а версия сообщения для него устанавливается в `MessageVersion.None`. Это служит для `TextMessageEncoder` указанием не включать в XML-документ информацию, относящуюся к протоколам SOAP или WS-Addressing. Кодировщик `WebMessageEncoder` поддерживает запросы и ответы в форматах XML или JSON. По умолчанию поведение оконечной точки `WebHttpBehavior` настроено так, что для запросов и ответов принимается формат `WebMessageFormat.Xml`. Для контроля над форматом сообщений предназначен атрибут `[WebGet]`, который может принимать значения `WebMessageFormat.Xml` или `WebMessageFormat.Json`.

Хотя форматы запроса и ответа можно задавать независимо, обычно указывается одно и то же значение. Поведение оконечной точки `WebScriptEnablingBehavior` именно так и устроено: формат запроса и ответа устанавливается в `WebMessageFormat.Json`. Это говорит кодировщику `WebMessageEncoder` о том, что для кодирования сообщений следует применять класс `JsonMessageEncoder`. Поведение `WebScriptEnablingBehavior` применяется для Web-приложений на основе технологии AJAX, которые вызывают службу из JavaScript-сценария. Попутно оно предоставляет поддержку для клиентских прокси-классов, которые создает система ASP.NET AJAX.

Дополнительную информацию о поведениях `WebHttpBehavior` и `WebScriptEnablingBehavior`, а также об атрибуте `[WebGet]` см. в главе 13.

Резюме

В этой главе были описаны средства сериализации и кодирования, имеющиеся в WCF. Как и в других частях WCF, существует много возможностей расширить механизм сериализации и настроить его под свои нужды. Ниже сформулированы некоторые общие принципы работы с сериализацией:

- всюду, где возможно, старайтесь применять сериализацию `DataContract`. Она применяется в WCF по умолчанию и предназначена для использования в сервисно-ориентированных архитектурах, когда контракт раскрывается явно;
- существует много ситуаций, когда приходится прибегать к классу `XMLSerializer`, например: поддержка существующих типов .NET, совместимость с Web-службами, разработанными в ASP.NET, контроль над формой сериализованного XML-документа. В таком случае помещайте в соответствующие места контрактов атрибут `[XmlSerializerFormat]`. Не забывайте помечать им контракт о службе, если в сериализаторе `XMLSerializer` нуждаются все вообще операции;
- для всех готовых привязок задан кодировщик по умолчанию. Если у вас возникнет мысль изменить принятый по умолчанию кодировщик, то лучше поищите привязку, которая лучше отвечала бы потребностям вашего приложения;
- при создании заказных привязок поинтересуйтесь, какой кодировщик поддерживается по умолчанию транспортным протоколом. Если вы не зададите кодировщик явно, то именно умалчиваемый и будет использоваться;
- если данные настолько объемны, что держать их целиком в памяти нецелесообразно, прибегайте к потоковой передаче, поддерживаемой WCF. Если передавать данные потоком по какой-либо причине невозможно, разбивайте их на более мелкие сообщения.

Глава 7. Размещение

Владельцем службы называется процесс операционной системы, отвечающий за время ее жизни и необходимый контекст. Он запускает и останавливает службу, а также предоставляет базовые функции управления. Помимо этого, владелец почти ничего не знает о WCF-службе, которая работает в его адресном пространстве.

Владельцем службы может быть любой процесс операционной системы. IIS и Windows Process Activation Services (WAS) обладают встроенной инфраструктурой, облегчающей размещение в них служб. Совместно с ASP.NET они часто используются в качестве среды для размещения служб. Помимо IIS и WAS, WCF-служба может размещаться в составе обычной управляемой службы Windows (NT-службы), которую запускает и останавливает сама операционная система. Владельцем службы может быть также любое приложение Windows, представленное полноценным окном или значком в системном лотке, и даже консольное приложение. Где бы ни размещалась служба, способы конфигурирования ее адреса, привязки, контракта и поведений практически не изменяются.

При выборе варианта размещения службы следует руководствоваться эксплуатационными требованиями: доступностью, надежностью, управляемостью и т.п. Например, при размещении в составе NT-службы вы получаете удобный интерфейс запуска и останова, знакомый большинству системных администраторов Windows. С другой стороны, размещение в Windows-приложении на рабочем столе привычно большинству конечных пользователей. Модель программирования WCF от размещения не зависит. При проектировании и реализации службы разработчик может не заботиться не только о ее адресе и привязке, но и о том, кто станет ее владельцем.

Любой владелец обязан уметь выполнять три вещи: создать экземпляр класса `ServiceHost` из пространства имен `System.ServiceModel`, добавить оконечные точки и начать прослушивание канала в ожидании сообщений. Если вы хотите, чтобы ваша программа могла стать владельцем службы, то должны структурировать ее, как показано в листинге. Если служба размещается в IIS или WAS, то такая инфраструктура предоставляется автоматически.

Листинг 7.1. Базовая структура размещения службы

```
ServiceHost myHost = new ServiceHost( typeof( MyService ) );
myHost.AddServiceEndpoint (typeof( IMyService ),
                        someBinding,
                        someUri);
myHost.Open();
```


В этой главе мы рассмотрим типичные способы размещения службы. В случае IIS все это довольно просто и описано в главе 1. Но мы поговорим о деталях того, как новая система Windows Process Activation Service (WAS) соотносится с инфраструктурой IIS. Мы также уделим внимание авторазмещению в составе NT-служб и клиентских приложений.

Размещение службы в Windows Process Activation Services

Windows Process Activation Services (WAS) – это инфраструктура размещения, встроенная в Vista и Windows Server 2008. Средства, которые раньше были доступны только в IIS – активация процесса, рециркуляция и управление идентичностью, теперь перенесены в WAS и предоставлены в распоряжение протоколов, отличных от HTTP.

WAS позволяет размещать службы в надежном окружении, которое не связано жестко с протоколом HTTP. Хотя этот протокол широко распространен и все его понимают, бывают случаи, когда он не подходит. Представьте, например, службу, которая получает односторонние сообщения для целей слежения и анализа, причем эти сообщения посылают клиенты, которые иногда отключаются от сети. Чтобы обеспечить возможность отправки сообщений в условиях отсутствия связи, необходим механизм очередей. Протокол MSMQ такой механизм обеспечивает, а HTTP нет. Или возьмем очень «болтливую» службу, которая быстро посылает много мелких сообщений в составе объемлющего диалога. В таком случае протокол TCP более эффективен, чем HTTP, поскольку не закрывает соединение в течение всего времени отправки многочисленных сообщений. Это примеры того, как в WAS можно разместить службу, непригодную для размещения в IIS.

WAS поддерживает различные протоколы с помощью архитектуры *адаптера прослушивателя*, когда *прослушиватель* абстрагируется от функции управления процессом. Определив интерфейс с прослушивателем, WAS может поддержать много разных прослушивателей, не усложняя систему. Таким образом, для коммуникации по протоколам HTTP, TCP, MSMQ и именованным каналам применяется единый механизм, что повышает общую надежность системы. Архитектура WAS изображена на рис. 7.1.

Служба WAS автоматически устанавливается в Vista и Windows Server 2008 вместе с IIS, поскольку IIS от нее зависит. При установке IIS w3svc регистрируется в качестве адаптера прослушивателя HTTP. При установке .NET 3.5 регистрируются адаптеры прослушивателей для TCP, MSMQ и именованных каналов. Можно установить WAS и без IIS. Для этого нужно активировать две функции Windows. Во-первых, Windows Process Activation Services, как показано на рис. 7.2. Чтобы добраться этого окна, следует нажать кнопку **Start** (Пуск), а затем последовательно выбрать **Control Panel** (Панель управления), **Programs** (Программы), **Turn Windows Features On or Off** (Включение и выключение функций Windows).

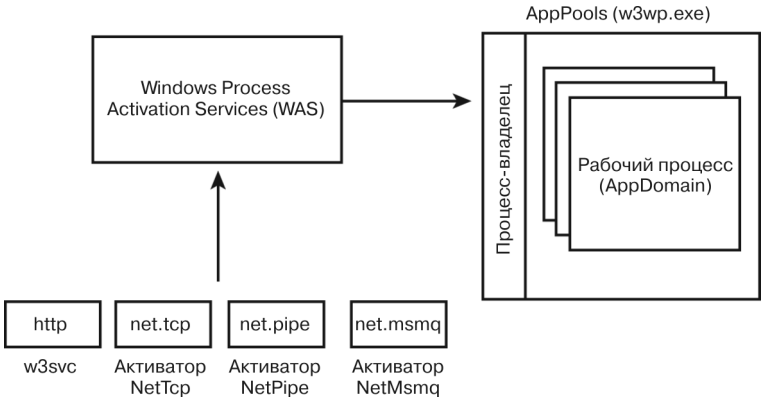

Рис. 7.1. Архитектура WAS

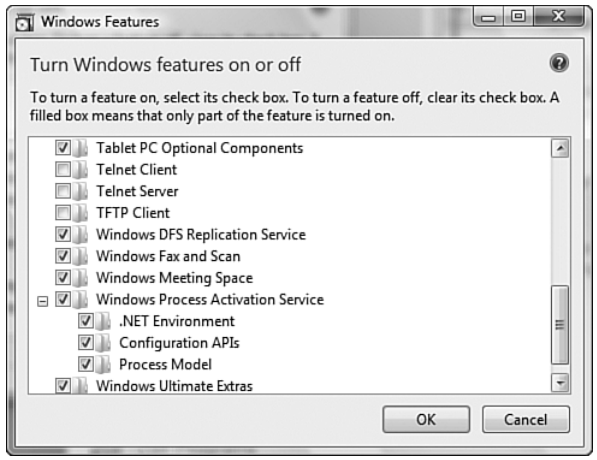

Рис. 7.2. Активация Windows Process Activation Services

После того как WAS активирована, необходимо отметить флажок WCF Non-HTTP Activation (Активация WCF для протоколов, отличных от HTTP), как показано на рис. 7.3. Если необходимо активировать HTTP для WCF-служб, то необходимо также отметить флажок WCF HTTP Activation, при этом автоматически будут включены необходимые функции IIS7.

Размещение службы внутри WAS аналогично размещению внутри IIS, описанному в главе 1. Вам понадобится виртуальное приложение, SVC-файл и/или некоторые элементы в секции `<system.serviceModel>` файла `web.config`. Чтобы разрешить использование протоколов, отличных от HTTP, необходимо выполнить следующие два шага.

Во-первых, добавьте информацию о привязке к протоколу в описание соответствующего Web-сайта в конфигурационном файле WAS. Например, для TCP

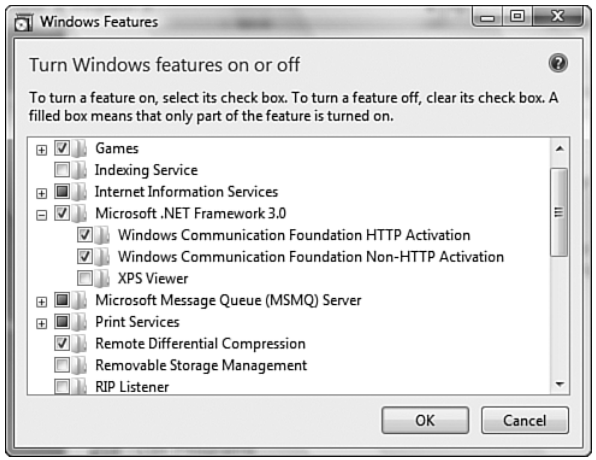

Рис. 7.3. Активация WCF для протоколов, отличных от HTTP

нужно задать конкретный порт. По умолчанию в привязке для `net.tcp` подразумевается 808:*, то есть выбирается порт 808, и прослушиватель готов принимать сообщения, отправленные из любого порта. Затем нужно изменить виртуальное приложение, разрешив альтернативный протокол. Обе эти настройки прописаны в конфигурационном файле `ApplicationHost`, который находится в папке %windir\System32\inetsrv\config, и могут быть изменены с помощью утилиты `appcmd.exe` в папке %windir\System32\inetsrv\. В листинге 7.2 показаны команды, которые осуществляют оба изменения. Предполагается, что виртуальное приложение называется WASHosted и определено внутри сайта Default Web Site.

Листинг 7.2. Разрешение протокола net.tcp для виртуального приложения

```
appcmd.exe set site "Default Web Site" –
             +bindings.[protocol='net.tcp',bindingInformation='808:*']
appcmd.exe set app "Default Web Site/WASHosted"
             /enabledProtocols:http,net.tcp
```

Протокол можно не только добавить, но и удалить. Например, следующая команда исключает HTTP из списка протоколов, разрешенных для того же приложения.

Листинг 7.3. Запрет протокола HTTP для виртуального приложения

```
appcmd.exe set app "Default Web Site/WASHosted"
                        /enabledProtocols:net.tcp
```

Во-вторых, включите в файл `web.config` привязку для любого из поддерживаемых WCF транспортных протоколов, в том числе: TCP, MSMQ и именованные каналы. В листинге 7.4 показан файл `web.config`, в котором сконфигурирована привязка к протоколу TCP.

Листинг 7.4. Конфигурация службы, размещенной внутри WAS

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>
      </services>
         <service name="EssentialWCF.StockService" >
            <endpoint address=""
                      binding="netTcpBinding"
                      contract="EssentialWCF.IStockService"/>
         </service>
      </services>
   </system.serviceModel>

</configuration>
```

Размещение службы в IIS 7

В сервере IIS 6, входящем в комплект поставки Windows 2003 и Windows XP SP2, появились пулы приложений – контейнеры для размещенных в них приложений. С их помощью был реализован контроль над запуском и остановом, управление идентичностью и рециркуляция на уровне отдельного приложения. По существу это означало изоляцию приложений и, стало быть, повышение надежности. Архитектура пула приложений обеспечивала все управление процессами.

В версии IIS 7, которая входит в состав Windows Vista и Windows Server 2008, управление процессами было обобщено на различные протоколы и перенесено в WAS. Система ASP.NET также модифицирована для поддержки активации процессов и размещения служб в WAS.

На рис. 7.4 изображено место IIS 7 в архитектуре WAS.

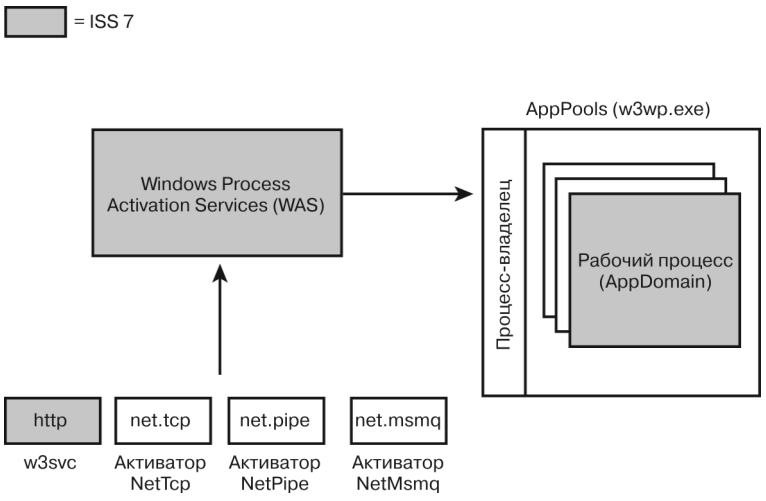

Рис. 7.4. Реализация IIS на базе WAS

Чтобы разместить службу в IIS 7, необходимо выполнить как минимум три шага, описанных в главе 1. Напомним, что вы должны создать виртуальное приложение IIS, подготовить SVC-файл, в котором определено, где находится реализация службы, и включить секцию `<system.serviceModel>` в файл `web.config`.

Виртуальное приложение в IIS – это комбинация Web-сайта, прослушивателя протокола и активации процесса. Web-сайт – это виртуальный каталог, в котором хранятся файлы. Роль прослушивателя в IIS играет процесс w3svc, который пользуется драйвером `http.sys` для сетевого ввода/вывода. Механизм активации процесса управляет подготовкой исполняющей среды для программы и определен как пул приложений AppPool внутри IIS.

Определив виртуальное приложение, вы должны поместить в него SVC-файл и файл `web.config`. SVC-файл содержит ссылку на реализацию службы, а в `web.config` определены адрес, привязка и контракт для оконечных точек и поведения службы.

Реализация службы, указанной в SVC-файле, может находиться в трех местах: в самом SVC-файле, в папке /bin виртуального каталога или в глобальном кэше сборок (GAC) на данной машине. По своему назначению SVC-файл аналогичен ASMX-файлу в IIS 6.

В файле `web.config` определены АПК службы и ее оконечных точек, то есть адрес, привязка и контракт. Поскольку служба размещена в IIS, а IIS знает только о транспорте HTTP (в отличие от TCP или MSMQ), то для оконечных точек в `web.config` можно использовать лишь привязки, в которых в качестве транспорта определен HTTP. Среди готовых привязок таких три: `basicHttpBinding`, `wsHttpBinding` и `wsDualHttpBinding`. При попытке определить для оконечной точки привязку к какому-нибудь другому протоколу, например TCP или MSMQ (скажем, `netTcpBinding`), при первой же активации службы возникнет ошибка. В привязке следует указывать относительный адрес, так как базовый адрес службы определен привязкой к протоколу и виртуальным путем к SVC-файлу.

Посмотрим, что происходит, когда виртуальное приложение создается и когда оно получает первый запрос, а также как обрабатываются последующие запросы.

При создании виртуального приложения в программе IIS Manager ассоциированный с ним URL регистрируется в IIS (w3svc). Начиная с этого момента, все запросы, полученные адаптером прослушивателя протокола HTTP, направляются приложению для обработки. В роли адаптера прослушивателя протокола HTTP выступает системный драйвер HTTP.SYS.

Когда прослушиватель протокола получает первый запрос к SVC-файлу, IIS вызывает WAS, требуя запустить рабочий процесс `w3wp.exe`, если он еще не запущен. Рабочий процесс ассоциируется с пулом `AppPool` для данного виртуального приложения. Менеджер приложений ASP.NET, работающий внутри процесса-исполнителя, получает запрос от IIS/WAS и загружает модули и обработчики, относящиеся к WCF. Слой WCF, отвечающий за размещение, просматривает секцию `<serviceModel>` в файле `web.config` и с помощью класса `ServiceHostFactory` создает объект `ServiceHost` для класса, указанного в элементе `<Service>`. Затем

в ServiceHost добавляются оконечные точки, определенные в секции <service>. Наконец, вызывается метод ServiceHost.Open, чтобы служба начала прослушивать входящие запросы. Когда служба запускается, она регистрирует свои оконечные точки в прослушивателе протокола, чтобы последующие запросы прослушиватель направлял ей уже непосредственно.

Включение функций ASMX в службе, размещенной в IIS

До появления WCF Web-службы в ASP.NET создавались в виде ASMX-файлов. Этот подход прекрасно поддерживал стандартные требования, предъявляемые к Web-службам, и обеспечивал надежность и расширяемость средствами HTTP-конвейера, лежащего в основе ASP.NET. Но в WCF службы по определению не зависят от транспортного протокола и ничего не знают о том, где размещены. Поэтому WCF-службы не могут зависеть от реализации HTTP-конвейера и, в частности, от наличия драйвера HTTP.SYS.

Подобно ASMX, технология WCF также предлагает надежную модель расширяемости. Но в ее основе лежит не HTTP-конвейер, а стек каналов. Каналы в WCF обладают большой гибкостью. Они знают не только о транспорте, подобном HTTP, но и о других элементах протокола, например, о безопасности и транзакциях. Стеки каналов описаны в главах 3 и 4.

WCF поддерживает специальную модель для размещения внутри IIS: режим совместимости с **ASP.NET**. При работе в этом режиме среду для WCF-служб предоставляет ASP.NET. Поэтому учитываются параметры, заданные в секциях <system.web/hostingEnvironment> и <system.web/compilation>. Однако в режиме совместимости работают не все средства ASP.NET, относящиеся к HTTP:

- HttpContext.Current. Оказывается равен null. В WCF-службе для тех же целей можно воспользоваться объектом OperationContext.Current;
- **авторизация файла или URL**;
- **олицетворение**;
- **состояние сеанса**;
- <system.web/Globalization>;
- ConfigurationManager.AppSettings. Можно получить настройки, заданные в web.config на уровне корня виртуального приложения или выше, поскольку httpContext равно null.

Чтобы включить в режиме совместимости средства ASP.NET, доступные ASMX-службам, необходимо изменить два параметра. На уровне приложения присвойте атрибуту <aspNetCompatibilityEnabled> элемента <serviceHostingEnvironment> в секции <system.serviceModel> значение true. А на уровне службы задайте для свойства AspNetCompatibilityRequirements значение Allowed. Если это проделать, то почти все средства ASP.NET становятся доступны WCF-службе. В таблице 7.1 описано соотношение между этими двумя параметрами.

Таблица 7.1. Параметры, делающие средства ASMX доступными для WCF-служб

aspNetCompatabiltyEnabled в web.config	AspNetCompatibility-RequirementsMode в [ServiceBehavior]	Функции ASMX включены
True	NotAllowed	Нет – ошибка активации
True	Allowed	Да
True	Required	Да
False (По умолчанию)	NotAllowed	Нет
False (По умолчанию)	Allowed	Нет
False (По умолчанию)	Required	Нет – ошибка активации

Однако остается еще несколько моментов, требующих пояснения.

- HttpContext.Current. И ConfigurationManager.AppSettings, и ConfigurationManager.GetSection работают. HttpContext.Current имеет корректное значение в потоках WCF threads.
- **Глобализация**. Можно задавать для потока культуру и получать доступ к секции Globalization в <system.web>.
- **Олицетворение**. WCF поддерживает олицетворение на уровне службы и операции с помощью поведений – в дополнение к тому, что реализовано в ASP.NET. Если служба разрешает олицетворение с помощью WCF, задание этого режима в ASP.NET, игнорируется. Если сама служба не реализует олицетворение, действуют правила ASP.NET.
- **Состояние сеанса**. Полностью реализовано и управляется конфигурацией ASP.NET. Сохранять состояние можно внутри процесса, на отдельном сервере или в базе данных SQL.

Если режим совместимости с ASP.NET включен, то службы могут пользоваться средствами ASP.NET. В листинге 7.5 мы задействуем два таких средства. Во-первых, для сохранения состояния сеанса используется механизм SessionState, доступный ASMX-службам. Управлять созданием экземпляров службы можно, задавая значения PerCall, PerSession или Single. Этот вопрос подробно обсуждается в главе 5 «Поведения». В данном примере мы задали значение PerSession, так что если клиент многократно обращается к службе с помощью одного и того же экземпляра прокси-класса, то состояние сеанса между вызовами будет сохраняться. В WCF есть много способов сохранить данные уровня сеанса, но для тех, кто знаком с ASMX, этот механизм наиболее удобен. Во-вторых, мы пользуемся знакомой секцией AppSettings в файле web.config для задания конфигурационных данных, специфичных для приложения. В коде службы для доступа к этим значениям применяется набор AppSettings, раскрываемый в виде свойства объекта ConfigurationManager.

Листинг 7.5. Доступ к состоянию сеанса ASMX и к параметрам приложения

```
using System;
using System.Web;
using System.Web.Configuration;
using System.Configuration;
using System.ServiceModel;
using System.Runtime.Serialization;
using System.ServiceModel.Activation;

namespace EssentialWCF
{
   [DataContract]
   public class StockPrice
   {
      [DataMember] public string Source;
      [DataMember] public int calls;
      [DataMember] public double price;
   }

   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      StockPrice GetPrice(string ticker);
   }

   [ServiceBehavior(InstanceContextMode =
                         InstanceContextMode.PerSession)]
   [AspNetCompatibilityRequirements
                  (RequirementsMode=
                         AspNetCompatibilityRequirementsMode.Required)]
   public class StockService : IStockService
   {
      public StockPrice GetPrice(string ticker)
      {
         StockPrice p = new StockPrice();
         int nCalls = 0;
         if (HttpContext.Current.Session["cnt"] != null)
               nCalls = (int)HttpContext.Current.Session["cnt"];
         HttpContext.Current.Session["cnt"] = ++nCalls;

         p.calls = nCalls;
         p.price = 94.85;
         p.Source = ConfigurationManager.AppSettings["StockSource"];
         return p;
      }
   }
}
```

Чтобы режим `PerSession` работал, на стороне клиента необходимо сохранять идентификатор сеанса и передавать его службе при каждом обращении. В случае ASP.NET это делается с помощью кука, передаваемого в заголовке HTTP. Поэтому для организации сеансов в ASMX клиент должен разрешить сохранение куков. Поскольку стандартные привязки к HTTP – `basicHttpBinding` и `wsHttpBinding` – по умолчанию запрещают куки, в клиентском файле `app.config` необходимо изменить конфигурацию привязки, задав параметр `AllowsCookies=true`. В листинге 7.6 показано, как включить режим совместимости с ASP.NET на стороне службы.

Листинг 7.6. Включение режима совместимости с ASP.NET на стороне службы

```
   <system.serviceModel>
      <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
      <services>
         <service behaviorConfiguration="MEXServiceTypeBehavior"
                    name="EssentialWCF.StockService">
            <endpoint address="" binding="basicHttpBinding"
                         contract="EssentialWCF.IStockService" />
            <endpoint address="mex" binding="mexHttpBinding"
                         contract="IMetadataExchange" />
         </service>
      </services>
      <behaviors>
         <serviceBehaviors>
            <behavior name="MEXServiceTypeBehavior">
               <serviceMetadata httpGetEnabled="true" />
            </behavior>
         </serviceBehaviors>
      </behaviors>
   </system.serviceModel>
```

В листинге 7.7 показано, как включить куки на стороне клиента. Этот файл был сгенерирован операцией Add Service Reference в Visual Studio. Обратите внимание, что атрибут `allowCookies` установлен в `true`.

Листинг 7.7. Включение куков в конфигурационном файле клиента

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <basicHttpBinding>
            <binding name="BasicHttpBinding_IStockService"
                        closeTimeout="00:01:00"
                        openTimeout="00:01:00" receiveTimeout="00:10:00"
                        sendTimeout="00:01:00"
                        allowCookies="true"
                        bypassProxyOnLocal="false"
                        hostNameComparisonMode="StrongWildcard"
                        maxBufferSize="65536" maxBufferPoolSize="524288"
                        maxReceivedMessageSize="65536"
                        messageEncoding="Text" textEncoding="utf-8"
                        transferMode="Buffered"
                        useDefaultWebProxy="true">
                        <readerQuotas maxDepth="32"
                              maxStringContentLength="8192"
```

```
                        maxArrayLength="16384"
                        maxBytesPerRead="4096"
                        maxNameTableCharCount="16384" />
                    <security mode="None">
                        <transport clientCredentialType="None"
                                proxyCredentialType="None"
                                realm="" />
                        <message clientCredentialType="UserName"
                                algorithmSuite="Default" />
                    </security>
                </binding>
            </basicHttpBinding>
        </bindings>
        <client>
            <endpoint
                address="http://myserver/WCFASMXState/StockService.svc"
                binding="basicHttpBinding"
                bindingConfiguration="BasicHttpBinding_IStockService"
                contract="localhost.IStockService"
                name="BasicHttpBinding_IStockService" />
        </client>
    </system.serviceModel>
</configuration>
```

Включение олицетворения в смысле ASP.NET для приложений, работающих в режиме совместимости, делается в WCF точно так же, как в .NET 1.X. Для этого нужно включить элемент `<identity impersonate="true"/>` в секцию `<system.web>` файла `web.config`. После этого верительные грамоты клиента будут автоматически передаваться службе, и служба станет выполнять операции с правами клиента.

Включить олицетворение можно одним из двух способов. Чтобы задать его на уровне службы в целом, используйте атрибут `impersonateCallerForAllOperations=true` в поведении службы и значение `ImpersonationOption.Allowed` в поведении операции. Чтобы разрешить олицетворение только на уровне одной операции, задайте `ImpersonationOption.Required` в поведении операции, ничего не изменяя в поведении службы.

В листинге 7.8 показано, как задать олицетворение на уровне операции в предположении, что это не запрещено на уровне службы в файле `web.config`. Когда клиент обращается к службе, идентификатор, под которым он вошел в систему, возвращается в члене `RequestedBy`. Если удалить поведение службы, то `RequestedBy` вернет принимаемое по умолчанию значение идентификатора Network Service. Более подробно олицетворение обсуждается в главе 8 «Безопасность».

Листинг 7.8. Разрешение олицетворения

```
namespace EssentialWCF
{
    [DataContract]
    public class StockPrice
    {
        [DataMember] public string RequestedBy;
        [DataMember] public double price;
    }

    [ServiceContract]
    public interface IStockService
    {
        [OperationContract]
        StockPrice GetPrice(string ticker);
    }

    [ServiceBehavior]
    [AspNetCompatibilityRequirements
                (RequirementsMode=
                        AspNetCompatibilityRequirementsMode.Required)]
    [ServiceContract]
    public class StockService : IStockService
    {
        [OperationBehavior(Impersonation =
                                ImpersonationOption.Required)]
        [OperationContract]
        public StockPrice GetPrice(string ticker)
        {
            StockPrice p = new StockPrice();
            p.RequestedBy = WindowsIdentity.GetCurrent().Name;
            p.price = 94.85;
            return p;
        }
    }
}
```


Авторазмещение

Чаще всего WCF-службы размещаются в IIS или WAS. Оба продукта, будучи построены на базе общей архитектуры, предоставляют надежные средства управления процессами и контроля над временем жизни, а также знакомый административный интерфейс. Поэтому такое решение оправдано в большинстве ситуаций, где инфраструктура IIS уже присутствует.

Но бывают случаи, когда размещать службы внутри IIS или WAS нежелательно. Например, вам необходимо явно управлять запуском и остановом. Или нужен специализированный интерфейс управления, отличный от тех, что предлагают IIS и WAS. Тогда вы можете разместить службу в любой программе, воспользовавшись классом `ServiceHost` из пространства имен `System.ServiceModel`. Это называется *авторазмещением* (self-hosting) WCF-службы.

Распространенная ситуация – размещение WCF-службы в управляемой службе Windows, которая запускается и останавливается вместе с операционной системой. Такой способ годится для любой системы, поддерживающей WCF, в том числе Windows XP, Windows 2003 Server, Windows Vista и Windows Server 2008. Подробно он рассматривается в разделе «Авторазмещение внутри управляемой службы Windows» ниже в этой главе.

Другой вариант – размещение службы в приложении для рабочего стола, написанном с использованием WinForms либо Windows Presentation Framework или

в консольном приложении. Такая служба могла бы работать как узел пиринговой сети, либо прослушивать хорошо известный адрес, на который клиенты отправляют сообщения, либо объявлять свой адрес каким-либо иным способом. Если в качестве транспорта служба использует очередь с постоянным хранилищем, то ей можно отправлять сообщения даже тогда, когда владеющая ей программа не запущена. Очередь, реализованная с помощью MSMQ или в виде таблиц в реляционной базе данных, представляет собой удобный механизм коммуникации между клиентом и службой; любая сторона может на время отключаться.

Реализовать авторазмещаемую службу очень просто. Три необходимых шага продемонстрированы в листинге 7.1. *Владелец* – программа, которая создает объект `ServiceHost` и вызывает его метод `Open`, – отвечает за то, чтобы служба оставалась дееспособной до момента останова. При создании `ServiceHost` можно задавать определенные параметры, например: откуда брать базовый адрес службы, организовывать ли службу как синглет и т.п. Но на этом все трудности и заканчиваются.

В листингах 1.1 и 1.2 из главы 1 показана минимальная реализация авторазмещения службы. В этом примере служба размещается в консольном приложении, которое может быть запущено на сервере или на рабочем столе администратора.

Авторазмещение внутри управляемой службы Windows

Управляемые службы Windows – это процессы операционной системы, управляемые компонентом Service Control Manager (SCM). Существует графический административный интерфейс в виде консоли управления службами (Services Microsoft Management Console), но с помощью технологии Windows Management Instrumentation и SCM API можно реализовать управление и с помощью других инструментов конфигурирования или в сценарии. Вы можете настраивать разнообразные параметры служб, например: должна ли служба запускаться автоматически вместе с ОС и от имени какого пользователя Windows она должна работать. В виде служб Windows реализованы многие приложения масштаба предприятия, в частности, Microsoft SQL Server и Microsoft Exchange.

Доступ к инфраструктуре служб Windows возможен как из неуправляемого кода через Win32 API, так и из управляемого – с помощью класса `ServiceBase`, находящегося в пространстве имен `System.ServiceProcess`. И в том, и в другом случае вы получаете базовый интерфейс администрирования, но никаких средств для собственно размещения, масштабирования, обеспечения безопасности и надежности. На вас ложится ответственность за реализацию коммуникации (с помощью MSMQ, именованных каналов, TCP-сокетов и т.д.), а также за организацию многопоточности, стратегию создания экземпляров и ограничение пропускной способности. К счастью, все это уже реализовано в WCF, поэтому при размещении WCF-службы внутри службы Windows вам ничего делать не придется.

В Visual Studio имеется готовый шаблон для создания службы Windows. В проекте, созданном по этому шаблону, есть статический метод `Main()`, который

запускает службу, и класс, производный от `ServiceBase`, который вы наполняете своим кодом. Сгенерированный код необходимо дополнить в двух направлениях: создать объект `ServiceHost`, который породит экземпляр службы, и зарегистрировать службу в SCM.

Прежде всего, следует добавить в метод `OnStart` код, который начнет обработку входящих сообщений. До появления WCF именно в этом месте вы создавали потоки, прослушиватели и реализовывали механизм рециркуляции – обычно руководствуясь параметрами в конфигурационных файлах. А теперь достаточно лишь создать объект `ServiceHost` и начать прослушивание. Полезно также воспользоваться классом `EventLog` для помещения в журнал информационного сообщения о запуске.

Затем следует реализовать класс `ProjectInstaller`, который определен в пространстве имен `System.Configuration.Install`. Он нужен для установки службы на тот компьютер, где она будет работать. Можно сделать это в программе установки или прямо в службе. Для установки службы вы можете воспользоваться утилитой installutil.exe, которая зарегистрирует ее в SCM, после чего служба готова к работе.

В листинге 7.9 показана полная реализация службы Windows.

Листинг 7.9. WCF-служба, размещенная в службе Windows

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.ServiceProcess;
using System.Configuration;
using System.Configuration.Install;
using System.ServiceModel;
using System.ServiceModel.Description;

namespace EssentialWCF
{
   [RunInstaller(true)]
   public class ProjectInstaller : Installer
   {
      private ServiceProcessInstaller process;
      private ServiceInstaller service;

      public ProjectInstaller()
      {
         process = new ServiceProcessInstaller();
         process.Account = ServiceAccount.LocalSystem;
         service = new ServiceInstaller();
         service.ServiceName = "EssentialWCF";
         Installers.Add(process);
         Installers.Add(service);
      }
   }

   [ServiceContract]
   public class StockService
```

```
    {
        [OperationContract]
        private double GetPrice(string ticker)
        {
            return 94.85;
        }
    }

    public partial class Service : ServiceBase
    {
        public Service()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            ServiceHost serviceHost = new
                              ServiceHost(typeof(StockService));
            serviceHost.Open();
            ServiceEndpoint endpoint =
                              serviceHost.Description.Endpoints[0];

            EventLog.WriteEntry(endpoint.Contract.Name + " Запущена"
                                        + " прослушивает " + endpoint.Address
                                        + " (" + endpoint.Binding.Name + ")",
                        System.Diagnostics. EventLogEntryType.Information);
        }

        protected override void OnStop()
        {
            EventLog.WriteEntry("EssentialWCF останавливается",
                        System.Diagnostics.EventLogEntryType.Information);
        }
    }
}
```

На рис. 7.5 приведен снимок с экрана, где показан Service Control Manager с запущенной службой EffectiveWCF.

Размещение нескольких служб в одном процессе

Правильный выбор уровня агрегирования возможностей системы – важный аспект проектирования. Если у системы слишком много интерфейсов, в ней будет трудно разобраться. Если слишком мало, она окажется монолитной, и модифицировать ее будет нелегко.

В главе 2 «Контракты» мы показали, как можно объединить интерфейсы нескольких классов в одной оконечной точке. Достигается это за счет механизма агрегирования интерфейсов в .NET. Мы также описали, как одна служба может раскрывать несколько оконечных точек. В этом разделе мы рассмотрим альтернативный подход. Вместо того чтобы агрегировать два интерфейса в один и раскрывать агрегат в виде одной службы, мы покажем, как разместить две независимые службы в одном процессе операционной системы.

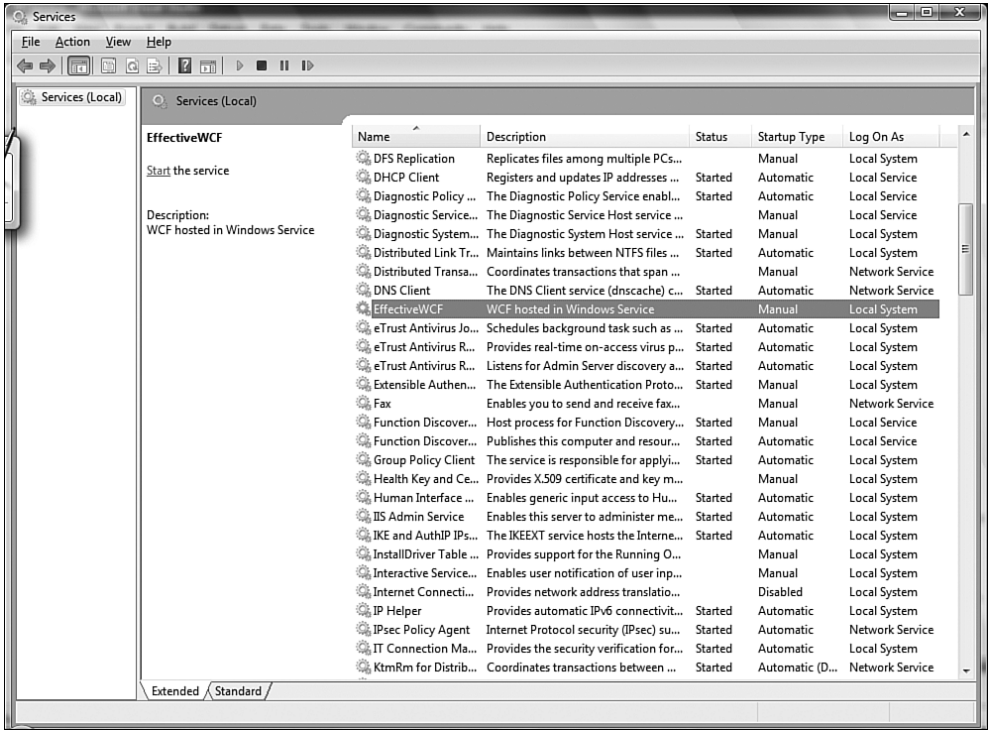

Рис. 7.5. Менеджер управления службами (SCM)

Класс `ServiceHost` раскрывает ровно одну службу. Поэтому для раскрытия нескольких служб в одном процессе необходимо создать несколько экземпляров класса `ServiceHost`. Именно так и поступает WAS – создает по одному объекту `ServiceHost` для каждой службы, объявленной в SVC-файле. В SVC-файле хранится имя службы, оконечные точки которой описаны в файле `web.config` приложения. В описании оконечной точки указаны адрес, привязка и контракт, поэтому у объекта `ServiceHost` есть все необходимое для того, чтобы начать прослушивание и диспетчеризацию входящих сообщений.

При авторазмещении служб вы можете точно так же создать несколько экземпляров `ServiceHost`. Каждый владелец абсолютно независим от всех остальных, общий у них только процесс операционной системы. У каждого владельца есть свой раздел внутри секции `<system.serviceModel>` файла `app.config`. После запуска `ServiceHost` WCF управляет потоками и порождением экземпляров самостоятельно, поэтому вашей программе для реализации этой логики делать ничего не нужно.

В листинге 7.10 показано консольное приложение, в котором размещены две службы. Метод GetStockPrice класса GoodStockService ждет десять секунд перед тем, как вернуть результат, а одноименный метод класса GreatStockService возвращает результат немедленно. В соответствии с конфигурацией поведения служб это простое приложение многопоточно, поэтому, пока GoodStockService спит, GreatStockService продолжает отвечать на запросы. И даже медленная служба многопоточна, так что новые входящие сообщения доставляются новым экземплярам GetStockPrice, создаваемым по мере необходимости.

Листинг 7.10. Авторазмещение нескольких служб в одном процессе

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.Configuration;
using System.Threading;

namespace EssentialWCF
{
   [ServiceContract]
   public class GoodStockService
   {
      [OperationContract]
      public double GetStockPrice(string ticker)
      {
         Thread.Sleep(10000);
         return 94.85;
      }
   }
   [ServiceContract]
   public class GreatStockService
   {
      [OperationContract]
      public double GetStockPrice(string ticker)
      {
         return 94.85;
      }
   }

   public class program
   {
      // Размещаем службы в консольном приложении.
      public static void Main( )
      {
         ServiceDescription desc = null;
         ServiceHost serviceHost1 = new
                           ServiceHost(typeof(GoodStockService));
         serviceHost1.Open();
         Console.WriteLine("Служба #1 готова.");

         ServiceHost serviceHost2 = new
                           ServiceHost(typeof(GreatStockService));
         serviceHost2.Open();
```



```
         Console.WriteLine("Служб #2 готова.");

         Console.WriteLine("Для завершения нажмите <ENTER>.\n\n");
         Console.ReadLine();

         // Закрываем оба ServiceHost и тем самым останавливаем службы.
         serviceHost1.Close();
         serviceHost2.Close();
      }
   }
}
```

В файле app.config, который показан в листинге 7.11, есть две секции <service> – по одной для каждой службы. Базовые адреса служб уникальны. Отметим, что адрес единственной оконечной точки каждой службы пуст, то есть она ожидает поступления сообщений на базовый адрес соответствующей службы. Для каждой службы может существовать не более одной оконечной точки с пустым адресом.

Листинг 7.11. Конфигурация нескольких служб с авторазмещением в одном процессе

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.serviceModel>
      <services>
         <service name="EssentialWCF.GoodStockService"
                        behaviorConfiguration="mexServiceBehavior">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8001/EssentialWCF/"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                           binding="basicHttpBinding"
                           contract="EssentialWCF.GoodStockService" />
         </service>

         <service name="EssentialWCF.GreatStockService"
                        behaviorConfiguration="mexServiceBehavior">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8002/EssentialWCF/"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                           binding="basicHttpBinding"
                           contract="EssentialWCF.GreatStockService" />
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
```

```
            <behavior name="mexServiceBehavior">
               <serviceMetadata httpGetEnabled="True"/>
            </behavior>
         </serviceBehaviors>
      </behaviors>

   </system.serviceModel>
</configuration>
```

Определение адресов службы и оконечных точек

WCF-служба представляет собой набор оконечных точек, каждая из которых имеет уникальный адрес. Адрес и привязка оконечной точки определяют, куда и как ей нужно отправлять сообщения. Помимо оконечных точек, адрес есть и у самой службы; он называется базовым адресом. Базовый адрес службы используется в качестве базы относительных адресов, определенных для ее оконечных точек. Задавая относительные, а не абсолютные адреса оконечных точек, мы упрощаем себе задачу управления ими, поскольку для изменения адресов всех оконечных точек разом достаточно изменить лишь базовый адрес службы.

Абсолютный адрес оконечной точки формируется путем дописывания ее относительного адреса в конец базового адреса службы. Например, если базовый адрес службы равен http://localhost/foo, а относительный адрес оконечной точки равен bar, то эта оконечная точка будет ожидать входящие сообщения по адресу http://localhost/foo/bar.

Если для оконечной точки задан абсолютный адрес, то он оказывается совершенно не связанным с базовым адресом службы. Например, базовый адрес службы может быть равен http://localhost/foo, а адрес ее оконечной точки – net.tcp://bar/MyOtherService/.

У службы может быть несколько базовых адресов, но не более одного на каждую схему URI. Если для оконечной точки задан относительный адрес, то WCF ищет базовый адрес службы, соответствующий транспортному протоколу, указанному в привязке оконечной точки. Например, если для службы определены базовые адреса http://localhost/ и net.tcp://bigserver/, а для оконечной точки – относительный адрес foo и привязка `basicHttpBinding`, то ее абсолютным адресом будет http://localhost/foo. Если для другой оконечной точки той же службы тоже задан относительный адрес foo, то абсолютным адресом будет netc.tcp/bigserver/foo.

При размещении в IIS базовым адресом службы становится адрес того виртуального каталога IIS, в котором находится SVC-файл. Если файл `MyService.SVC` находится в каталоге http://localhost/foo/, то базовый адрес службы равен http://localhost/foo/. Если служба размещается в IIS, то адреса оконечных точек в файле `web.config` должны быть относительными.

В листинге 7.12 приведен конфигурационный файл службы. Отметим следующие моменты:

- **Базовые адреса**. Для службы определены два базовых адреса с разными протоколами. Если бы в обоих адресах был указан один и тот же протокол, то оказалось бы невозможно понять, какими должны быть полные адреса оконечных точек, поэтому в момент активации WCF возбудила бы исключение.
- **Пустой относительный адрес**. Адрес первой оконечной точки пуст, поэтому ее абсолютный адрес совпадает с тем базовым адресом службы, для которого используется тот же самый протокол.
- **Непустой относительный адрес**. Для второй оконечной точки задан адрес ws. Добавив базовый адрес с тем же протоколом, получим, что ее абсолютный адрес равен http://localhost:8000/ EssentialWCF/ws.

Листинг 7.12. Адресация службы и оконечных точек в конфигурационном файле

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.StockService"
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EssentialWCF/"/>
                  <add baseAddress="net.tcp://localhost:8001/EssentialWCF/"/>
               </baseAddresses>
            </host>
            <endpoint address=""
                        binding="basicHttpBinding"
                        contract="EssentialWCF.IStockService" />
            <endpoint address="secure"
                        binding="wsHttpBinding"
                        contract="EssentialWCF.IStockService" />
            <endpoint address="fast"
                        binding="netTcpBinding"
                        contract="EssentialWCF.IStockService" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

С помощью классов из пространства имен `System.ServiceModel.Description` вы можете получить от объекта `ServiceHost` всю информацию об адресах и привязках. В листинге 7.13 показано, как это сделать.

Листинг 7.13. Печать информации об адресах и привязках

```
foreach (Uri uri in serviceHost.BaseAddresses)
   Console.WriteLine("Base Addr Uri     :  {0}", uri.AbsoluteUri);

foreach (ServiceEndpoint endpoint in serviceHost.Description.Endpoints)
{
```

```
    Console.WriteLine("\nEndpoint - address:  {0}",
                                  endpoint.Address);
    Console.WriteLine("           binding:  {0}",
                                 endpoint.Binding.Name);
    Console.WriteLine("           contract: {0}",
                                 endpoint.Contract.Name);
}
```

На рис. 7.6 показано, что выведет эта программа для конфигурационного файла из листинга 7.12.

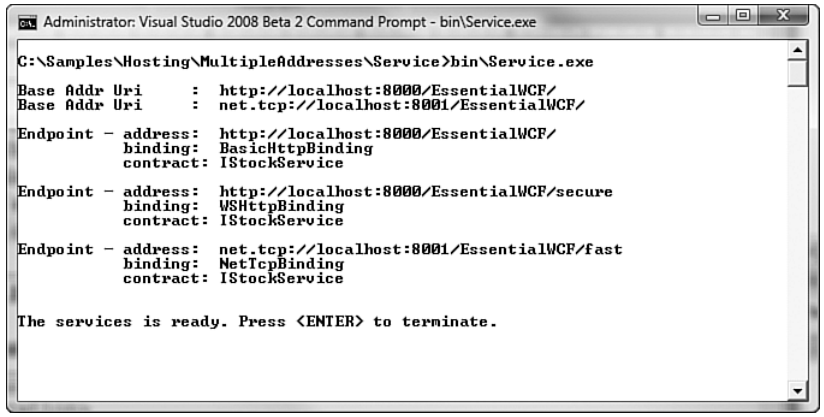

Рис. 7.6. Просмотр адресов и привязок работающей службы

Резюме

В вопросе о размещении WCF демонстрирует завидную гибкость. WCF-службы можно размещать практически в любом процессе операционной системы. Владелец службы отвечает за ее запуск и останов, а также предоставляет базовые средства управления. При выборе способа размещения службы следует учитывать эксплуатационные требования, как то: доступность, надежность и управляемость.

Как IIS, так и Windows Process Activation Services (WAS) располагают встроенной инфраструктурой для размещения WCF-служб. Средства, которые раньше были доступны только для IIS, – активация процессов, рециркуляция и управление идентичностью – теперь перенесены в WAS и могут использоваться для протоколов, отличных от HTTP. Таким образом, WAS является надмножеством IIS, но IIS по-прежнему осается идеальной средой для размещения WCF-служб на базе HTTP. В режиме совместимости с ASP.NET WCF поддерживает многие возможности ASMX-служб.

Помимо IIS, WCF-службы часто размещаются в управляемых службах Windows. Эти процессы управляются с помощью компонента Service Control Manager (SCM), и для них имеется хорошо знакомый административный интерфейс.

В Visual Studio встроен готовый шаблон для создания служб Windows, поэтому с учетом простоты разработки и управления службы Windows неплохо подходят для авторазмещения WCF-служб.

В таблице 7.2 перечислены наиболее употребительные варианты размещения.

Таблица 7.2. Варианты размещения

Владелец	Когда предпочесть
IIS	Размещайте в IIS службы, которые должны работать без участия человека и запускаться и останавливаться вместе с операционной системой. Если вы производите развертывание в среде, где IIS уже работает, то административные политики, скорее всего, уже подготовлены и персонал IT-департамента обучен. Но IIS поддерживает только протокол HTTP. Размещайте в IIS написанные с помощью WCF службы, (а не ASMX), для которых желательно сохранить доступ к некоторым средствам, предлагаемым в ASMX/ASP.NET
WAS	Размещайте в WAS службы, которые должны работать без участия человека и запускаться и останавливаться вместе с операционной системой, но при этом пользуются протоколами TCP, MSMQ, именованными каналами и другим транспортными механизмами
Управляемая служба	Размещайте в управляемой службе (NT-службе), если вам необходим специализированный административный интерфейс для запуска и останова владельца. Управляемые службы можно сконфигурировать так, чтобы они запускались и останавливались вместе с системой. Многие коммерческие приложения реализованы в виде управляемых служб
Персональное приложение	Размещайте в персональном приложении службы, предназначенные для интерактивного взаимодействия с пользователем. Типичным примером являются узлы пиринговых сетей
Консольное приложение	Консольное приложение хорошо для тестирования, поскольку его просто отлаживать

Глава 8. Безопасность

В современном мире трудно представить себе грань бизнес-приложений, более критичную, чем безопасность. Конечно, производительность и доступность тоже немаловажны, но приложение, которое безопасно лишь *иногда*, мало кому нужно (на самом деле, оно даже скорее вредно, чем полезно). Обращаясь к онлайновому банковскому сервису, мы верим, что организация, предоставляющая его, сделала все возможное для предотвращения злонамеренного использования, порчи данных, действий хакеров и раскрытия сведений о наших финансах посторонним лицам. Того же самого ожидают от авторов WCF-служб, рассчитанных на потребителей.

Эта глава посвящена концепциям, лежащим в основе безопасности, и тем практическим средствам, с помощью которых обеспечивается безопасность служб в WCF (когда это необходимо). Начнем с основных понятий, а потом перейдем к деталям, сопровождая изложение многочисленными примерами.

Мы также скажем несколько слов о создании и использовании сертификатов, так как эта тема важна для понимания последующего материала. Вооружившись этими познаниями, мы приступим к рассмотрению механизмов обеспечения безопасности на транспортном уровне и на уровне сообщений.

Значительная часть этой главы посвящена практическим аспектам защиты служб в наиболее распространенных ситуациях. Они разбиты на две больших категории: безопасность в сетях Интранет и Интернет.

Закончим мы эту главу демонстрацией того, как включить аудит, чтобы отслеживать и диагностировать проблемы, возникающие при аутентификации и авторизации клиентов, обращающихся к операциям службы.

Концепции безопасности в WCF

Прежде чем переходить к коду, конфигурации и процедурам реализации безопасных служб, необходимо познакомиться с основными понятиями, относящимися к безопасности: аутентификацией, авторизацией, конфиденциальностью и целостностью. Определив их, мы далее обратимся к идеям, положенным в основу безопасности на уровне транспорта и сообщений в WCF.

Аутентификация

Одна из самых важных проблем безопасности – узнать, кто стучится в дверь. Аутентификацией называется процедура установления личности субъекта, например, путем проверки предоставляемых им верительных грамот в виде имени и

пароля. Не менее важно, чтобы обратившаяся сторона была уверена, что ей отвечает именно та служба, к которой она обращалась, а не какой-нибудь самозванец.

WCF предоставляет несколько механизмов взаимной аутентификации, в том числе на основе сертификатов или учетных записей и групп Windows. Применяя эти и другие средства, каждая сторона может надежно удостовериться в том, что имеет дело с ожидаемым партнером.

Авторизация

После того как подлинность обратившейся стороны установлена, необходимо решить, разрешено ей сделать то, за чем она обратилась. Этот шаг называется *авторизацией*. Отметим, что можно авторизовать и анонимных пользователей, поэтому, хотя обычно авторизация следует за аутентификацией, строго говоря, эти два этапа независимы.

Авторизацию можно выполнять по-разному: прибегнув к услугам встроенных в систему или нестандартных провайдеров авторизации, с помощью специально написанного кода внутри службы, ролей ASP.NET, групп Windows, каталога Active Directory, менеджера авторизации (Authorization Manager) и т.д.

Конфиденциальность

Когда речь идет о секретной информации, мало толку идентифицировать и авторизовать клиента, если результаты обращения передаются так, что их может прочитать любой желающий. Под *конфиденциальностью* понимается механизм, предотвращающий чтение информации, которой обмениваются клиент и служба, посторонними лицами. Обычно это достигается за счет шифрования, и WCF предоставляет для этого разнообразные средства.

Целостность

Последняя задача, которую нужно решить для обеспечения безопасности, – гарантировать, что сообщение не было изменено в процессе передачи от клиента службе и наоборот. Обычно, чтобы добиться этого, отправитель включает в сообщение цифровую подпись или подписанную свертку сообщения (дайджест), а получатель проверяет подпись, сверяя ее с содержимым полученного сообщения. Если результат вычисления подписи на стороне получателя не совпадает со значением, которое пришло в составе сообщения, сообщение надлежит отбросить.

Отметим, что целостность можно обеспечить даже тогда, когда конфиденциальность не нужна. Иногда допустимо посылать информацию в открытом виде при условии, что получатель способен проверить ее неизменность с помощью цифровой подписи.

Безопасность на уровне транспорта и сообщений

В WCF мы встречаем две основных категории безопасности; обе относятся к безопасности того, что передается между клиентом и службой (иногда эту кон-

цепцию называют *безопасностью передачи*). Первая – защита данных при передаче по сети, или *безопасность на уровне транспорта*. Вторая – *безопасность на уровне сообщения*, то есть способ защиты самого сообщения вне зависимости от механизма его транспортировки.

Безопасность на уровне транспорта обеспечивает защиту отправляемых данных безотносительно к их содержимому. Обычно для этой цели применяют протокол Secure Sockets Layer (SSL), с помощью которого шифруются и подписываются пакеты, передаваемые по протоколу HTTPS. Существуют и другие механизмы защиты на транспортном уровне, выбор того или другого зависит от привязки WCF. Как вы увидите ниже, в WCF многие протоколы передачи данных, например TCP, по умолчанию безопасны.

У безопасности на транспортном уровне есть ограничение: предполагается, что на всех участках сетевого тракта и у всех участников конфигурация параметров безопасности согласована. Иными словами, если между отправителем и получателем сообщения имеется посредник, то нет никакой гарантии, что на участке после посредника безопасность по-прежнему обеспечена (если только посредник не контролируется поставщиком исходной службы). А коль скоро это так, то при прохождении этого участка данные могут быть скомпрометированы. Кроме того, нельзя быть уверенным, что сам посредник не изменил сообщение перед тем, как передать его дальше. Особенно эти соображения важны для служб, доступ к которым производится через маршрутизаторы в Интернете, а для служб, потребляемых внутри корпоративной интрасети это менее существенно.

Безопасность на уровне сообщений дает гарантию целостности и конфиденциальности отдельных сообщений вне зависимости от поведения сети. Инфраструктура открытых и закрытых ключей позволяет зашифровать и подписать сообщение, так что оно будет защищено даже при передаче по незащищенному транспортному каналу (например, по обычному протоколу HTTP).

Надо ли включать безопасность на уровне транспорта и сообщений, обычно задается в конфигурационном файле; в листинге 8.1 приведены два простых примера.

Листинг 8.1. Примеры безопасности на уровне транспорта и сообщений

```
<basicHttpBinding>
   <binding name="MyBinding">
      <security mode="Transport">
         <transport clientCredentialType="Windows"/>
      </security >
   </binding>
</basicHttpBinding>

<wsHttpBinding>
   <binding name="MyBinding">
      <security mode="Message">
         <transport clientCredentialType="None"/>
      </security >
   </binding>
</wsHttpBinding>
```


Далее в этой главе мы увидите разнообразные примеры обеспечения безопасности на уровне как транспорта, так и сообщений, а в некоторых случаях на том и другом одновременно.

Шифрование на базе сертификатов

Сертификаты и представляемые с их помощью утверждения – это безопасный универсальный метод доказательства подлинности. Они прекрасно приспособлены для шифрования и аутентификации. В WCF применяются сертификаты стандарта X.509, которые широко поддержаны многими поставщиками программного обеспечения. Интернет-браузеры и серверы пользуются этим форматом для хранения ключей шифрования и подписей при обмене данными по протоколу SSL. Сертификаты обеспечивают сильное шифрование, хорошо документированы и признаны сообществом.

Основной недостаток сертификатов – это расходы на их приобретение у третьей стороны (удостоверяющего центра) и сложности с организационным обеспечением. Как их распространять? Что делать, если сертификат украден? Как восстановить данные, если сертификат утрачен? Если сертификаты хранятся на клиентском компьютере, как получить доступ к ним, находясь в дороге? Есть немало подходов к решению этих проблем, начиная от хранения сертификатов в каталоге, который находится в интрасети или в открытой части Интернета, до хранения в смарт-карте, которую можно носить с собой в бумажнике. Как бы ни решалась организационная проблема, сертификаты представляют собой хорошее средство для шифрования и аутентификации.

Основные идеи

Идея шифрования сообщений асимметричным ключом в целом довольно проста. Представьте себе алгоритм, который может зашифровать произвольную строку одним ключом, а расшифровать другим. Теперь представьте, что у меня есть такая пара ключей, и один из них делаю открытым и доступным любому лицу через Интернет, а второй остается закрытым и доступен только мне. Если мой приятель захочет послать мне сообщение, он найдет мой открытый ключ, зашифрует с его помощью сообщение и отправит его в зашифрованном виде. Если такое сообщение перехватит мой недруг, то прочесть его он не сможет, так как только у меня есть ключ, с помощью которого его можно расшифровать. Отправляя своему приятелю ответ, я нахожу его открытый ключ и зашифровываю им свое сообщение. Расшифровать его сможет только он, так что вся переписка оказывается конфиденциальной.

При использовании цифровых подписей тоже используется шифрование, но в обратном порядке. Цифровая подпись – это просто строка, зашифрованная закрытым ключом, следовательно, расшифровать ее может любой, кому известен соответствующий открытый ключ. Информация, получающаяся после расшифровки (например, мое имя) не секретна; всякий, кто расшифрует подпись моим открытым ключом, убедится, что подписал сообщение именно я.

Еще один важный аспект сертификатов – это отношение доверия. Возвращаясь к обмену сообщениями с приятелем, как я могу быть уверен, что располагаю именно его открытым ключом, а не ключом нашего недруга? Чтобы клиент и служба могли проверить правильность и подлинность сертификатов друг друга, а также убедиться, что ни один из них не был отозван, они должны доверять некоей третьей стороне. Не страшно, если сертификаты клиента и службы были выпущены разными удостоверяющими центрами, коль скоро оба центра доверяют третьему, общему для них. Этот общий центр называется корневым и обычно подписывает свои сертификаты сам, не доверяя больше никому. Получив сертификат от службы, клиент проверяет всю цепочку подписавших его удостоверяющих центров, желая убедиться, что она заканчивается центром, которому можно доверять. Если это так, клиент считает, что сертификат подлинный, иначе отбрасывает его. В WCF есть возможность отключить проверку пути сертификации, чтобы в ходе разработки и тестирования можно было пользоваться сертификатами, которые подписаны не достойными доверия центрами.

Подготовка

Сертификаты можно применять для обеспечения безопасности на уровне транспорта или сообщений. Наиболее распространенный способ шифрования на транспортном уровне – SSL – применяет к транспортному протоколу сертификат, выпущенный для сервера. Шифрование на уровне сообщений применяется к отдельным сообщениям. Если для шифрования транспорта необходимо устанавливать сертификат вместе со службой, то для шифрования сообщений имеются различные режимы – как с серверными, так и с клиентскими сертификатами.

В примерах в разделах «Безопасность на транспортном уровне» и «Безопасность на уровне сообщений» ниже будут фигурировать два компьютера: рабочая станция под управлением Windows Vista и сервер Windows 2003. У рабочей станции есть сертификат MyClientCert, а у сервера – MyServerCert. В листинге 8.2 приведены команды, которые нужно выполнить в Vista для генерации необходимых сертификатов. Программа makecert.exe создает сертификат. Флаг `-pe` позволяет экспортировать закрытый ключ. Флаг `-n` задает имя сертификата, по которому мы будем ссылаться на него в ходе аутентификации. Флаг `-sv` определяет имя файла, содержащего закрытый ключ. Флаг `-sky` определяет тип ключа субъекта и может принимать значения `exchange` (для обмена) или `signature` (для цифровой подписи). Утилита Pvt2pfx объединяет закрытый и открытый ключи в один файл.

Если вы ведете разработку на одной машине, замените MyServer на localhost. Все прочее остается тем же самым.

Промышленные сертификаты. Имейте в виду, что сгенерированные таким образом сертификаты не должны использоваться в ходе промышленной эксплуатации. Для этих целей сертификаты следует заказывать в доверенном удостоверяющем центре.

Листинг 8.2. Генерирование сертификатов

```
makecert.exe -r -pe -sky exchange
             –n "CN=MyClientCert"  MyClientCert.cer
             -sv MyClientCert.pvk
pvk2pfx.exe  -pvk MyClientCert.pvk
             -spc MyClientCert.cer
             -pfx MyClientCert.pfx

makecert.exe -r -pe -sky exchange
             -n "CN=MyServer.com"  MyServerCert.cer
             -sv MyServerCert.pvk
pvk2pfx.exe  -pvk MyServerCert.pvk
             -spc MyServerCert.cer
             -pfx MyServerCert.pfx
```

.Cer-файл содержит открытый ключ, .pvk-файл – закрытый ключ, а .pfx-файл – это файл для обмена ключами, содержащий оба. Для установки ключей применяется оснастка Certificates (Сертификаты) для консоли управления Microsoft.

1. Установить сертификат на сервере, поместив его в хранилище сертификатов локального компьютера:
   а) **Импортировать `MyServerCert.pfx` в папку Personal**. Это позволит серверу расшифровывать сообщения, зашифрованные открытым ключом, а также зашифровывать сообщения закрытым ключом;
   б) **Импортировать `MyClientCert.cer` в папку Trusted People**. Это позволит серверу расшифровывать сообщения, зашифрованные закрытым ключом MyClientCert, например, информационные сообщения и цифровые подписи на этапе аутентификации. Кроме того, сервер сможет зашифровывать сообщения закрытым ключом MyClientCert.
2. Установить сертификат на клиенте, поместив его в хранилище сертификатов текущего пользователя:
   а) **Импортировать `MyClientCert.pfx` в папку Personal**. Это позволит клиенту расшифровывать сообщения, зашифрованные его открытым ключом, а также зашифровывать сообщения своим закрытым ключом;
   б) **Импортировать `MyServerCert.cer` в папку Trusted People**. Это позволит клиенту расшифровывать сообщения, зашифрованные закрытым ключом MyServerCert, например, информационные сообщения и цифровые подписи на этапе аутентификации. Кроме того, клиент сможет зашифровывать сообщения закрытым ключом MyServerCert.

Безопасность на транспортном уровне

Как следует из самого названия, безопасность на транспортном уровне обеспечивает защиту информации в коммуникационном канале между клиентом и службой. На этом уровне может производиться шифрование и аутентификация. Доступные виды шифрования и протоколы аутентификации определяются стеком каналов (привязкой).

Как минимум, безопасность на транспортном уровне гарантирует, что все данные, которыми обмениваются клиент и служба, зашифрованы, поэтому понять, что находится в сообщении, могут только стороны-участники. Конкретный алгоритм шифрования либо определяется транспортным протоколом (HTTPS подразумевает SSL), либо задается в привязке (MSMQ может использовать шифры RC4Stream или AES).

Помимо шифрования, безопасность на транспортном уровне может включать аутентификацию – требуется, чтобы на этапе установления коммуникационного канала клиент передал службе свои верительные грамоты. В качестве последних могут выступать цифровые сертификаты, маркеры SAML, маркеры Windows или разделяемый секрет, например, имя и пароль. Кроме того, на транспортном уровне, еще до установления безопасного канала между клиентом и службой проверяется подлинность самой службы. Это защищает от атаки «с человеком посередине» и от подлога.

Шифрование по SSL

SSL – удобный и безопасный способ шифрования коммуникационного канала. Он хорошо освоен в IT-департаментах, способен работать через брандмауэры, и на рынке для него имеются разнообразные средства управления и настройки производительности. Применение SSL в сочетании с привязкой `basicHttpBinding` обеспечивает Web-службе максимально широкую доступность.

Для установления зашифрованного по SSL канала необходим цифровой сертификат с асимметричными ключами (открытый/закрытый). После того как канал установлен, для шифрования сообщений в обе стороны применяется более эффективный симметричный алгоритм шифрования.

Цифровой сертификат можно получить из различных источников. Существуют публичные удостоверяющие центры, например компания Verisign, которые выпускают сертификаты для тестирования и для промышленной эксплуатации. В комплект поставки Windows Server тоже входит служба выпуска сертификатов, поэтому вы можете самостоятельно сгенерировать сертификаты, которым будут доверять внутренние узлы организации и ее партнеры. Кроме того, в составе .NET имеется утилита MakeCert для выпуска тестовых сертификатов.

SSL поверх HTTP

SSL-шифрование можно применять к большинству транспортных протоколов (существенным исключением является протоколы с очередями), но чаще всего он используется совместно с HTTP. Если вы работаете с привязкой к транспорту HTTP, то вне зависимости от того, размещена ли служба в IIS или автоматически, необходимо сконфигурировать драйвер HTTP.SYS для поддержки SSL. В случае IIS привязку можно добавить с помощью инструмента IIS Administration. Для IIS 7 нужно выделить Web-сайт, под которым определен корень виртуального приложения, а затем выбрать пункт Bindings (Привязки) на панели Actions (Действия). В результате появится диалоговое окно, в котором вы сможете выбрать сертификат для SSL-коммуникаций (рис. 8.1).

Рис. 8.1. Конфигурирование IIS 7 для работы с SSL

При авторазмещении службы на платформе Windows Server 2008 или Vista можно воспользоваться инструментом *netsh*. В листинге 8.3 показана командная строка для конфигурирования HTTP.SYS так, чтобы разрешить SSL-трафик через порт 8001. IP-адрес 0.0.0.0 означает «любой адрес». 40-разрядное шестнадцатеричное число – это цифровой отпечаток сертификата, установленного на данном компьютере. Получить отпечаток можно, открыв сертификат в оснастке Certificates консоли управления Microsoft и просмотрев детальные данные о нем. Указанный в конце GUID – это идентификатор приложения; он говорит о том, кто разрешил доступ. Годится любой сгенерированный вами GUID, который в дальнейшем будет ассоциироваться с вашим приложением.

Листинг 8.3. Конфигурирование HTTP.SYS для использования SSL на указанном порту с помощью netsh

```
netsh http add sslcert 0.0.0.0:8001
      1db7b6d4a25819b9aa09c8eaec9275007d562dcf
      {4dc3e181-e14b-4a21-b022-59fc669b0914}
```

После того как сертификат зарегистрирован в HTTP.SYS, вы можете сконфигурировать службу для шифрования по протоколу SSL. В листинге 8.4 показан конфигурационный файл службы, в котором задана привязка `basicHttpBinding`, шифрование на транспортном уровне, но без аутентификации клиента. Отметим, что для этой авторазмещаемой службы заданы два базовых адреса, один для зашифрованной, другой для открытой коммуникации. Это позволяет обращаться к оконечной точке MEX по незашифрованному каналу, а затем переходить на шифрованный. Если точка MEX не раскрывается или к ней допустимо обращаться по зашифрованному каналу, то второй адрес не нужен.

Листинг 8.4. Шифрование с привязкой basicHttpBinding

```
<system.serviceModel>
   <services>
      <service name="EffectiveWCF.StockService"
                  behaviorConfiguration="MyBehavior">
         <host>
            <baseAddresses>
               <add baseAddress="http://localhost:8000/EffectiveWCF" />
               <add baseAddress="https://localhost:8001/EffectiveWCF"/>
            </baseAddresses>
         </host>
         <endpoint address=""
                   binding="basicHttpBinding"
                   bindingConfiguration="MyBinding"
                   contract="EffectiveWCF.IStockService" />
         <endpoint address="mex"
                   binding="mexHttpBinding"
                   contract="IMetadataExchange" />
      </service>
   </services>
   <behaviors>
      <serviceBehaviors>
         <behavior name="MyBehavior">
            <serviceMetadata httpGetEnabled="true" />
         </behavior>
      </serviceBehaviors>
   </behaviors>
   <bindings>
      <basicHttpBinding>
         <binding name="MyBinding">
            <security mode="Transport">
               <transport clientCredentialType="None"/>
            </security >
         </binding>
      </basicHttpBinding>
   </bindings>
</system.serviceModel>
```

SSL поверх TCP

Как и в случае HTTP, коммуникации по протоколу TCP можно зашифровать с помощью SSL. Конфигурационные параметры безопасности на уровне транспорта TCP аналогичны HTTP. Достаточно внести всего три изменения в файл, представленный в листинге 8.4.

Во-первых, привязку `basicHttpBinding` для оконечной точки, отличной от MEX, нужно заменить на `netTcpBinding`. Во-вторых, в базовом адресе службы нужно указать URI, соответствующий схеме TCP, а не HTTP, то есть `net.tcp://{hostname}[:port]/{service location}`. В-третьих, весь раздел `basicHttpBinding` в элементе `bindings` следует заменить на `netTcpBinding`. Новая конфигурация показана в листинге 8.5.

Листинг 8.5. Шифрование с привязкой netTcpBinding

```
<system.serviceModel>
   <services>
      <service name="EffectiveWCF.StockService"
                  behaviorConfiguration="MyBehavior">
         <host>
            <baseAddresses>
               <add baseAddress="http://localhost:8000/EffectiveWCF" />
               <add baseAddress="net.tcp://localhost:8002/EffectiveWCF" />
            </baseAddresses>
         </host>
         <endpoint address=""
                   binding="netTcpBinding"
                   bindingConfiguration="MyBinding"
                   contract="EffectiveWCF.IStockService" />
         <endpoint address="mex"
                   binding="mexHttpBinding"
                   contract="IMetadataExchange" />
      </service>
   </services>
   <behaviors>
      <serviceBehaviors>
         <behavior name="MyBehavior">
            <serviceMetadata httpGetEnabled="true" />
         </behavior>
      </serviceBehaviors>
   </behaviors>
   <bindings>
      <netTcpBinding>
         <binding name="MyBinding">
            <security mode="Transport">
               <transport clientCredentialType="None"/>
            </security >
         </binding>
      </netTcpBinding>
   </bindings>
</system.serviceModel>
```

Аутентификация клиента

Чтобы аутентифицироваться, клиент представляет некие свидетельства, которым служба доверяет. Формат свидетельств может быть любым при условии, что и клиент, и служба его понимают и доверяют источнику.

Если клиент и служба разделяют общий секрет, например имя и пароль, и клиент посылает правильные верительные грамоты, служба верит, что клиент – именно тот, за кого себя выдает. Это механизм простой аутентификации, принятый в HTTP. В среде, состоящей только из компьютеров под управлением Windows, где клиенты и службы работают от имени учетных записей, определенных в Active Directory или в домене, между ними уже существуют доверительные отношения. В таком случае можно применить аутентификацию средствами Windows, используя систему Kerberos или NTLM-свертки. Если клиент и служба не входят в один и тот же домен Windows, то оптимальна аутентификация с помощью сертификатов, когда клиент посылает сертификат, выпущенный источником, которому служба доверяет.

Служба описывает требования к аутентификации клиента в атрибуте `clientCredentialType` элемента `transport`, если задан режим безопасности на транспортном уровне (`security mode="Transport"`). Делается это при конфигурировании привязки в конфигурационном файле или в коде. Для разных привязок допустимы различные схемы аутентификации клиента. В таблице 8.1 перечислены возможные варианты для готовых привязок.

Таблица 8.1. Аутентификация клиента в режиме безопасности на транспортном уровне

	Нет	Имя/пароль	Windows	Сертификат
`basicHttpBinding`	✓	✓	✓	✓
`wsHttpBinding`	✓	✓	✓	✓
`wsDualHttpBinding`	✓		✓	✓
`netTcpBinding`	✓		✓	✓
`netNamedPipeBinding`		✓		✓
`netMsmqBinding`	✓		✓	✓
`netPeerTcpBinding`		✓		✓
`msmqIntegrationBinding`	✓		✓	✓
`wsFederationHttpBinding`				

В режиме безопасности на транспортном уровне клиент, желающий аутентифицироваться, должен присоединить имеющиеся у него свидетельства к каналу до начала отправки сообщений, причем свидетельства должны соответствовать требованиям службы. Например, если в привязке к HTTP требуется простая аутентификация, то клиент должен послать имя и пароль. Если же в привязке задана аутентификация по сертификату, то клиент должен подписать сообщение своим закрытым ключом и отправить цифровой сертификат, выпущенный центром, которому служба доверяет (если у службы еще нет сертификата клиента).

Простая аутентификация и привязка basicHttpBinding

В листинге 8.4 показана конфигурация службы с привязкой `basicHttpBinding`, в которой безопасность на транспортном уровне обеспечивается за счет SSL-шифрования. Чтобы в этом случае задать имя и пароль, следует изменить значение атрибута `clientCredentialType` на `Basic`. В листинге 8.6 приведен фрагмент модифицированной конфигурации службы, которая требует аутентификации на транспортном уровне. Такая служба пригодна для обмена данными через Интернет, поскольку верительные грамоты передаются по защищенному каналу.

Листинг 8.6. Простая аутентификация в привязке basicHttpBinding

```
<basicHttpBinding>
   <binding name="MyBinding">
      <security mode="Transport">
         <transport clientCredentialType="Basic"/>
      </security >
   </binding>
</basicHttpBinding>
```

При использовании простой аутентификации клиент должен передать службе имя и пароль. Это можно сделать с помощью прокси-класса или путем прямых манипуляций с каналом. В листинге 8.7 показано, как передать верительные грамоты службе, для которой задана привязка `basicHttpBinding` и простая аутентификация.

Листинг 8.7. Передача клиентом имени и пароля

```
proxy.ClientCredentials.UserName.UserName = "MyDomain\\Me";
proxy.ClientCredentials.UserName.Password = "SecretPassword";
```

Аутентификация по имени и паролю (простая) годится, когда наличие общего секрета у клиента и службы допустимо, а ущерб от возможной компрометации не слишком велик. Поскольку пароли обычно записываются на бумажках, которые так и лежат на столе, или хранятся в базе данных или в конфигурационных файлах, то их легко скопировать или подсмотреть. Чтобы пароли не «застаивались», их надо почаще менять (кто не видел предупреждения «срок действия вашего пароля истекает через 10 дней»?), а это дополнительные сложности. Кроме того, людям свойственно использовать один и тот же пароль для разных учетных записей, поэтому компрометация одной ведет и к компрометации всех остальных.

Аутентификация по верительным грамотам Windows

Есть и другие схемы аутентификации, более безопасные, чем по коду и паролю. Если вы работаете в среде, где развернут каталог Active Directory, то можно пользоваться аутентификацией Windows. В этом случае верительной грамотой служит идентификатор пользователя или процесса. Это решение требует однократной регистрации, так как после входа пользователя в домен его идентификатор посылается службе автоматически, и код в листинге 8.7, уже не нужен. В листинге 8.8 показана привязка `net.tcp` с Windows-аутентификацией.

Листинг 8.7. Windows-аутентификация в привязке basicHttpBinding

```
<basicHttpBinding>
   <binding name="MyBinding">
      <security mode="Transport">
```

```
        <transport clientCredentialType="Windows"/>
      </security >
   </binding>
</basicHttpBinding>
```

Аутентификация по сертификату и привязка netTcpBinding

Цифровые сертификаты обеспечивают более надежную по сравнению с паролями аутентификацию. Когда нужна безопасная, быстрая, основанная на сертификатах связь, наиболее подходящей является привязка `netTcpBinding`. Сертификаты хорошо работают в смешанных моделях безопасности, которые встречаются в сложных интранет-сетях, где есть компьютеры под управлением Windows и UNIX, а аутентификация производится с помощью LDAP-каталогов, разработанных сторонними фирмами. Даже в Интернете привязка `netTcpBinding` может оказаться очень удобной для межсерверных коммуникаций, если существует возможность открыть некоторые порты на брандмауэре.

В листинге 8.9 показана конфигурация службы с безопасностью на транспортном уровне и аутентификацией клиента по сертификату. Стоит отметить несколько моментов. Во-первых, для запроса клиентского сертификата задано соответствующее значение атрибута `clientCredentialType` в привязке `netTcpBinding`. Во-вторых, в элементе `<serviceCredential>` прописан сертификат сервера. Это необходимо, чтобы сервер знал, какой сертификат и какую пару ключей использовать в процессе квитирования по протоколу SSL. В-третьих, поскольку атрибут `certificationValidationMode` равен PeerTrust, служба не будет проверять цепочку удостоверения клиентских сертификатов. Это необходимо при работе с сертификатами, которые были сгенерированы утилитой `MakeCert.exe`, а не получены от заслуживающего доверия центра.

Листинг 8.9. Аутентификация по сертификату в привязке NetTcpBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>

      <services>
         <service name="EffectiveWCF.StockService"
                  behaviorConfiguration="MyBehavior">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EffectiveWCF" />
                  <add baseAddress="net.tcp://localhost:8001/EffectiveWCF" />
               </baseAddresses>
            </host>
            <endpoint address=""
                      binding="netTcpBinding"
                      bindingConfiguration="MyBinding"
                      contract="EffectiveWCF.IStockService" />

            <endpoint address="mex"
                      binding="mexHttpBinding"
                      contract="IMetadataExchange" />
```



```
         </service>
      </services>

      <behaviors>
         <serviceBehaviors>
            <behavior name="MyBehavior">
               <serviceMetadata httpGetEnabled="true" />
                  <serviceCredentials>
                     <serviceCertificate findValue="localhost"
                                         storeLocation="LocalMachine" storeName="My"
                                         x509FindType="FindBySubjectName"/>
                     <clientCertificate>
                        <authentication certificateValidationMode="PeerTrust"/>
                     </clientCertificate>
                  </serviceCredentials>
            </behavior>
         </serviceBehaviors>
      </behaviors>

      <bindings>
         <netTcpBinding>
            <binding name="MyBinding">
               <security mode="Transport">
                  <transport clientCredentialType="Certificate"/>
                  <message clientCredentialType="None"/>
               </security>
            </binding>
         </netTcpBinding>
      </bindings>

   </system.serviceModel>
</configuration>
```

Чтобы начать обмен данными со службой, клиент должен представить свой сертификат. Сделать это можно в конфигурационном файле или в коде. В листинге 8.10 показан код клиента, который присоединяет свой сертификат к каналу, чтобы служба могла его аутентифицировать. В режиме доверия партнеру (PeerTrust) служба будет искать сертификат в папке Trusted People. Если найдет, то разрешит доступ, иначе откажет.

Листинг 8.10. Код клиента, аутентифицирующегося по сертификату

```
StockServiceClient proxy = new StockServiceClient();
proxy.ClientCredentials.ServiceCertificate.
                        Authentication.CertificateValidationMode =
                System.ServiceModel.Security.
                        X509CertificateValidationMode.PeerTrust;
proxy.ClientCredentials.ClientCertificate.SetCertificate(
                StoreLocation.CurrentUser,
                StoreName.My,
                X509FindType.FindBySubjectName,
                "MyClientCert");
try
```

```
{
   double p = proxy.GetPrice("msft");
   Console.WriteLine("Цена:{0}", p);
}
catch (Exception ex)
{
   Console.WriteLine("Message:{0}", ex.Message);
   if (ex.InnerException != null)
      Console.WriteLine("Inner:{0}", ex.InnerException.Message);
}
```

Идентификация службы

При установлении защищенного коммуникационного канала со службой клиент может аутентифицироваться различными описанными в этой главе способами: по имени и паролю, средствами Windows или по сертификату. Но не менее важно аутентифицировать саму службу. Если клиент собирается передавать секретные данные, то, не аутентифицировав службу, он рискует пасть жертвой распространенных в Интернете атак подлогом. Чтобы застраховаться от этого, WCF проверяет подлинность службы перед тем, как установить защищенный на транспортном уровне канал связи.

Когда оконечная точка MEX вызывается для получения WSDL-документа, она возвращает и опознавательные признаки службы. Если привязка поддерживает протокол WS-Security (а его поддерживают все готовые привязки, кроме `basicHttpBinding`), то в WSDL-документе будет содержаться информация, необходимая для идентификации службы. В зависимости от привязки и механизма аутентификации, эта информация может различаться.

Если утилита `svcutil` применяется для генерации клиентского прокси-класса и конфигурационного файла путем обращения к работающей службе, то опознавательные признаки службы записываются в конфигурационный файл. Во время выполнения проверяется, что клиент работает с ожидаемой службой, иначе WCF не установит защищенный канал.

В листинге 8.11 показан конфигурационный файл, который `svcutil` сгенерировала для службы с привязкой `wsHttpBinding` и аутентификацией клиента по сертификату в целях обеспечения безопасности на уровне сообщений. Обратите внимание, что в файл включен зашифрованный сертификат сервера. Если служба, с которой клиент попытается установить безопасный канал, не располагает таким сертификатом, то WCF возбудит исключение.

Листинг 8.11. Опознавательные признаки службы, сгенерированные для аутентификации по сертификату

```
<client>
   <endpoint address="http://localhost:8000/EffectiveWCF"
             binding="wsHttpBinding"
             bindingConfiguration="WSHttpBinding_IStockService"
             contract="IStockService"
             name="WSHttpBinding_IStockService">
```



```
      <identity>
         <certificate encodedValue=
➥"AwAAAAEAAAAUAAAAHbe21KJYGbmqCcjq7JJ1AH1WLc8gAAAAAQAAAL
➥0BAAAwggG5MIIBY6ADAgECAhAN4tyIi6rOqEYmrBcPIOHPMA0GCSqGS
➥Ib3DQEBBAUAMBYxFDASBgNVBAMTC1Jvb3QgQWdlbmN5MB4XDTA3MDky
➥ODIyMzgxOVoXDTM5MTIzMTIzNTk1OVowFDESMBAGA1UEAxMJbG9jYWx
➥ob3N0MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCwYocYLHnP+c
➥hgEurathCfwIxGhqL86lOdbEMuf1dfYjgkAhUBmkwVMhH8TVxyZkujp
➥09FKprT/beuzWmZVRjucaa/4yzOTsbrma0NGPI8V31Z+TYcU9zhNXn9
➥4d5eNH9Bc7QOhgwlRbl74Il8iPC9WGEB1V1PS/KYoYyCqNC8AwIDAQA
➥Bo0swSTBHBgNVHQEEQDA+gBAS5AktBh0dTwCNYSHcFmRjoRgwFjEUMB
➥IGA1UEAxMLUm9vdCBBZ2VuY3mCEAY3bACqAGSKEc+41KpcNfQwDQYJK
➥oZIhvcNAQEEBQADQQB+lCX4+Jk8L6NSQ8YjR51mkFSu6u3XZst/j5wq
➥gPIukU812/GEE4N/b8jXIXo6hyQqpvl9HKXnlTmYNivwXH2Q" />
      </identity>
   </endpoint>
</client>
```

В листинге 8.12 показан конфигурационный файл, который `svcutil` сгенерировала для службы с привязкой `wsHttpBinding` и Windows-аутентификацией клиента. Обратите внимание, что в файл включены верительные грамоты сервера. Если служба, с которой клиент попытается установить безопасный канал, работает от имени другой учетной записи Windows, то WCF возбудит исключение. Может случиться так, что в среде разработки служба, сгенерировавшая WSDL-документ, работала от имени одной учетной записи, а на промышленном сервере – от имени другой. Тогда конфигурационный файл на стороне клиента нужно будет изменить, так чтобы он соответствовал новым опознавательным признакам службы.

Листинг 8.12. Опознавательные признаки службы, сгенерированные для Windows-аутентификации

```
<client>
   <endpoint address="http://localhost:8000/EffectiveWCF"
             binding="wsHttpBinding"
             bindingConfiguration="WSHttpBinding_IStockService"
             contract="IStockService"
             name="WSHttpBinding_IStockService">
      <identity>
         <userPrincipalName value="MyDomain\Me"/>
      </identity>
   </endpoint>
</client>
```

Если клиент по какой-то причине не сможет проверить, что служба работает от имени заданной в конфигурационном файле учетной записи, то возбудит исключение. Например, если вы ведете разработку, отключившись от сети и не имея доступа к Active Directory, то клиент будет пытаться получить верительные грамоты службы, пока не истечет таймаут. Чтобы избежать такой ситуации, можете заменить в конфигурационном файле элемент `<userPrincipalName>` на `<servicePrincipalName>`, а значение `MyDomain\Me` атрибута `value` – на `host/localhost`.

Безопасность на уровне сообщений

В этом режиме для обеспечения конфиденциальности каждое сообщение шифруется и подписывается еще до передачи транспортному каналу. Прочитать сообщения сможет только сторона, знающая, как его расшифровать.

Иногда безопасность на уровне сообщений может обеспечить конфиденциальность в течение более длительного времени, чем безопасность на транспортном уровне. Типичный пример – наличие промежуточного посредника. Что если сообщение, адресованное службе, на самом деле проходит через посредника, который ставит его в очередь или маршрутизирует? На транспортном уровне можно гарантировать безопасность только до посредника, но не далее. За посредником клиент утрачивает контроль над конфиденциальностью, поскольку шифрование использовалось только на участке до него. Если же сообщение зашифровано, то посредник сможет прочитать только его заголовок, но не содержимое. И лишь конечный получатель, чьим открытым ключом было зашифровано сообщение, сможет расшифровать его своим закрытым ключом и получить доступ к содержимому. Следовательно, мы обеспечили конфиденциальность на всем пути от отправителя к получателю.

Как и на транспортном уровне, безопасность на уровне сообщений основывается на сертификатах стандарта X.509, хотя возможны и нестандартные реализации. На стороне службы должен быть установлен сертификат, чтобы клиент, начиная коммуникацию, мог послать зашифрованное сообщение. Это необходимо еще на этапе согласования параметров связи, чтобы верительные грамоты, если таковые потребуются, были защищены. По умолчанию большинство готовых привязок WCF, за исключением `basicHttpBinding` и `netNamedPipeBinding` применяют шифрование на уровне сообщений. Это гарантирует, что даже в режиме, подразумеваемом по умолчанию, коммуникация будет безопасной.

Аутентификация для привязки wsHttpBinding

В привязке `wsHttpBinding` используется безопасность на уровне сообщений. Сообщения, которыми клиент и служба обмениваются по протоколу HTTP, шифруются в соответствии со спецификацией WS-Security. Не требуется конфигурировать HTTP.SYS или IIS для поддержки SSL, поскольку WS-Security обеспечивает безопасную коммуникацию поверх любого протокола. А раз так, то оконечная точка службы и точка MEX могут располагаться на одном и том же порту, что сильно упрощает безопасное размещение в IIS. Потенциальный недостаток привязки `wsHttpBinding` состоит в том, что в ней используется порт 80, а не 443, предназначенный для SSL, поэтому могут возникнуть трудности с применением аппаратных ускорителей шифрования.

Привязка `wsHttpBinding` поддерживает многочисленные способы аутентификации клиента. По умолчанию применяется Windows-аутентификация, но возможны и другие, в том числе None, Basic и Certificate.

Windows-аутентификация

В листинге 8.13 показана конфигурация привязки `wsHttpBinding` для безопасной передачи на уровне сообщений. Отметим, что имеется только один базовый адрес, потому что в отличие от безопасности на транспортном уровне SSL-канал не нужен. По умолчанию `wsHttpBinding` на транспортном уровне использует Windows-аутентификацию. Поэтому такая конфигурация будет хорошо работать только в интранет-сетях, где клиент и служба входят в один и тот же домен. В Интернете и для коммуникации между Windows-машинами, не связанными отношением доверия, она не годится.

Листинг 8.13. Привязка wsHttpBinding с шифрованием и Windows-аутентификацией

```
<system.serviceModel>
   <services>
      <service name="EffectiveWCF.StockService"
               behaviorConfiguration="MyBehavior">
         <host>
            <baseAddresses>
               <add baseAddress="http://localhost:8000/EffectiveWCF" />
            </baseAddresses>
         </host>
         <endpoint address=""
                   binding="wsHttpBinding"
                   contract="EffectiveWCF.IStockService" />
         <endpoint address="mex"
                   binding="mexHttpBinding"
                   contract="IMetadataExchange" />
      </service>
   </services>
   <behaviors>
      <serviceBehaviors>
         <behavior name="MyBehavior">
            <serviceMetadata httpGetEnabled="true" />
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

Без аутентификации

Если вы вообще не хотите аутентифицировать клиента, задайте значение None атрибута `clientCredentialType`. В листинге 8.14 показана конфигурация привязки, в которой клиент не аутентифицируется.

Листинг 8.14. Привязка wsHttpBinding с шифрованием и без аутентификации клиента

```
<system.serviceModel>
   <services>
      <service name="EffectiveWCF.StockService"
```

```
            behaviorConfiguration="MyBehavior">
         <host>
            <baseAddresses>
               <add baseAddress="http://localhost:8000/EffectiveWCF" />
            </baseAddresses>
         </host>
         <endpoint address=""
                   binding="wsHttpBinding"
                   bindingConfiguration="MyBinding"
                   contract="EffectiveWCF.IStockService" />
      </service>
   </services>
   <bindings>
      <wsHttpBinding>
         <binding name="MyBinding">
            <security mode="Message">
               <transport clientCredentialType="None"/>
            </security >
         </binding>
      </wsHttpBinding>
   </bindings>
</system.serviceModel>
```

Аутентификация по сертификату

Применение сертификатов для аутентификации в привязке `wsHttpBinding` – основа создания безопасных приложений, работающих в Интернете. Конфигурируется этот режим примерно так же, как и другие схемы аутентификации с шифрованием сообщений.

В листинге 8.15 приведен конфигурационный файл службы, в которой используется аутентификация по сертификату. Следует отметить несколько моментов. Во-первых, атрибут `clientCredentialType` в привязке `wsHttpBinding` установлен так, что служба будет требовать от клиента предоставления сертификата. Во-вторых, в элементе `<serviceCredential>` прописан сертификат сервера. Это необходимо, чтобы клиент мог зашифровать сообщения открытым ключом сервера. В-третьих, служба сконфигурирована так, чтобы не проверять цепочку удостоверений клиентских сертификатов, поскольку атрибут `certificationValidationMode` равен PeerTrust. Без этого нельзя было бы работать с сертификатами, сгенерированными утилитой `MakeCert.exe`.

Листинг 8.15. Конфигурация службы, обеспечивающая аутентификацию клиента по сертификату

```
<system.serviceModel>
   <services>
      <service name="EffectiveWCF.StockService"
               behaviorConfiguration="MyBehavior">
         <host>
            <baseAddresses>
               <add baseAddress="http://localhost:8000/EffectiveWCF" />
            </baseAddresses>
         </host>
```



```
         <endpoint address=""
                   binding="wsHttpBinding"
                   bindingConfiguration="MyBinding"
                   contract="EffectiveWCF.IStockService" />
      </service>
   </services>

   <behaviors>
      <serviceBehaviors>
         <behavior name="MyBehavior">
            <serviceMetadata httpGetEnabled="true" />
            <serviceCredentials>
               <serviceCertificate findValue="localhost"
                        storeLocation="LocalMachine" storeName="My"
                        x509FindType="FindBySubjectName"/>
               <clientCertificate>
                  <authentication certificateValidationMode="PeerTrust"/>
               </clientCertificate>
            </serviceCredentials>
         </behavior>
      </serviceBehaviors>
   </behaviors>

   <bindings>
      <wsHttpBinding>
         <binding name="MyBinding">
            <security mode="Message">
               <message clientCredentialType="Certificate"/>
            </security>
         </binding>
      </wsHttpBinding>
   </bindings>
</system.serviceModel>
```

Для взаимодействия со службой, требующей аутентификации клиента по сертификату, клиент должен присоединять сертификат к каждому сообщению. Это можно сделать как в коде, так и в конфигурационном файле. В последнем случае файл, сгенерированный `svcutil`, необходимо модифицировать, включив в него сертификаты. Точнее, следует добавить поведение оконечной точки, в котором задан клиентский сертификат. Если используются сертификаты из не заслуживающего доверия источника, то в поведении нужно еще указать PeerTrust в качестве метода проверки сертификата. В листинге 8.16 приведен конфигурационный файл на стороне клиента, измененный так, чтобы сертификат присоединялся к сообщениям.

Листинг 8.16. Конфигурация клиента для аутентификации по сертификату

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <system.serviceModel>
      <bindings>
         <wsHttpBinding>
            <binding name="WSHttpBinding_IStockService"
```

```
            .
            .
            <security mode="Message">
               <transport clientCredentialType="Windows"
                          proxyCredentialType="None"
                          realm="" />
               <message clientCredentialType="Certificate"
                        negotiateServiceCredential="true"
                        algorithmSuite="Default"
                        establishSecurityContext="true" />
            </security>
         </binding>
      </wsHttpBinding>
   </bindings>
   <client>
      <endpoint address="http://localhost:8000/EffectiveWCF"
                binding="wsHttpBinding"
                bindingConfiguration="WSHttpBinding_IStockService"
                contract="IStockService"
                behaviorConfiguration="ClientCert"
                name="WSHttpBinding_IStockService">
         <identity>
            <certificate encodedValue=
➥"AwAAAAEAAAAUAAAAHbe21KJYGbmqCcjq7JJ1AH1WLc8gAAAAAQAAAL
➥0BAAAwggG5MIIBY6ADAgECAhAN4tyIi6rOqEYmrBcPIOHPMA0GCSqGS
➥Ib3DQEBBAUAMBYxFDASBgNVBAMTC1Jvb3QgQWdlbmN5MB4XDTA3MDky
➥ODIyMzgxOVoXDTM5MTIzMTIzNTk1OVowFDESMBAGA1UEAxMJbG9jYWx
➥ob3N0MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCwYocYLHnP+c
➥hgEurathCfwIxGhqL86lOdbEMuf1dfYjgkAhUBmkwVMhH8TVxyZkujp
➥09FKprT/beuzWmZVRjucaa/4yzOTsbrma0NGPI8V31Z+TYcU9zhNXn9
➥4d5eNH9Bc7QOhgwlRbl74Il8iPC9WGEB1V1PS/KYoYyCqNC8AwIDAQA
➥Bo0swSTBHBgNVHQEEQDA+gBAS5AktBh0dTwCNYSHcFmRjoRgwFjEUMB
➥IGA1UEAxMLUm9vdCBBZ2VuY3mCEAY3bACqAGSKEc+41KpcNfQwDQYJK
➥oZIhvcNAQEEBQADQQB+lCX4+Jk8L6NSQ8YjR51mkFSu6u3XZst/j5wq
➥gPIukU812/GEE4N/b8jXIXo6hyQqpvl9HKXnlTmYNivwXH2Q" />
         </identity>
      </endpoint>
   </client>

   <behaviors>
      <endpointBehaviors>
         <behavior name="ClientCert">
            <clientCredentials>
               <serviceCertificate>
                  <authentication certificateValidationMode="PeerTrust"/>
               </serviceCertificate>
            <clientCertificate
               findValue="MyClientCert"
               storeLocation="CurrentUser"
               storeName="My"
               x509FindType="FindBySubjectName"/>
            </clientCredentials>
         </behavior>
      </endpointBehaviors>
   </behaviors>
 </system.serviceModel>
</configuration>
```


Добавить в описание службы клиентский сертификат можно и с помощью кода. Клиент ищет сертификат в локальном хранилище и с помощью прокси-класса добавляет его в описание службы перед тем, как обратиться к ней. После этого WCF станет присоединять сертификат к каждому отправляемому службе сообщению. В листинге 8.10 приведен необходимый код. Когда `svcutil` генерирует конфигурационный файл клиента по данным, полученным от службы, не требующей аутентификации клиента, она вставляет в определение оконечной точки элемент `<identity>`. Этот элемент содержит сигнатуру службы, для которой был сгенерирован данный файл. Во время выполнения сигнатура сверяется с той, что хранится в конфигурационном файле. Если клиент попытается соединиться со службой, которая имеет другую сигнатуру, будет выведено сообщение об ошибке:

```
"The expected identity is 'identity(http://schemas.xmlsoap.org/ws/2005/05/
identity/right/possessproperty: http://schemas.xmlsoap.org/ws/2005/05/
identity/claims/thumbprint)' for the 'http://localhost:8000/EffectiveWCF'
target endpoint."

(Ожидается опознавательный признак ... для оконечной точки ...)
```

Обеспечение безопасности служб с помощью интегрированных в Windows средств

В этом разделе мы рассмотрим вопросы, возникающие при развертывании и потреблении служб внутри организации и вообще в среде, где компьютеры доверяют друг другу. Если к службе обращается другой компьютер в сети Windows-машин, то мы можем воспользоваться системой разделяемой аутентификации и авторизации, которая при развертывании в Интернете недоступна.

Поскольку мы работаем в локальной сети, никто не мешает применять привязку к TCP (`netTcpBinding`) или – если речь идет о коммуникациях в пределах одной машины – к именованным каналам (`netNamedPipeBinding`) для повышения производительности. Можно также задействовать такие механизмы повышения надежности, как MSMQ (с помощью привязки `netMsmqBinding`).

Описание демонстрационной среды

Примеры из этого раздела моделируют работу WCF-служб и их клиентов в локальной сети, отгороженной от внешнего мира корпоративным брандмауэром. Будем считать, что имеется библиотека классов, содержащих описание и реализацию контрактов, а служба и клиент реализованы в виде консольных приложений – `SampleHost` и `ClientConsole` соответственно. Топология сети изображена на рис. 8.2, где клиент, владелец службы и другие ресурсы (например, база данных) находятся за корпоративным брандмауэром, отделяющим внутреннюю сеть от Интернета.

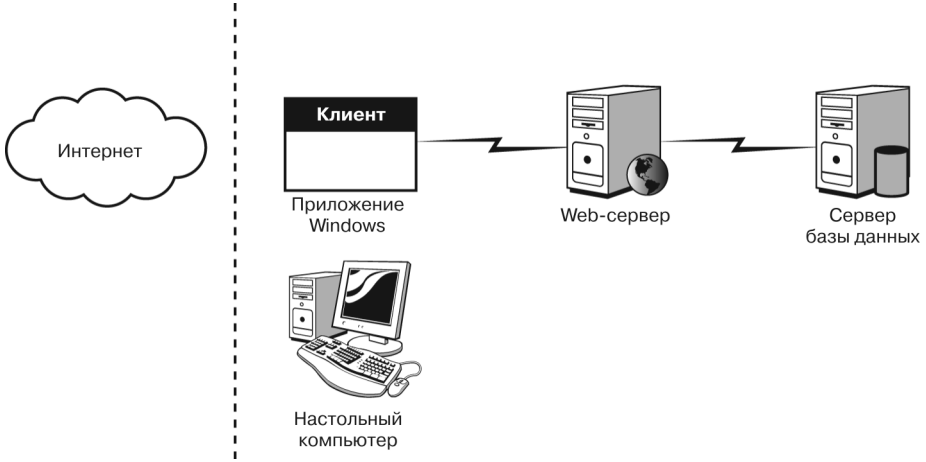

Рис. 8.2. Службы, развернутые в корпоративной локальной сети, где клиентами служат компьютеры под управлением Windows

Контракт службы `SampleService` описан в файле `ISamples.cs` и содержит три простых операции: `GetSecretCode`, `GetMemberCode` и `GetPublicCode`, как показано в листинге 8.17.

Листинг 8.17. Файл ISamples.cs, содержащий интерфейс контракта службы

```
using System.ServiceModel;

namespace SampleService
{
   [ServiceContract]
   public interface ISamples
   {
      [OperationContract]
      string GetSecretCode();
      [OperationContract]
      string GetMemberCode();
      [OperationContract]
      string GetPublicCode();
   }
}
```

Интерфейс `ISamples` реализован в файле `Samples.cs`, который приведен в листинге 8.18.

Листинг 8.18. Класс реализации службы Samples.cs

```
using System;
using System.Security.Principal;
using System.ServiceModel;
using System.Threading;

namespace SampleService
```



```
{
   public class Samples : ISamples
   {
      public string GetSecretCode()
      {
         DisplaySecurityDetails();
         return "The Secret Code";
      }

      public string GetMemberCode()
      {
         DisplaySecurityDetails();
         return "The Member-Only Code";
      }

      public string GetPublicCode()
      {
         DisplaySecurityDetails();
         return "The Public Code";
      }

      private static void DisplaySecurityDetails()
      {
         Console.WriteLine("Windows Identity = " +
➥WindowsIdentity.GetCurrent().Name);
         Console.WriteLine("Thread CurrentPrincipal Identity = " +
➥Thread.CurrentPrincipal.Identity.Name);
         Console.WriteLine("ServiceSecurityContext Primary Identity
➥= " + ServiceSecurityContext.Current.PrimaryIdentity.Name);
         Console.WriteLine("ServiceSecurityContext Windows Identity
➥= " + ServiceSecurityContext.Current.WindowsIdentity.Name);
      }
   }
}
```

Мы также создали две локальные учетные записи, на которые будем ссылаться в последующих примерах. Запустите консоль Computer Management (Управление компьютером) и раскройте узел Local Users and Groups (Локальные пользователи и группы). В ветви Users (Пользователи) создайте две учетные записи; мы назвали их Peter Admin (имя пользователя «peter») и Jessica Member (имя «jessica»).

Аутентификация пользователей средствами Windows

Сначала рассмотрим, как по умолчанию ведет себя служба, работающая по протоколу TCP, когда для аутентификации используются верительные грамоты Windows. Служба сконфигурирована с привязкой `netTcpBinding`, как показано в листинге 8.19. Мы разрешили публиковать метаданные для генерации прокси-класса.

Листинг 8.19. Конфигурация службы, работающей по протоколу TCP, с параметрами безопасности по умолчанию

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```
   <system.serviceModel>
      <services>
         <service behaviorConfiguration="ServiceBehavior"
                  name="SampleService.Samples">
            <endpoint address="" binding="netTcpBinding" name="netTcp"
                      contract="SampleService.ISamples" />
            <endpoint address="mex" binding="mexHttpBinding" name="mex"
                      contract="IMetadataExchange" />
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8080/Samples" />
                  <add baseAddress="net.tcp://localhost:8090/Samples" />
               </baseAddresses>
            </host>
         </service>
      </services>
      <behaviors>
         <serviceBehaviors>
            <behavior name="ServiceBehavior" >
               <serviceMetadata httpGetEnabled="true" />
            </behavior>
         </serviceBehaviors>
      </behaviors>
   </system.serviceModel>
</configuration>
```

Приложение ClientConsole просто создает экземпляр сгенерированного прокси-класса и последовательно вызывает каждую его операцию (листинг 8.20).

Листинг 8.20. Приложение ClientConsole, обращающееся к службе SampleService по протоколу TCP

```
using System;

namespace ClientConsole
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Для обращения к службе нажмите ENTER");
         Console.ReadLine();

         Samples.SamplesClient proxy =
                     new Samples.SamplesClient("netTcp");

         try
         {
            Console.WriteLine(proxy.GetPublicCode());
            Console.WriteLine(proxy.GetMemberCode());
            Console.WriteLine(proxy.GetSecretCode());
         }
         catch (Exception e)
         {
            Console.WriteLine("Исключение = " + e.Message);
         }

         Console.ReadLine();
      }
   }
}
```


Когда ClientConsole обращается к SampleHost, все вызовы завершаются успешно, и SampleHost выводит на консоль опознавательные признаки (методом DisplaySecurityDetails). Мы видим, что служба работает от имени пользователя, запустившего приложение ClientConsole. Этого следовало ожидать, так как никаких других аутентифицируемых субъектов мы еще не ввели.

Задание альтернативных аутентифицируемых субъектов

Генерируемые WCF прокси-классы предоставляют механизм задания альтернативных верительных грамот для передачи службе. В ряде ситуаций это бывает полезно. Например, если клиентское приложение поддерживает работу от имени нескольких пользователей, то верительные грамоты того или иного можно передать во время выполнения, чтобы служба знала, какие действия разрешено выполнять текущему пользователю.

В листинге 8.21 мы с помощью прокси-класса Samples.SamplesClient передаем имя и пароль ранее созданного пользователя «peter».

Листинг 8.21. Передача альтернативных верительных грамот с помощью сгенерированного клиентского прокси-класса

```
using System;
using System.Net;

namespace ClientConsole
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Для обращения к службе нажмите ENTER");
         Console.ReadLine();

         Samples.SamplesClient proxy = new Samples.SamplesClient("netTcp");

         proxy.ClientCredentials.Windows.ClientCredential =
                  new NetworkCredential("MACHINENAME\\peter", "p@ssw0rd1");

         try
         {
            Console.WriteLine(proxy.GetPublicCode());
            Console.WriteLine(proxy.GetMemberCode());
            Console.WriteLine(proxy.GetSecretCode());
         }
         catch (Exception e)
         {
```

```
            Console.WriteLine("Исключение = " + e.Message);
        }

        Console.ReadLine();
    }
  }
}
```

И теперь все три операции завершаются успешно, но метод `DisplaySecurityDetails` показывает, что владельцем по-прежнему является пользователь (`WindowsIdentity.GetCurrent().Name`), а служба работает от имени MACHINENAME\peter. WCF автоматически включила переданные клиентом верительные грамоты в контекст безопасности и сделала соответствующего пользователя владельцем потока.

Имена и пароли пользователей на других платформах. Можно задавать простые имена и пароли пользователей, работающих на платформах, отличных от Windows. Такую возможность поддерживают некоторые привязки, например `wsHttpBinding` (см. перечень возможностей привязок в таблице 8.1). Для этого безопасность на уровне сообщений необходимо сконфигурировать с параметром `clientCredentialType="UserName"`. Однако при этом WCF будет требовать включения безопасности на транспортном уровне (например, по сертификату), чтобы обеспечить конфиденциальность и целостность верительных грамот при передаче по сети. Сертификаты описаны в этой главе выше.

Авторизация пользователей средствами Windows

Мы показали, как идентифицировать пользователей по верительным грамотам Windows; теперь посмотрим, как в этой ситуации решается вопрос об определении разрешений (авторизации). Для начала воспользуемся стандартным атрибутом безопасности `PrincipalPermissionAttribute`. Этим атрибутом можно снабдить члены класса и тем самым разрешить или запретить к ним доступ со стороны клиентов.

Первым делом снабдим методы атрибутами, так чтобы доступ к `GetSecretCode` был разрешен только Питеру, а доступ к `GetMemberCode` – только Питеру и Джессике. Добавьте в определение `GetSecretCode` следующую строку (заменив MACHINENAME именем своего компьютера):

```
[PrincipalPermission(SecurityAction.Demand, Name = @"MACHINENAME\peter")]
```

а в определение `GetMemberCode` – такие строки:

```
[PrincipalPermission(SecurityAction.Demand, Name = @"MACHINENAME\peter")]
[PrincipalPermission(SecurityAction.Demand, Name = @"MACHINENAME\jessica")]
```

Запустите службу и клиента, не меняя значения Peter в свойстве `ClientCredentials`. Все три операции службы завершатся успешно. Теперь замените в коде клиента «peter» на «jessica», изменив также пароль (если необходимо). На этот раз при запуске клиента в методе `GetSecretCode` возникнет исключение из-за отказа в доступе.

Конечно, такой подход позволяет аутентифицировать и авторизовать известные учетные записи Windows для доступа к конкретным операциям службы. Но в большинстве промышленных систем необходим более простой способ ведения списков пользователей с разными правами.

У атрибута `PrincipalPermissionAttribute` есть также параметр `Role`, позволяющий задать группу Windows, а не имя отдельного пользователя. Прежде чем читать дальше, откройте консоль управления компьютером и временно создайте локальные группы Sample Admins и Sample Members. В первую поместите только пользователя Peter, а во вторую – пользователей Peter и Jessica. Теперь измените атрибуты, как показано в листинге 8.22.

Листинг 8.22. Разрешение доступа на основе ролей

```
using System;
using System.Security.Permissions;
using System.Security.Principal;
using System.ServiceModel;
using System.Threading;

namespace SampleService
{
   public class Samples : ISamples
   {
      [PrincipalPermission(SecurityAction.Demand,
                           Role="Sample Admins")]
      public string GetSecretCode()
      {...}

      [PrincipalPermission(SecurityAction.Demand,
                           Role="Sample Members")]
      public string GetMemberCode()
      {...}

      public string GetPublicCode()
      {...}

      private static void DisplaySecurityDetails() {...}
}
```

При запуске клиента должны получиться такие же результаты, как и раньше, но теперь для решения вопроса о том, каким пользователям разрешено обращаться к тем или иным операциям, мы опираемся на механизм групп Windows.

Использование встроенных групп Windows. Чтобы использовать в качестве ролей в атрибуте `PrincipalPermissionAttribute` стандартные группы Windows, добавьте перед именем группы слово BUILTIN вместо имени компьютера. Например, группа Administrators записывается как @«BUILTIN\Administrators», а группа Users – как @«BUILTIN\Users». Отметим также, что в языке C# символ @ отключает интерпретацию знака обратной черты как специального символа, без него нужно было бы включить две обратных черты, чтобы избежать ошибок компиляции.

Авторизация с использованием AzMan

Компонент Windows Authorization Manager (AzMan) предоставляет любым приложениям, в том числе и WCF, централизованный (и, стало быть, более удобный) механизм авторизации на основе политик, определенных в хранилищах авторизационных данных. В состав AzMan входит оснастка консоли MMC для управления хранилищами политик авторизации и связанными с ними уровнями доступа. Исполняющая среда Authorization Manager не зависит от физической организации хранилища, которое может находится в SQL Server, Active Directory, ADAM или XML-файле в зависимости от используемой операционной системы.

В этом разделе мы воспользуемся простым хранилищем политик авторизации в виде XML-файла для конфигурирования ролевого доступа к нашей службе. Для этого нам понадобится консоль MMC, в которую нужно включить оснастку Authorization Manager, выбрав из меню File пункт Add/Remove Snap-In (Добавить/удалить оснастку).

Для создания хранилища политик авторизации нужно находиться в режиме разработки (а не в режиме администратора), где имеется доступ ко всем функциям. В меню Action (Действия) выберите пункт Options (Параметры), а затем Developer Mode (Режим разработки). Щелкните правой кнопкой мыши по узлу Authorization Manager (Менеджер авторизации) и выберите пункт New Authorization Store (Новое хранилище политик авторизации). Откроется диалоговое окно, показанное на рис. 8.3.

Отметьте переключатель XML file, оставьте отмеченным переключатель Schema version 1.0 и введите имя и описание хранилища. В зависимости от операционной системы могут быть доступны еще хранилища Active Directory, ADAM и SQL Server.

Создав хранилище, выделите XML-файл, а затем щелкните правой кнопкой мыши и выберите из контекстного меню пункт New Application (Новое приложение). Назовите приложение `AzManDemo` и нажмите OK.

Чтобы определить роли, которым мы назначим разрешения, раскройте узел AzManDemo в левой панели, а затем узел Definitions (Определения). Щелкните правой кнопкой мыши по узлу Role Definitions (Определения ролей) и выберите из меню пункт New Role Definition (Определение новой роли). Мы создадим две роли: Member Role и Admin Role, но для последней нажмите еще кнопку Add (Добавить) в диалоговом окне New Role Definition, чтобы включить эту роль в состав определения роли Admin Role (рис. 8.4).

Чтобы приписать роль пользователям, щелкните правой кнопкой мыши по узлу Role Assignments (Назначение ролей) и выберите из меню пункт New Role Assignment (Новое назначение роли). Под узлом Role Assignments должны уже находиться роли Admin and Member. Поочередно щелкните по ним правой кнопкой мыши, выберите пункт Assign Users and Groups (Назначение пользователей и групп), затем From Windows and Active Directory (Из Windows и Active Directory). Включите учетную запись «Peter» в роль Admin Role, а запись «Jessica» в роль Member Role. Окончательная конфигурация показана на рис. 8.5.

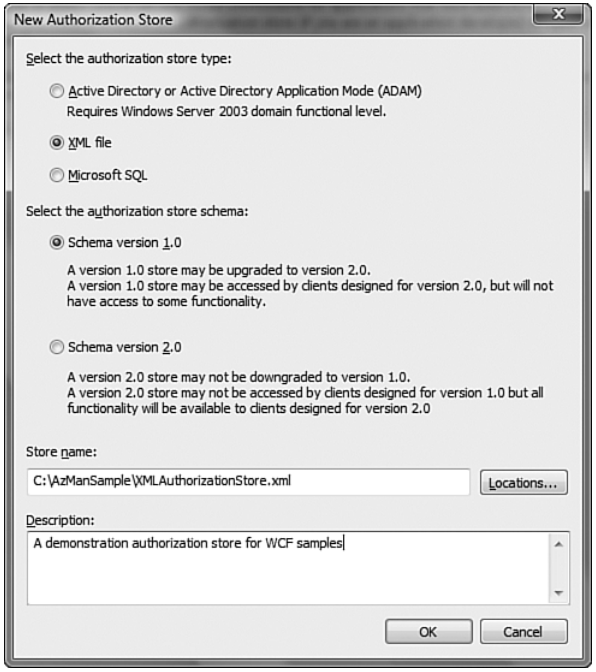

Рис. 8.3. Конфигурирование хранилища политик авторизации в XML-файле

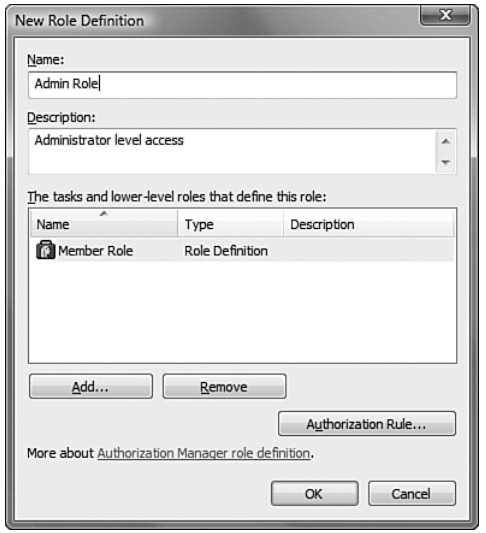

Рис. 8.4. Создание определения роли в менеджере авторизации

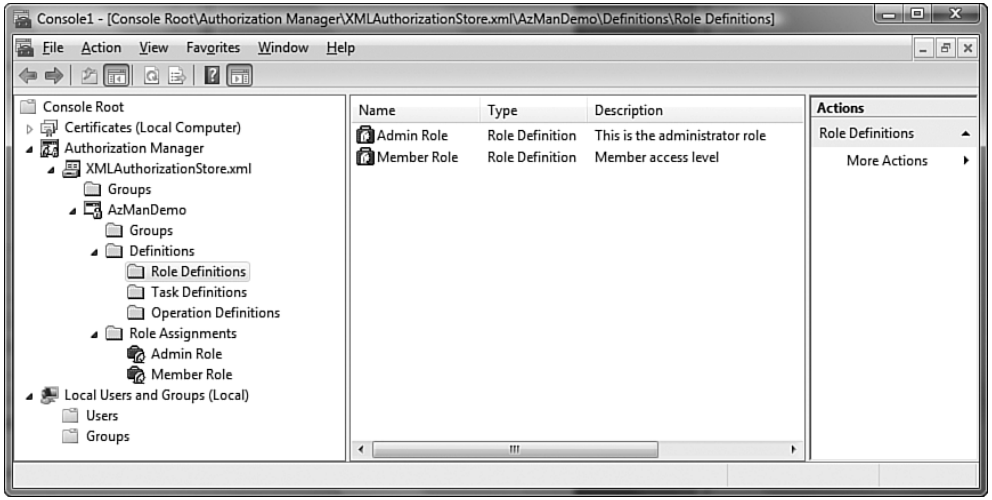

Рис. 8.5. Пример конфигурации менеджера авторизации

Сконфигурировав роли и приписав их пользователям, мы можем сообщить WCF о необходимости воспользоваться менеджером авторизации. Гибкость WCF в сочетании с возможностью доступа к исполняющей среде AzMan с помощью сборки AzRoles открывает перед нами целый ряд вариантов. Например, можно создать свой класс `ServiceAuthorizationManager` и вручную вызывать AzRoles для проверки того, разрешен ли роли доступ к операции. Но уже имеющаяся в ASP.NET 2.0 функциональность позволяет интегрировать AzMan и WCF с меньшими усилиями.

Система поставщиков информации о ролях в ASP.NET будет нам полезна, потому что WCF может автоматически интегрироваться с ее службами и потому что уже имеется встроенный класс `AuthorizationStoreRoleProvider`, умеющий работать с созданным нами хранилищем политик авторизации.

Чтобы воспользоваться менеджером авторизации, добавим элемент `<roleManager>` в секцию `<roleManager>` файла `App.config` приложения `SampleHost`. В описание поведения служб следует включить элемент `<serviceAuthorization>`, который позволит задействовать механизм ролей ASP.NET совместно с поставщиком `AuthorizationStoreRoleProvider`. Еще понадобится задать путь к XML-хранилищу политик авторизации в элементе `<connectionStrings>`. Описанные настройки показаны в листинге 8.23.

Листинг 8.23. Конфигурация службы для работы по протоколу TCP и интеграции с менеджером авторизации

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <connectionStrings>
      <add name="AuthorizationStore"
           connectionString="msxml://C:\AzManSample\XMLAuthorizationStore.xml"
      />
   </connectionStrings>

      <system.serviceModel>
         <services>
            <service behaviorConfiguration="ServiceBehavior"
                     name="SampleService.Samples">
               <endpoint address="mex" binding="mexHttpBinding"
                         bindingConfiguration=""
                 name="mex" contract="IMetadataExchange" />
               <endpoint address="" binding="netTcpBinding" bindingConfiguration=""
                 name="netTcp" contract="SampleService.ISamples" />
               <host>
                  <baseAddresses>
                     <add baseAddress="http://localhost:8080/Samples" />
                     <add baseAddress="net.tcp://localhost:8090/Samples" />
                  </baseAddresses>
               </host>
            </service>
         </services>
         <behaviors>
            <serviceBehaviors>
               <behavior name="ServiceBehavior" >
                  <serviceAuthorization
                        principalPermissionMode="UseAspNetRoles"
                        roleProviderName="AuthorizationStoreRoleProvider" />
                  <serviceMetadata httpGetEnabled="true" />
               </behavior>
            </serviceBehaviors>
         </behaviors>
      </system.serviceModel>

      <system.web>
         <roleManager
               defaultProvider="AuthorizationStoreRoleProvider"
               enabled="true"
               cacheRolesInCookie="true"
               cookieName=".ASPROLES"
               cookiePath="/"
               cookieProtection="All"
               cookieRequireSSL="false"
               cookieSlidingExpiration="true"
               cookieTimeout="30"
         >
            <providers>
               <clear />
               <add
                  name="AuthorizationStoreRoleProvider"
                  type="System.Web.Security.AuthorizationStoreRoleProvider"
                  connectionStringName="AuthorizationStore"
                  applicationName="AzManDemo" />
            </providers>
         </roleManager>
      </system.web>
</configuration>
```

Наконец, чтобы связать операции с группами, определенными в менеджере авторизации, изменим атрибут `PrincipalPermisionAttribute`, так чтобы он ссылался на определения ролей в хранилище. Измените названия созданных ранее групп Sample Admins и Sample Members на Admin Role и Member Role соответственно, чтобы согласовать их с именами в AzMan.

```
[PrincipalPermission(SecurityAction.Demand, Role="Admin Role")]
[PrincipalPermission(SecurityAction.Demand, Role="Member Role")]
```

При запуске клиента от имени каждого из пользователей Peter и Jessica мы снова увидим, что Питеру разрешен неограниченный доступ, а Джессика не может обращаться к методу `GetSecretCode`. Однако теперь доступ конфигурируется с помощью AzMan, и для задания ролей, пользователей, задач и операций в нашем распоряжении оказались удобные инструменты и хранилища политик авторизации, так что для настройки прав работы с приложением изменять код почти не придется.

Олицетворение пользователей

По умолчанию WCF-служба обращается к локальным и удаленным ресурсам, предъявляя верительные грамоты субъекта, от имени которого запущен ее владелец. На службу возлагается обязанность аутентифицировать клиента, а затем авторизовать его, проверив наличие прав доступа к ресурсам (само обращение к ним будет производиться от имени владельца службы). При запуске служб, получающих верительные грамоты Windows, у нас есть и еще одна возможность – *олицетворение* (impersonation).

Олицетворением называется процедура, в результате которой программа начинает исполняться от имени альтеративного субъекта. Служба может олицетворить клиента, «притворившись» им. Обычно это делается на протяжении одного вызова, но при желании служба может сохранить маркер олицетворения и воспользоваться им повторно. Потоку, в котором выполняется вызов, можно приписать идентификатор олицетворяемого субъекта, и тогда все операции будут выполняться с правами этого субъекта.

Олицетворение важно потому, что, притворившись на время клиентом, служба может получить только доступ к тем ресурсам, на которые у клиента есть права. При работе с привилегиями клиента проще гарантировать, что используются только те данные и ресурсы, которые этому клиенту доступны.

Олицетворение – это соглашение между службой и ее клиентами. Для олицетворения более высокого уровня требуется согласие клиента и в некоторых случаях разрешение со стороны системы, управляющей компьютером, на котором работает служба. Для начала сконфигурируем код службы, так чтобы поддерживалось олицетворение. Это делается с помощью атрибута `OperationBehaviorAttribute`, у которого есть параметр `Impersonation`. Значениями этого параметры могут быть члены перечисления `ImpersonationOption`. Пример см. в листинге 8.24.

Листинг 8.24. Заказ олицетворения с помощью атрибута OperationBehaviorAttribute

```
[OperationBehavior(Impersonation = ImpersonationOption.Required)]
public string GetSecretCode()
{
   DisplaySecurityDetails();
   return "The Secret Code";
}
```

Параметр `ImpersonationOption` может быть равен `NotAllowed`, тогда олицетворение отключено; `Required` – требуется, чтобы клиент был согласен на олицетворение (иначе обращение закончится с ошибкой); `Allowed` – олицетворение будет выполнено при получении согласия клиента, в противном случае работа продолжится без олицетворения. Последний режим необычен, его рекомендуется избегать.

Атрибут `OperationBehavior` можно задать только для определенных операций или разрешить олицетворение для всех вообще операций в конфигурационном файле. В листинге 8.25 показано, как используется параметр `ImpersonateCallerForAllOperations`, по умолчанию равный `false`.

Листинг 8.25. Разрешение олицетворения с помощью ImpersonateCallerForAllOperations

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service behaviorConfiguration="ServiceBehavior"
                  name="SampleService.Samples">
            <endpoint address="mex" binding="mexHttpBinding"
                      bindingConfiguration=""
              name="mex" contract="IMetadataExchange" />
            <endpoint address="" binding="netTcpBinding" bindingConfiguration=""
              name="netTcp" contract="SampleService.ISamples" />
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8080/Samples" />
                  <add baseAddress="net.tcp://localhost:8090/Samples" />
               </baseAddresses>
            </host>
         </service>
      </services>
      <behaviors>
         <serviceBehaviors>
            <behavior name="ServiceBehavior" >
               <serviceAuthorization
                  principalPermissionMode="UseWindowsGroups"
                  impersonateCallerForAllOperations="true" />
               <serviceMetadata httpGetEnabled="true" />
            </behavior>
         </serviceBehaviors>
      </behaviors>
```

```
   </system.serviceModel>
</configuration>
```

Олицетворение в коде. Выполнить олицетворение можно и программно. Объект `WindowsIdentity`, возвращаемый свойством `ServiceSecurityContext.Current`, имеет метод `Impersonate`, который можно вызвать, чтобы олицетворить пользователя. Но предварительно убедитесь, что `WindowsIdentity` не равен `null`.

Далее, клиент (в случае, когда необходимо полное олицетворение или делегирование) должен явно выразить свое согласие. Это тоже можно задать как в конфигурационном файле, так и в коде. Конфигурация должна выглядеть примерно так, как показано в листинге 8.26.

Листинг 8.26. Задание поддержки олицетворения в конфигурационном файле

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <behaviors>
         <endpointBehaviors>
            <behavior name="EndBehave">
               <clientCredentials>
                  <windows allowedImpersonationLevel="Impersonation" />
               </clientCredentials>
            </behavior>
         </endpointBehaviors>
      </behaviors>
      <bindings>
         <netTcpBinding>...</netTcpBinding>
      </bindings>
      <client>
         <endpoint address="net.tcp://localhost:8090/Samples"
               behaviorConfiguration="EndBehave"
               binding="netTcpBinding"
               bindingConfiguration="netTcp"
               contract="ClientConsole.Samples.ISamples"
               name="netTcp">
            <identity>...</identity>
         </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```

А в листинге 8.27 показано, как уровень олицетворения задается программно с помощью прокси-класса.

Листинг 8.27. Задание олицетворения с помощью клиентского прокси-класса

```
using System;
using System.Net;
using System.Security.Principal;

namespace ClientConsole
{
   class Program
   {
      static void Main(string[] args)
      {
         Samples.SamplesClient proxy = new Samples.SamplesClient("netTcp");
         proxy.ClientCredentials.Windows.AllowedImpersonationLevel =
               TokenImpersonationLevel.Delegation;

         ...
      }
   }
}
```


Свойство `AllowedImpersonationLevel` – неважно, задано оно в конфигурационном файле или в коде, – может принимать значения, определенные в перечислении `TokenImpersonationLevel`:

- **None**. Олицетворение не выполняется.
- **Anonymous**. Олицетворение применяется для проверки прав доступа, но служба не знает «личности» клиента. Допустимо только для локальных привязок, например `NetNamedPipeBinding`.
- **Identify**. Олицетворение не выполняется, но служба знает, кто ее вызвал, и может на основе этого решать, предоставлять ли доступ.
- **Impersonate**. Служба может идентифицировать клиента так же, как в режиме `Identify`, но еще и применяет олицетворение для доступа к ресурсам на той же машине.
- **Delegate**. То же, что `Impersonate`, но полученные верительные грамоты можно использовать и для доступа к сетевым ресурсам.

Будьте осторожны, применяя олицетворение и подумайте, что может случиться, если часть системы будет скомпрометирована. Например, если вы разрешаете делегирование (в данном случае с помощью конфигурационного файла и разрешений, хранящихся в Active Directory, который по умолчанию запрещает такие действия), и к вашей службе обращается пользователь с правами администратора домена, то в случае, если служба скомпрометирована, она может быть использована для доступа к произвольным сетевым ресурсам в домене с максимальными полномочиями. Очевидно, риск при этом велик, и вы должны тщательно продумать все последствия олицетворения и возможность отказать в доступе с помощью рассмотренного выше атрибута `PrincipalPermissionAttribute`.

Если вы приняли в расчет все риски, то олицетворение может стать эффективным инструментом управления доступом к ресурсам со стороны службы на основе тех разрешений, которые выданы ее клиентам.

Обеспечение безопасности служб, работающих через Интернет

В этом разделе мы поговорим о том, как защитить службы, работающие через Интернет. На рис. 8.6 показано приложение Windows, которое обращается к службам через Интернет. Из него становится понятна пропагандируемая Microsoft мо-

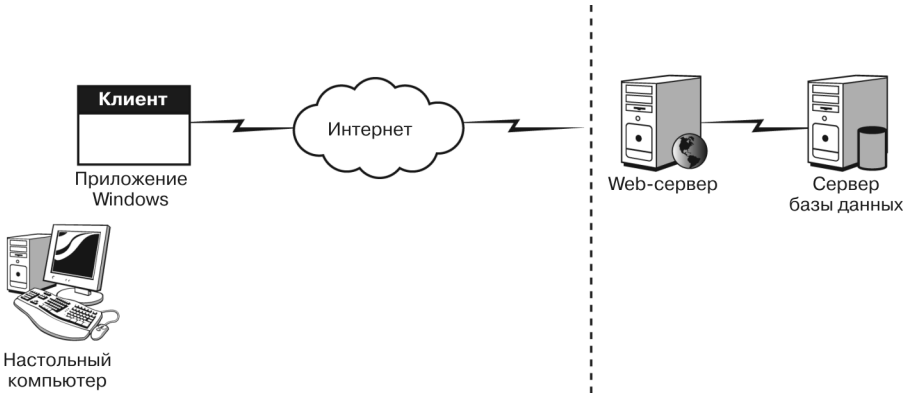

Рис. 8.6. Службы и Windows-приложения, обращающиеся к ним через Интернет

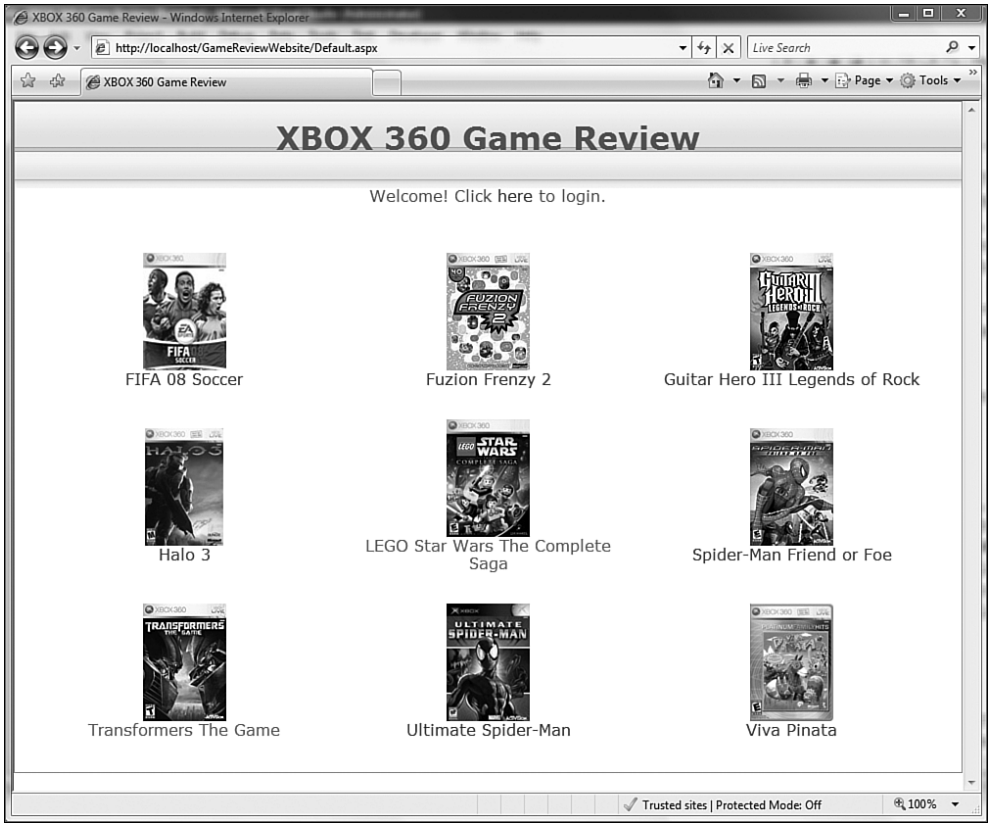

Рис. 8.7. Пример Интернет-приложения

дель ПО + службы, в которой клиентские приложения, работающие на настольном компьютере, обращаются по сети к различным службам. Для таких приложений нужны механизмы управления пользователями, заходящими из Интернета. Обычно имеется некая база данных, в которой хранятся имена, пароли и роли. Так делается по разным причинам, в том числе из соображений удобства управления учетными записями, сферы действия системы безопасности и простоты резервного копирования и восстановления. В ASP.NET 2.0 подобная возможность предоставляется в виде служб уровня приложения, в частности для управления членством и авторизации на основе ролей. WCF интегрирована с этими службами аутентификации и авторизации. Это означает, что разработчики могут пользоваться готовыми поставщиками, входящими в состав ASP.NET, для управления доступом к службам WCF.

Поскольку это Web-приложения, то мы рассмотрим вопрос об аутентификации с помощью форм и посмотрим, как служба, работающая с привязкой к протоколу HTTP, может аутентифицировать запросы с помощью кука.

Интеграция с ASP.NET

В ASP.NET и в WCF модели активации и размещения несколько различаются. WCF проектировалась для поддержки активации служб, работающих по различным транспортным протоколам, включая TCP, HTTP и MSMQ, тогда как ASP.NET предназначена прежде всего для активации по протоколу HTTP. Кроме того, WCF изначально поддерживает различные модели размещения, в том числе авторазмещение и размещение внутри IIS. Если владельцем является IIS, то WCF-служба может получать сообщения напрямую или работать в режиме совместимости с ASP.NET. По умолчанию она работает вместе с ASP.NET в одном и то же домене приложений (AppDomain). Все это позволяет WCF демонстрировать согласованное поведение при различных вариантах размещения и транспортных протоколах. Если вам это не нужно, а интерес представляет только протокол HTTP, то в режиме совместимости WCF может пользоваться некоторыми средствами ASP.NET, в частности объектом `HttpContext`, авторизацией по URL и расширяемостью за счет вставки модулей `HTTPModule`. В листинге 8.28 показано, как включить режим совместимости с ASP.NET в конфигурационном файле.

Листинг 8.28. Включение режима совместимости с ASP.NET (в файле web.config)

```
<system.serviceModel>
   <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
</system.serviceModel>
```

Служба может и сама сообщить, нужен ей режим совместимости с ASP.NET или нет. Для этого предназначен атрибут `AspNetCompatibilityRequirements-Attribute`. В листинге 8.29 приведен пример службы, для которой этот атрибут задан. Для краткости большая часть кода опущена.

Листинг 8.29. Атрибут AspNetCompatibilityRequirementsAttribute

```
namespace EssentialWCF
{
   [ServiceContract(Namespace="EssentialWCF")]
   [AspNetCompatibilityRequirements(RequirementsMode=
         AspNetCompatibilityRequirementsMode.Required)]
   public class GameReviewService
   {
      [OperationContract]
      [WebGet]
      public GameReview[] Reviews(string gameIdAsString)
      {
      }

      [OperationContract]
      [WebInvoke]
      [PrincipalPermission(SecurityAction.Demand, Role = "User")]
      public void AddReview(string gameIdAsString, string comment)
      {
      }
   }
}
```

Важно понимать, что режим совместимости с ASP.NET нужен не всегда. Например, ниже мы будем рассматривать использование поставщика информации о членстве для аутентификации в WCF-службах. Для этой функции режим совместимости с ASP.NET не обязателен. Но если вы хотите получить доступ к свойствам `Principal` и `Identity` класса `HttpContext` или воспользоваться другими относящимися к безопасности средствами, например, авторизацией доступа к файлу или URL, то без режима совместимости не обойтись. В этой ситуации WCF ведет себя подобно Web-службам ASP.NET, поскольку их возможности схожи. Отметим, что вы не должны применять режим совместимости с ASP.NET, если намереваетесь размещать службу вне IIS или пользоваться транспортами, отличными от HTTP.

Аутентификация с помощью поставщика информации о членстве

В ASP.NET 2.0 есть немало различных служб, в том числе членство, роли, профили и т.д. Это готовые каркасы, которыми разработчики могут пользоваться без написания дополнительного кода. Например, поставщик информации о членстве предоставляет средства для управления пользователями: создание, удаление, изменение и т.п. ASP.NET позволяет обращаться к этому поставщику из механизма аутентификации пользователей с помощью форм.

А WCF имеется аналогичный механизм аутентификации пользователей с помощью службы проверки членства. Применять его можно даже в отсутствие какого-либо Web-приложения ASP.NET, то есть входящая в состав ASP.NET служба проверки членства пригодна для аутентификации доступа к WCF-службе. Поскольку служба проверки членства предлагает собственные средства управления

пользователями, нам придется прибегнуть к маркерам UserName. В листинге 8.30 приведен пример привязки, в которой для аутентификации пользователей применяются маркеры UserName. Сами по себе эти маркеры не зашифрованы, поэтому для шифрования необходимо включить безопасность на уровне транспорта или сообщений. В данном случае мы шифруем их на транспортном уровне, так обычно и поступают. Важно отметить, что WCF настаивает на применении в этой ситуации шифрования, отказаться от него не получится.

Листинг 8.30. Использование имени и пароля в качестве верительных грамот (в файле web.config)

```
<system.serviceModel>
   <bindings>
      <wsHttpBinding>
         <binding name="MembershipBinding">
            <security mode="TransportWithMessageCredential">
               <message clientCredentialType="UserName"/>
            </security>
         </binding>
      </wsHttpBinding>
   </bindings>
</system.serviceModel>
```

Далее, чтобы воспользоваться службой проверки членства ASP.NET, необходимо сконфигурировать поведение службы, указав, что аутентификация пользователя по имени должна проводиться с помощью поставщика информации о членстве. Мы воспользуемся поставщиком `System.Web.Security.SqlMembershipProvider`, который хранит информацию о пользователях в базе данных SQL. В листинге 8.31 приведен пример такой конфигурации поведения.

Листинг 8.31. Конфигурация службы для работы с поставщиком SqlMembershipProvider (в файле web.config)

```
<system.serviceModel>
   <behaviors>
      <serviceBehaviors>
         <behavior name="ServiceBehavior">
            <serviceCredentials>
               <userNameAuthentication
                  userNamePasswordValidationMode="MembershipProvider"
                  membershipProviderName="AspNetSqlMembershipProvider"
               />
            </serviceCredentials>
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

В листинге 8.30 предполагается использование поставщика информации о членстве, принимаемого ASP.NET по умолчанию. А в листинге 8.32 показано, как поставщик информации о членстве по умолчанию сконфигурирован в файле `machine.config`.

Листинг 8.32. Конфигурация поставщика информации о членстве (в файле machine.config)

```
<system.web>
   <processModel autoConfig="true"/>
   <httpHandlers/>
   <membership>
      <providers>
         <add name="AspNetSqlMembershipProvider"
              type="System.Web.Security.SqlMembershipProvider,
➥System.Web, Version=2.0.0.0, Culture=neutral,
➥PublicKeyToken=b03f5f7f11d50a3a"
              connectionStringName="LocalSqlServer"
              enablePasswordRetrieval="false"
              enablePasswordReset="true"
              requiresQuestionAndAnswer="true"
              applicationName="/"
              requiresUniqueEmail="false"
              passwordFormat="Hashed"
              maxInvalidPasswordAttempts="5"
              minRequiredPasswordLength="7"
              minRequiredNonalphanumericCharacters="1"
              passwordAttemptWindow="10"
              passwordStrengthRegularExpression=""/>
      </providers>
   </membership>
</system.web>
```

Разработчики часто допускают ошибку, указывая для шифрования на транспортном уровне самоподписанный сертификат. WCF попытается проверить его и, потерпев неудачу, выдаст такое сообщение об ошибке:

```
Could not establish trust relationship for the SSL/TLS secure channel with
authority 'localhost'.

(Не могу подтвердить доверие к удостоверяющему центру 'localhost'
для организации безопасного SSL/TLS-канала)
```

Если посмотреть на вложенное исключение, окажется, что сертификат не удалось проверить. Первоначально было возбуждено такое исключение:

```
The remote certificate is invalid according to the validation procedure.

(Сертификат удаленной стороны не прошел процедуру проверки)
```

В листинге 8.3 показано, как заставить WCF принять сертификат, которой невозможно проверить, в том числе самоподписанный. Этот код должен был реализован клиентом, но только для целей тестирования на этапе разработки.

Листинг 8.33. Применение самоподписанного сертификата на этапе разработки

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
   System.Net.ServicePointManager.ServerCertificateValidationCallback
      += new System.Net.Security.RemoteCertificateValidationCallback(
      RemoteCertValidate);
}

static bool RemoteCertValidate(object sender,
                               X509Certificate cert, X509Chain chain,
                               System.Net.Security.SslPolicyErrors error)
{
   return true;
}
```


Авторизация по роли с использованием поставщика информации о ролях

В ASP.NET имеется механизм авторизации на основе принадлежности пользователя к определенной роли. В нем также используется модель поставщиков, которая абстрагирует детали хранения сведений о ролях от кода приложения. Имеется несколько готовых поставщиков информации о ролях: `SqlRoleProvider`, `WindowsTokenRoleProvider` и `AuthorizationStoreRoleProvider`. Поскольку мы говорим о приложениях, обращенных к Интернету, то рассмотрим только поставщика `SqlRoleProvider`. Для его использования в ASP.NET нужно предпринять несколько шагов. Во-первых, следует разрешить использование ролей. Это делается в файле `app.config` или `web.config` с помощью элемента `roleManager`.

```
<roleManager enabled ="true" />
```

Роли мы тем самым разрешили, но не указали, с каким поставщиком будем работать. Следующий шаг – конфигурирование поведения службы, в котором описывается поставщик информации о ролях. В листинге для элемента `serviceAuthorization` заданы атрибуты `principalPermissionMode` и `roleProviderName`. Атрибут `principalPermissionMode` говорит, как выполнять авторизацию. В данном случае мы задали значение `"UseAspNetRoles"`, то есть попросили использовать роли ASP.NET. Кроме того, указано имя поставщика.

Листинг 8.34. Авторизация доступа к службе с помощью ролей ASP.NET

```
<system.serviceModel>
   <behaviors>
      <serviceBehaviors>
         <behavior name="ServiceBehavior">
            <serviceAuthorization
               principalPermissionMode="UseAspNetRoles"
               roleProviderName ="AspNetSqlRoleProvider"
               />
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

Здесь мы выбрали принимаемого по умолчанию поставщика информации о ролях для SQL Server. А в листинге 8.35 показана стандартная конфигурация этого поставщика в файле `machine.config`.

Листинг 8.35. Конфигурация поставщиков информации о ролях (в файле machine.config)

```
<roleManager>
   <providers>
      <add name="AspNetSqlRoleProvider"
           connectionStringName="LocalSqlServer"
           applicationName="/"
           type="System.Web.Security.SqlRoleProvider,
➥System.Web, Version=2.0.0.0, Culture=neutral,
➥PublicKeyToken=b03f5f7f11d50a3a"
          />
      <add name="AspNetWindowsTokenRoleProvider"
           applicationName="/"
           type="System.Web.Security.WindowsTokenRoleProvider,
➥System.Web, Version=2.0.0.0, Culture=neutral,
➥PublicKeyToken=b03f5f7f11d50a3a"
      />
   </providers>
</roleManager>
```

В приложениях ASP.NET контроль доступа обычно производится путем вызова метода `User.IsInRole`. Такой подход хорошо работает для Web-страниц, которые открывают или закрывают доступ к некоторым своим частям в зависимости от результатов проверки, но для WCF-служб он непригоден. WCF производит авторизацию на уровне службы с помощью атрибута `PrincipalPermissionAttribute`. В листинге 8.36 приведен пример службы, для которой задана проверка разрешений. С помощью атрибута проверяется, принадлежит ли пользователь роли Administrator. Если нет, то ему не предоставляется доступ к службе.

Листинг 8.36. Атрибут PrincipalPermissionAttribute

```
namespace EssentialWCF
{
   [ServiceContract(Namespace="EssentialWCF")]
   [AspNetCompatibilityRequirements(RequirementsMode
                =AspNetCompatibilityRequirementsMode.Allowed)]
   public class GameReviewApprovalService
   {
      [OperationContract]
      [PrincipalPermission(SecurityAction.Demand,
         Role = "Administrator")]
      public void Approve(int gameReviewId, bool approved)
      {
      }

      [OperationContract]
      [PrincipalPermission(SecurityAction.Demand,
         Role = "Administrator")]
      public GameReview[] ReviewsToApprove()
      {
      }
   }
}
```


Аутентификация с помощью форм

До сих пор мы показывали, как можно обратиться через Интернет к службе, размещенной в Windows-приложении. На рис. 8.8 представлено Web-приложение, которое обращается к службам из браузера. Сейчас мы поговорим о том, как такие приложения могут вызывать WCF-службы. Для этого нам придется использовать стандартные приемы работы по протоколу HTTP, обеспечивающие безопасный доступ. Речь идет о применении HTTP-куков для аутентификации и SSL для шифрования. Протокол SSL мы уже рассматривали выше в этой главе, поэтому сейчас сосредоточимся на куках.

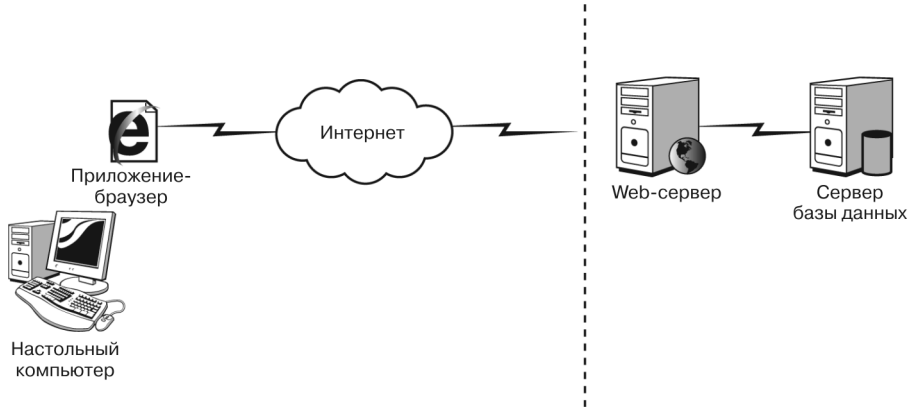

Рис. 8.8. Доступ к службам из Web-приложения через Интернет

В ASP.NET имеется механизм аутентификации с помощью форм (Forms Authentication), основанный на использовании HTTP-куков. Смысл его в том, что пользователь вводит свое имя и пароль в HTML-форму, которая затем отправляется на Web-сервер. Если пользователь успешно аутентифицирован, браузеру посылается кук, который в дальнейшем используется как маркер. При каждом следующем обращении к серверу браузер будет возвращать ему тот же кук, подтверждая, что пользователь уже прошел аутентификацию. По умолчанию механизм аутентификации с помощью форм работает напрямую со службой проверки членства ASP.NET, и их сочетание позволяет обезопасить Web-приложение почти без программирования. Для Web-приложений это замечательно, но в применении к WCF-службам бесполезно.

К сожалению, в данный момент WCF не интегрирована с механизмом Forms Authentication. Но это можно легко исправить. В листинге 8.37 показан нестандартный атрибут, позволяющий таки применять аутентификацию с помощью форм совместно с WCF-службой. Он устанавливает принципала в текущем контексте `HttpContext` таким же, как в текущем потоке. Атрибут совсем простой, но в результате контроль прав доступа с помощью атрибута `PrincipalPermissionAttribute` работает и с Forms Authentication.

Листинг 8.37. Атрибут UseFormsAuthentication

```
using System;
using System.Collections.ObjectModel;
using System.Data;
using System.Configuration;
using System.Security.Principal;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.Threading;
using System.Web;
using System.Web.Security;

namespace EssentialWCF
{
   public class UseFormsAuthentication : IDispatchMessageInspector
   {
      public UseFormsAuthentication()
      {
      }

      #region Члены IDispatchMessageInspector

      public object AfterReceiveRequest(ref Message request,
         IClientChannel channel,
         InstanceContext instanceContext)
      {
         IPrincipal currentUser =
         System.Web.HttpContext.Current.User;

         if ((currentUser is System.Web.Security.RolePrincipal) &&
             (currentUser != Thread.CurrentPrincipal))
             Thread.CurrentPrincipal = currentUser;

         return null;
      }

      public void BeforeSendReply(ref Message reply,
         object correlationState)
      {
      }

      #endregion
   }

   [AttributeUsage(AttributeTargets.Class)]
   public class UseFormsAuthenticationBehaviorAttribute : Attribute,
                                                          IServiceBehavior
   {
      #region Члены IServiceBehavior

      public void AddBindingParameters(
         ServiceDescription serviceDescription,
```



```
         ServiceHostBase serviceHostBase,
         Collection<ServiceEndpoint> endpoints,
         BindingParameterCollection bindingParameters)
      {
      }

      public void ApplyDispatchBehavior(
         ServiceDescription serviceDescription,
         ServiceHostBase serviceHostBase)
      {
         foreach (ChannelDispatcher channelDispatch in
               serviceHostBase.ChannelDispatchers)
         {
            foreach (EndpointDispatcher endpointDispatch in
               channelDispatch.Endpoints)
            {
            endpointDispatch.DispatchRuntime.MessageInspectors.Add(
         new UseFormsAuthentication());
            }
         }
      }

      public void Validate(ServiceDescription serviceDescription,
         ServiceHostBase serviceHostBase)
      {
      }

      #endregion
   }
}
```

В листинге 8.38 показана служба, в которой использован атрибут `UseFormsAuthentication`. Следует отметить, что этот атрибут предназначен только для работы в режиме совместимости с ASP.NET. Служба игрового обозрения GameReviewService, раскрывается с помощью новой привязки `webHttpBinding`. Она позволяет всем пользователям просматривать в браузере обзоры игр, но только аутентифицированные пользователи могут добавлять обзоры. Эта привязка раскрывает WCF-службы в стиле REST/POX и хорошо интегрируется с расширением ASP.NET AJAX Extensions. Подробнее об этом см. главу 13.

Листинг 8.38. Служба, в которой используется UseFormsAuthentication

```
using System;
using System.Data.Linq;
using System.Linq;
using System.Net;
using System.Security.Permissions;
using System.Security.Principal;
using System.ServiceModel;
using System.ServiceModel.Activation;
using System.ServiceModel.Web;
using System.Threading;

namespace EssentialWCF
```

```
{
   [UseFormsAuthenticationBehaviorAttribute]
   [ServiceContract(Namespace="EssentialWCF")]
   [AspNetCompatibilityRequirements(RequirementsMode
      =AspNetCompatibilityRequirementsMode.Required)]
   public class GameReviewService
   {
      public GameReviewService()
      {
      }

      [OperationContract]
      [WebGet]
      public GameReview[] Reviews(string gameIdAsString)
      {
         WebOperationContext wctx = WebOperationContext.Current;
         wctx.OutgoingResponse.Headers.Add(
         HttpResponseHeader.CacheControl,"no-cache");

         int gameId = Convert.ToInt32(gameIdAsString);
         GameReview[] value = null;

         try
         {
            using (GameReviewDataContext dc = new GameReviewDataContext())
            {
               var query =    from r in dc.GameReviews
                              where (r.GameID == gameId) &&
                                    (r.Approved)
                              orderby r.Created descending
                              select r;
               value = query.ToArray();
            }
         }
         catch
         {
            wctx.OutgoingResponse.StatusCode =
               System.Net.HttpStatusCode.InternalServerError;
         }

         return value;
      }

      [OperationContract]
      [WebInvoke]
      [PrincipalPermission(SecurityAction.Demand, Role = "User")]
      public void AddReview(string gameIdAsString, string comment)
      {
         string userName = Thread.CurrentPrincipal.Identity.Name;
         int gameId = Convert.ToInt32(gameIdAsString);
         bool  bAutomaticApproval =
               Thread.CurrentPrincipal.IsInRole("Administrator");

         using (GameReviewDataContext dc = new GameReviewDataContext())
         {
```



```
            dc.GameReviews.Add(new GameReview()
            { GameID = gameId,
              Review = comment, Approved = bAutomaticApproval,
              User = userName, Created = System.DateTime.Now });
            dc.SubmitChanges();
         }
      }
   }
}
```

Протоколирование и аудит

В этой главе вы узнали, как много существует вариантов конфигурирования безопасности WCF-служб и клиентских приложений. При таком разнообразии очень важно уметь диагностировать проблемы, связанные с аутентификацией и авторизацией. Кроме того, во многих отраслях, в частности в банках, здравоохранении, а также в компаниях, стремящихся соблюдать закон Сарбанеса-Оксли и другие законодательные акты, абсолютно необходимо наличие средств для аудита обращений к инфраструктуре подсистемы безопасности (как успешных, так и неудачных).

К счастью, WCF поддерживает конфигурируемый механизм создания протоколов и аудиторской отчетности обо всех действиях, касающихся безопасности служб.

Включить аудит безопасности можно, добавив в конфигурационный файл элемент `ServiceSecurityAuditBehavior`, как показано в листинге 8.39.

Листинг 8.39. Конфигурирование службы для аудита событий безопасности с помощью элемента ServiceSecurityAuditBehavior

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service behaviorConfiguration="ServiceBehavior"
                  name="SampleService.Samples">
            <endpoint address="" binding="netTcpBinding" name="netTcp"
                      contract="SampleService.ISamples" />
            <endpoint address="mex" binding="mexHttpBinding" name="mex"
                      contract="IMetadataExchange" />
            <host>...</host>
         </service>
      </services>
      <behaviors>
         <serviceBehaviors>
            <behavior name="ServiceBehavior" >
               <serviceSecurityAudit
                  auditLogLocation ="Application"
                  messageAuthenticationAuditLevel="SuccessOrFailure"
                  serviceAuthorizationAuditLevel="SuccessOrFailure"
                  suppressAuditFailure="false" />
               <serviceMetadata httpGetEnabled="true" />
```

```
            </behavior>
          </serviceBehaviors>
        </behaviors>
    </system.serviceModel>
</configuration>
```

Атрибут `auditLogLocation` определяет, какой журнал событий использовать для аудита: Default, Application или Security. Атрибуты `messageAuthenticationAuditLevel` и `serviceAuthorizationAuditLevel` могут принимать значения None (никогда), Success (в случае успеха), Failure (в случае неудачи) или SuccessOrFailure (в случае успеха или неудачи). Наконец, атрибут `suppressAuditFailure` можно задать равным `true`, чтобы система не возбуждала исключения при неудачной попытке записи сообщения в журнал.

Если запустить службу, сконфигурированную, как показано в листинге 8.39, то в журнал Application будут заноситься записи о событиях MessageAuthentication и ServiceAuthorization (в случае как успешной, так и неудачной попытки аутентификации или авторизации. В каждой записи будет храниться информация о «личности» обратившегося клиента, время, URI и протокол службы. Если по какой-то причине поместить запись в журнал не получится, система возбудит исключение.

Сочетая политику аудита с детальным протоколированием сообщений и средствами трассировки (см. главу 9), вы сможете эффективно и надежно отслеживать поведение и использование своих WCF-приложений.

Резюме

Нет сомнений, что безопасность – не самая простая функция приложения, особенно если оно состоит из нескольких программ, развернутых на разных машинах и даже в разных компаниях. Наказание за пренебрежение надлежащей политикой безопасности бывает суровым, а общественное доверие, утраченное в результате компрометации данных, трудно, а то и невозможно завоевать вновь. Поэтому так важно заранее продумывать, где, когда и как применять меры безопасности. Если обеспечена аутентификация как клиента, так и службы, то стороны могут быть уверены, что общаются с тем, с кем и ожидают. Авторизация позволяет проверить, можно ли дать клиенту доступ к запрашиваемой функции или данным напрямую или при посредничестве службы. И, наконец, конфиденциальность данных можно гарантировать с помощью шифрования, а целостность – применением цифровых подписей.

Вы видели, как для обеспечения безопасности на уровне транспорта и сообщений применяются сертификаты. Особенно они полезны для аутентификации и защиты передаваемых данных. Помимо сертификатов, есть и другие способы обеспечить безопасный обмен данными между службой и ее клиентами.

Было детально рассмотрено несколько сценариев раскрытия службы в интранет- и интернет-сетях. Это поможет вам решить, к какой категории относится

ваша система, и выбрать практическую основу для реализации. Наконец, мы показали, как задействовать встроенные в WCF механизмы аудита и протоколирования событий, относящихся к безопасности. Имея на руках протоколы, вы сможете быстро диагностировать возникающие проблемы, а также сохранить историю запросов на аутентификацию и авторизацию.

Хотя тема безопасности потенциально весьма сложна, WCF предлагает многообразные способы защиты служб, а также их потребителей. Иногда детали выглядят пугающе, особенно для тех, кто незнаком с теоретическими основами безопасности, но применение WCF позволяет свести большинство из них к заданию нескольких параметров в конфигурационном файле или в коде программы.

Глава 9. Диагностика

В предыдущих главах вы видели, что WCF предлагает многочисленные способы конфигурирования распределенных приложений и расширения среды за счет написанного вами кода. Добавьте сюда сложности взаимодействия между разными машинами и даже разными компаниями, и станет понятно, что неожиданности могут возникать в самых разных местах.

Отладка распределенных приложений – далеко не тривиальное занятие. Пусть даже у вас есть доступ ко всем процессам и ко всем таблицам символов, которые необходимы для пошагового выполнения программ через границы вызовов. Но ведь удаленные части системы могли быть написаны совсем другой командой, применяющей иные принципы кодирования. Трудности возникают и при попытке отфильтровать диагностическую информацию, относящуюся к одному потоку выполнения, например, найти все, касающееся сеанса одного пользователя, который обращался к различным службам на разных машинах.

Однако, сложность создания распределенной системы не ограничивается первоначальной разработкой. Необходимо еще позаботиться о простоте сопровождения в условиях промышленной эксплуатации. Администраторам необходимы средства эффективного поиска причин неполадок, чтобы можно было сообщить о них разработчикам.

К счастью, в WCF встроено немало диагностических инструментов, которые зачастую позволяют обнаружить причину проблемы, всего лишь включив несколько строк в конфигурационный файл. Ниже вы увидите, что WCF пользуется средствами трассировки и диагностики, которые уже есть в каркасе .NET Framework. А, стало быть, вы можете базироваться на том, что уже знаете, и интегрировать механизмы диагностики WCF с другими приложениями.

В этой главе мы покажем, как пользоваться средствами трассировки для перехвата событий WCF и как протоколировать информацию из передаваемых сообщений. Описываются прослушиватели трассировки и приводятся примеры того, как конфигурировать параметры для различных событий. Также мы рассмотрим поставляемый в комплекте с WCF инструмент Service Trace Viewer, который позволяет инспектировать данные, пересекающие границу службы.

Демонстрационное WCF-приложение

В этой главе мы будем иметь дело с приложением SelfHost, которое включено в Windows SDK. О том, как скачать, сконфигурировать и запустить этот пример, читайте в MSDN по адресу http://msdn2.microsoft.com/en-us/library/ms750530.aspx.

Если SDK у вас уже установлен, то приложение SelfHost вы найдете в папке Basic\Service\Hosting\SelfHost\; имеются версии как на C#, так и на VB.NET.

SelfHost – это пример начального уровня. Он состоит из двух проектов: простой консольной службы и консольного же клиентского приложения. Клиент несколько раз обращается к службе, и результаты выводятся в обеих консолях.

Трассировка

В основе диагностических средств WCF лежит механизм трассировки, предоставляемый самим каркасом .NET Framework. Пространство имен `System.Diagnostics` содержит классы, позволяющие приложениям выводить трассировочную информацию в разных форматах и в разные места.

Средства трассировки базируются на понятии источника и прослушивателя трассировки. *Источники трассировки* конфигурируются с помощью класса `System.Diagnostics.TraceSource` и позволяют приложению выводить детальную информацию о ходе выполнения: данные или события. Получать отправленные источником трассировочные данные могут один или несколько *прослушивателей*, то есть объектов классов, производных от `System.Diagnostics.TraceListener`.

С помощью этих средств WCF выводит информацию о действиях, произведенных в ходе обработки обращений к службе. Чтобы получить эту информацию, ничего специально программировать не надо; администратору достаточно сконфирурировать источник и прослушиватель, как описано ниже. Но разработчик может добавлять и собственную трассировку, если пожелает.

Сквозная трассировка

Центральный механизм мониторинга WCF-приложений называется *сквозной трассировкой* (end-to-end – E2E). Идея в том, чтобы использовать классы из пространства имен `System.Diagnostics` для передачи идентификаторов между различными частями распределенного приложения, в результате чего их действия можно скоррелировать и получить общую картину. Сквозная трассировка позволяет проследить всю последовательность действий, выполняемых разными службами на разных машинах, например, начиная с исходного запроса клиента через всю цепочку бизнес-логики к конечной службе.

Для сквозной трассировки применяется специальная XML-схема, позволяющая сохранить детали обработки при передаче через логические границы. XML-документ создается путем регистрации объекта типа `System.Diagnostics.XMLWriterTraceListener`, который представляет трассировочную информацию в формате E2E XML (определен на странице http://schemas.microsoft.com/2004/06/E2ETraceEvent).

В листинге 9.1 показан сокращенный фрагмент сквозной трассы в формате XML.

Листинг 9.1. Пример сквозной трассы

```
<E2ETraceEvent xmlns="...">
   <System xmlns="...">
```

```
      <EventID>131085</EventID>
         <TimeCreated SystemTime="2007-05-06T15:28:11.4178040Z" />
         <Source Name="System.ServiceModel" />
         <Correlation ActivityID="{7175a87f-b796-4f8a-a416-f2b284d4df39}" />
         <Execution ProcessName="Host.vshost" ProcessID="533" ThreadID="5"/>
         <Computer>LAERTES</Computer>
   </System>
   <ApplicationData>
      <TraceData><DataItem><TraceRecord>
         <Description>Activity boundary.</Description>
         <AppDomain>Host.vshost.exe</AppDomain>
         <ExtendedData>
            <ActivityName>Construct ServiceHost 'service'.</ActivityName>
            <ActivityType>Construct</ActivityType>
         </ExtendedData>
      </TraceRecord></DataItem></TraceData>
   </ApplicationData>
</E2ETraceEvent>
```

Обратите особое внимание на элемент `Correlation` и свойство `ActivityID`. Это ключ к объединению отдельных фрагментов трассы, формируемых различными источниками, в единый логический поток. Идеи, стоящие за корреляцией, описываются ниже.

Деятельности и корреляция

Деятельностью (activity) в WCF называется логическое подмножество функциональности, которое служит для группировки трассировочных записей с целью облегчить дальнейшую их идентификацию и мониторинг. В качестве примера можно привести обработку одного обращения к оконечной точке службы. Хотя деятельности полезны и сами по себе, для эффективного мониторинга необходим механизм, позволяющий отслеживать переходы от одной деятельности к другой.

Идея *корреляции* состоит в ассоциировании нескольких деятельностей для формирования логической цепочки операций в распределенном приложении. При корреляции учитываются *переходы* (transfer) от одной деятельности к другой в пределах одной оконечной точки и *распространение* (propagation), связывающее деятельности разных оконечных точек.

Деятельности коррелируются путем передачи *идентификатора деятельности* (activity ID). Это GUID, генерируемый объектом класса `System.Diagnostics.CorrelationManager`, который ассоциируется с трассой и может быть получен из статического свойства `System.Diagnostics.Trace.CorrelationManager`. У него есть два основных метода: `StartLogicalOperation()` и `StopLogicalOperation()`, которые связывают ассоциированные действия в логическую цепочку для целей трассировки.

Включение трассировки

По умолчанию трассировка отключена. Чтобы включить ее, необходимо сконфигурировать источник и прослушиватели трассировки, которые будут обрабатывать и сохранять трассировочную информацию.

В листинге 9.2 показаны части файла `SelfHost App.config`, имеющие отношение к трассировке.

Листинг 9.2. Включение трассировки в конфигурационном файле

```
<configuration>
   <system.serviceModel ... />
   <system.diagnostics>
      <sources>
         <source  name="System.ServiceModel" propagateActivity="true"
                  switchValue="Warning,ActivityTracing">
            <listeners>
               <add type="System.Diagnostics.DefaultTraceListener"
                    name="Default">
                  <filter type="" />
               </add>
               <add initializeData="app_tracelog.svclog"
                    type="System.Diagnostics.XmlWriterTraceListener"
                    name="tracelog" traceOutputOptions="Timestamp">
                  <filter type="" />
               </add>
            </listeners>
         </source>
      </sources>
   </system.diagnostics>
</configuration>
```

Здесь элемент `<source>` ссылается на источник трассировки `System.ServiceModel`, с помощью которого WCF выводит трассировочную информацию. В элемент `<listeners>` мы можем добавить один или несколько прослушивателей, которые будут получать и обрабатывать эту информацию. Свойство `type` обозначает класс прослушивателя, а свойство `initializeData` содержит передаваемые ему аргументы, например, местоположение файла. В данном случае `XmlWriterTraceListener` сконфигурирован для записи информации в файл `app_tracelog.svlog`.

Редактор конфигурации служб. Чтобы не скрывать детали работы подсистемы диагностики WCF, мы включаем трассировку и протоколирование вручную, задавая параметры в соответствующих конфигурационных файлах. Но позже мы покажем, как с помощью редактора конфигурации служб (Service Configuration Editor) можно быстро и безошибочно вносить такие изменения, не залезая в сами файлы.

У источника трассировки имеется свойство `switchValue`, которое позволяет задать уровень детализации. В таблице 9.1 перечислены возможные значения этого свойства.

Таблица 9.1. Допустимые значения свойства* `switchValue` *источника трассировки

Значение	Пояснение
`Off`	Отключает источник трассировки

Таблица 9.1. Допустимые значения свойства `switchValue` источника трассировки

Значение	Пояснение
`Critical`	Трассируются только самые серьезные ошибки приложения и среды, например, отказ службы или невозможность ее запуска
`Error`	Ошибки в логике приложения или среды, например, исключение, после которого невозможно восстановить нормальную работу
`Warning`	Ситуации, которые могут привести к исключению или отказу в будущем, или уведомления о том, что приложение восстановилось после исключения
`Information`	Детальная информация о системных событиях, которая может оказаться полезной для отладки, упрощенного аудита и других видов мониторинга
`Verbose`	Полная информация о каждом шаге обработки. Полезно для точного выявления источника проблемы
`ActivityTracing`	Использует корреляцию для отслеживания потока выполнения между логически связанными компонентами распределенного приложения

Отметим, что значение `ActivityTracing` можно объединять с селектором уровня детализации (например, `switchValue="Warning, ActivityTracing"`).

Рекомендации по выбору уровня детализации

Если задан высокий уровень детализации, то объем трассировочной информации может оказаться очень велик. Это увеличивает нагрузку на систему и осложняет задачу отделения зерен от плевел. Мы рекомендуем начинать диагностику с уровня трассировки `Warning`.

В обычных условиях эксплуатации оставляйте уровень `Critical` или `Error` до тех пор, пока не возникнет необходимость в получении более подробной информации для целей диагностики или мониторинга.

Протоколирование сообщений

Трассировка используется для вывода информации о потоке выполнения и отдельных действия различных компонентов распределенного приложения. Механизм же протоколирования сообщений предназначен для сохранения содержимого сообщений, которыми обмениваются клиент и служба. Этот механизм можно сконфигурировать так, чтобы перехватывались сообщения на уровне службы, на транспортном уровне или только неверно сформированные. Данные, собранные в ходе протоколирования сообщений, могут оказаться полезны в разных ситуациях – от диагностики до создания контрольного журнала использования службы.

Включение протоколирования сообщений

Как и трассировка, протоколирование сообщений основано на классах из пространства имен `System.Diagnostics` и по умолчанию выключено. Чтобы его включить, необходимо сначала добавить прослушиватель трассировки (например, `XMLWriterTraceListener`) для обработки информации от источника трассировки `System.ServiceModel.MessageLogging`.

В листинге 9.3 показано, как сконфигурировать приложение SelfHost для протоколирования сообщений.

Листинг 9.3. Включение протоколирования сообщений в конфигурационном файле

```
<configuration>
   <system.serviceModel>
      <services ... />
      <behaviors ... />
      <diagnostics>
         <messageLogging
            logEntireMessage="true"
            logMessagesAtServiceLevel="true"
            maxMessagesToLog="4000"/>
      </diagnostics>
   </system.serviceModel>

   <system.diagnostics>
      <sources>
         <source name="System.ServiceModel.MessageLogging">
            <listeners>
               <add name="messages"
                    type="System.Diagnostics.XmlWriterTraceListener"
                    initializeData="messages.svclog" />
            </listeners>
         </source>
      </sources>
   </system.diagnostics>
</configuration>
```

Секция `<system.diagnostics>` похожа на ту, что мы ранее использовали для включения трассировки. Мы добавили источник `System.ServiceModel.MessageLogging`, с помощью которого выводятся сообщения для протоколирования, и обрабатываем получаемую от него информацию тем же прослушивателем `XmlWriterTraceListener`, что и выше.

Но в отличие от трассировки формат и уровень детализации сообщений задаются в элементе `<messageLogging>`, добавленном в секцию `<system.diagnostics>`. В таблице 9.2 перечислены возможные атрибуты элемента `messageLogging`. Они могут употребляться в любом сочетании, а те, что опущены, принимают значения по умолчанию, также описанные в таблице 9.2.

Таблица 9.2. Атрибуты элемента messageLogging

Атрибут	Значение по умолчанию	Описание
`logEntireMessage`	false	Если true, протоколируется заголовок и тело сообщения; если false – только заголовок
`logMalformedMessages`	false	Протоколируются неправильно сформированные сообщения
`logMessagesAtServiceLevel`	false	Сообщения протоколируются в том виде, в каком были получены или отправлены самой службой
`logMessagesAtTransportLevel`	false	Сообщения протоколируются либо непосредственно перед кодированием транспортным уровнем, либо сразу после получения от транспортного уровня
`maxMessagesToLog`	10 000	Максимальное количество запротоколированных сообщений. Затем протоколирование приостанавливается
`maxSizeOfMessageToLog`	262 144	Максимальный размер протоколируемого сообщения в байтах. Если сообщение длиннее, оно игнорируется, а в трассу выводится предупреждение

Отметим, что сообщения, протоколируемые на транспортном уровне, могут быть зашифрованы в зависимости от параметров, заданных в привязке или конфигурационном файле.

Дополнительные конфигурационные параметры

В предыдущих разделах мы описали базовые возможности конфигурирования трассировки и протоколирования. А сейчас поговорим еще о нескольких параметрах, которые могут оказаться полезны для конфигурирования WCF-приложений.

Обобществление прослушивателей

Выше для каждого источника (трассировки или сообщений) мы задавали отдельный прослушиватель. Но можно сконфигурировать один общий прослушиватель и ассоциировать его с несколькими источниками, направляя весь вывод в одно место, например, в XML-файл. В листинге 9.4 показано, как можно записать трассировочную информацию и протокол сообщений в один и тот же выходной файл.

Листинг 9.4. Задание общего прослушивателя для трассы и протокола сообщений

```
<configuration>
   <system.serviceModel ... />
   <system.diagnostics>
      <sources>
         <source  name="System.ServiceModel" propagateActivity="true"
                  switchValue="Warning,ActivityTracing">
            <listeners>
               <add name="diagnostics" />
            </listeners>
         </source>
         <source name="System.ServiceModel.MessageLogging">
            <listeners>
               <add name="diagnostics" />
            </listeners>
         </source>
      </sources>
      <sharedListeners>
         <add name="diagnostics"
                  type="System.Diagnostics.XmlWriterTraceListener"
                  initializeData="diagnostics.svclog" />
      </sharedListeners>
   </system.diagnostics>
</configuration>
```

Для каждого источника добавляется прослушиватель с тем же именем, что у одного из общих прослушивателей. В данном случае общим является прослушиватель «diagnostics», который записывает трассу и сообщения в файл `diagnostics.svclog`.

Фильтры сообщений

По умолчанию протоколируются все сообщения, относящиеся к уровню, заданному в элементе `<messageLogging>`. Но чтобы уменьшить нагрузку на систему и размер файлов-протоколов, можно ограничиться лишь сообщениями, которые отвечают заданным вами критериям.

Фильтры сообщений – это выражения на языке XPath; сообщение протоколируется только, если вычисление каждого такого выражения дает true. Все прочие сообщения пропускаются, за исключением неправильно сформированных, на которые фильтры не распространяются. Фильтры задаются в элементе `<filters>`, вложенном в элемент `<messageLogging>`, как показано в листинге 9.5.

Листинг 9.5. Добавление фильтра для протоколирования сообщений

```
<configuration>
   <system.serviceModel>
      <services ... />
      <behaviors ... />
      <diagnostics>
         <messageLogging logMalformedMessages="true"
```

```
                        logMessagesAtTransportLevel="true">
            <filters>
               <add nodeQuota="1000"
                     xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
                     xmlns:wsa10="http://www.w3.org/2005/08/addressing">
/s12:Envelope/s12:Header/wsa10:Action[starts-with(text(),'http://
Microsoft.ServiceModel.Samples/ICalculator')]
               </add>
            </filters>
         </messageLogging>
      </diagnostics>
   </system.serviceModel>

   <system.diagnostics ... />
</configuration>
```

Этот пример может показаться сложным, но громоздкость ему придает лишь определение пространств имен, используемых в выражении XPath. Это пространства имен для конверта SOAP и схем адресации. В самом же выражении мы анализируем заголовок сообщения, чтобы понять, адресовано ли оно службе ICalculator, определенной в приложении SelfHost. Сообщения другим службам игнорируются.

Автоматический сброс источника трассировки

Если вы хотите, чтобы каждая операция трассировки или протоколирования сообщения сразу же записывалась на диск, то нужно включить режим автоматического сброса в элементе `<trace>`, который вложен в `<system.diagnostics>`, как показано в листинге 9.6.

Листинг 9.6. Включение режима автоматического сброса

```
<configuration>
   <system.serviceModel ... />
   <system.diagnostics>
      <sources ... />
         <trace autoflush="true" />
      </system.diagnostics>
</configuration>
```

По умолчанию автоматический сброс выключен. Прежде чем включать этот режим на промышленной системе, проверьте в тестовой среде, как он отразится на работе, поскольку нагрузка на систему при этом возрастает, особенно с увеличением интенсивности потока сообщений.

Счетчики производительности

Вместе с .NET Framework 3.0 устанавливаются три набора счетчиков, относящихся к WCF. В мониторе производительности они находятся в категориях ServiceModelService, ServiceModelEndpoint и ServiceModelOperation. Вы можете включить их для своего приложения в конфигурационном файле, как показано в листинге 9.7.

Листинг 9.7. Включение счетчиков производительности

```
<configuration>
   <system.serviceModel>
      <diagnostics performanceCounters="ServiceOnly">
         <messageLogging logMalformedMessages="true"
          logMessagesAtTransportLevel="true" />
      </diagnostics>
   </system.serviceModel>
</configuration>
```

Для включения счетчиков производительности необходимо добавить атрибут `performanceCounters` в элемент `<system.serviceModel><diagnostics>`. Допустимы значения `Off` (по умолчанию), `ServiceOnly` и `All`. Включать все счетчики имеет смысл на этапе разработки и для целей диагностики, но, поскольку обновление счетчиков обходится не даром, в промышленном режиме лучше ограничиться значением `ServiceOnly`, которое включает лишь счетчики в категории `ServiceModelService`.

Наблюдение за счетчиками производительности. Чтобы добавить счетчики в монитор производительности, необходим работающий экземпляр WCF-службы. Убедитесь, что счетчики включены в конфигурационном файле, запустите службу, добавьте счетчики, за которыми хотите понаблюдать, а потом запускайте клиента.

Windows Management Instrumentation (WMI)

WCF умеет раскрывать свои настройки и состояние через инструментарий управления Windows (Windows Management Instrumentation – WMI). Многие популярные приложения для администрирования и управления, например, Microsoft Operations Manager и HP OpenView пользуются WMI для доступа к различным компьютерам в сети предприятия. В оболочку Windows PowerShell также встроены средства работы с WMI, что позволяет писать сценарии для решения конкретных задач управления и мониторинга. Включить поставщика WMI для своего приложения можно в конфигурационном файле, как показано в листинге 9.8.

Листинг 9.8. Включение поставщика WMI

```
<configuration>
   <system.serviceModel>
      <diagnostics wmiProviderEnabled="true">
         <messageLogging logMalformedMessages="true"
          logMessagesAtTransportLevel="true" />
      </diagnostics>
   </system.serviceModel>
</configuration>
```

WMI включается так же, как счетчики производительности. Добавьте атрибут `wmiProviderEnabled` в узел `<system.serviceModel><diagnostics>`. После

включения административные приложения смогут отслеживать работу вашего WCF-приложения и управлять им.

Редактор конфигурации служб

До сих пор мы вручную изменяли конфигурационные XML-файлы, чтобы включить трассировку и протоколирование сообщений. На практике проще воспользоваться поставляемым в составе SDK редактором конфигурации служб, это поможет не наделать ошибок. Если Window SDK установлен, то эту программу вы найдете в меню **All Programs** (Все программы), Microsoft Windows SDK, Tools. Но из Visual Studio ее можно запустить и быстрее, достаточно щелкнуть правой кнопкой мыши по конфигурационному файлу и выбрать из контекстного меню пункт **Edit WCF Configuration** (Редактировать конфигурацию WCF).

Воспользуемся редактором для изменения файла `App.config` в проекте нашей службы. Щелкните правой кнопкой мыши по файлу `App.config` в проекте службы, выберите пункт меню **Edit WCF Configuration** и раскройте узел **Diagnostics** на панели **Configuration**.

Чтобы включить протоколирование сообщений и трассировку, щелкните по ссылкам Enable MessageLogging и Enable Tracing. В результате система будет сконфигурирована, как показано на рис. 9.1.

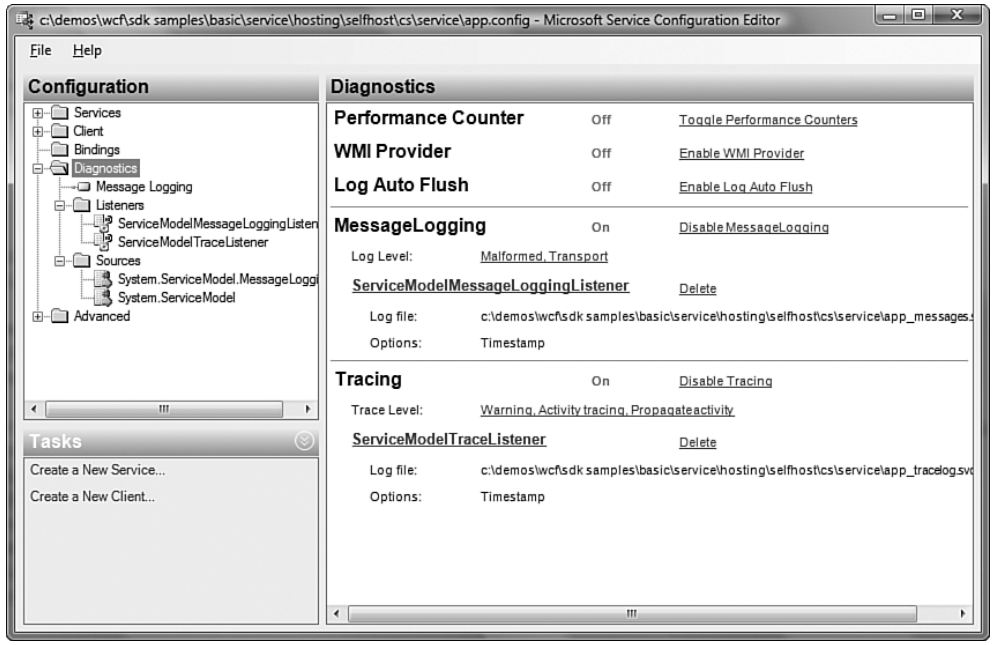

Рис. 9.1. После включения протоколирования сообщений и трассировки

Обратите внимание, что теперь в левой панели появились источники и прослушиватели. Щелкая по каждому из этих узлов, вы можете задать дополнительные параметры.

Параметры трассировки

Включив трассировку, изучите, какие можно задать для нее параметры. Для этого щелкните по ссылке справа от метки **Trace Level**.

Здесь можно включить трассировку и распространение деятельностей (см. выше в этой главе, обычно то и другое включают), а также уровень детализации – от **Off** до **Verbose**. Напомним, что от уровня детализации зависит размер файлов, в которые записываются трассировочные данные. В большом файле трудно найти нужную информацию, поэтому рекомендуем задавать минимально необходимый уровень.

Параметры протоколирования

Чтобы открыть диалоговое окно параметров протоколирования сообщений, щелкните по ссылке справа от метки **Log Level** в разделе **MessageLogging**. Как уже было сказано, можно задать три режима протоколирования: неправильно сформированные сообщения; сообщения, полученные или отправленные на уровне службы, и сообщения, передаваемые на транспортный уровень или получаемые от него.

Щелкнув по узлу **Message Logging**, который вложен в узел **Diagnostics** на левой панели, вы увидите дополнительные параметры протоколирования (рис. 9.2).

Эти параметры влияют на поведение источника `ServiceModel.MessageLogging`. Они напрямую отображаются на конфигурационный элемент `<messageLogging>`, назначение свойств которого описано в таблице 9.2.

Выше мы говорили, что можно задавать фильтры в виде выражений XPath, которые отбирают подлежащие протоколированию сообщения. В контекстном меню узла **Message Logging** есть пункт **New XPath Filter**, при выборе которого на панели задач можно будет задать параметры фильтра. На рис. 9.3 изображен тот же фильтр, который мы раньше задавали вручную.

Здесь можно задать максимальное число просматриваемых узлов XML-сообщения (атрибут `nodeQuota`). Тут же перечислены наиболее употребительные пространства имен и их префиксы. Если необходимо, вы можете определить дополнительные пространства имен и ссылаться на них с помощью префиксов из выражения XPath.

Конфигурирование источников

Включение трассировки и протоколирования в редакторе автоматически конфигурирует источники трассировки `System.ServiceModel` и `System.ServiceModel.MessageLogging`. Чтобы увидеть их параметры, раскройте узел **Diagnostics/Sources** на панели **Configuration**. Выбрав источник `System.ServiceModel`, вы увидите картину, изображенную на рис. 9.4.

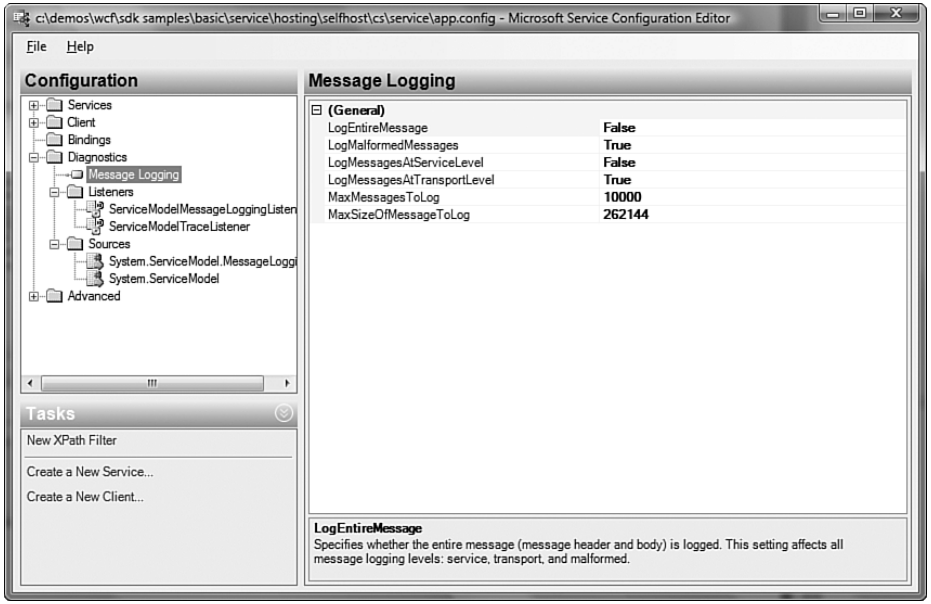

Рис. 9.2. Дополнительные параметры протоколирования сообщений

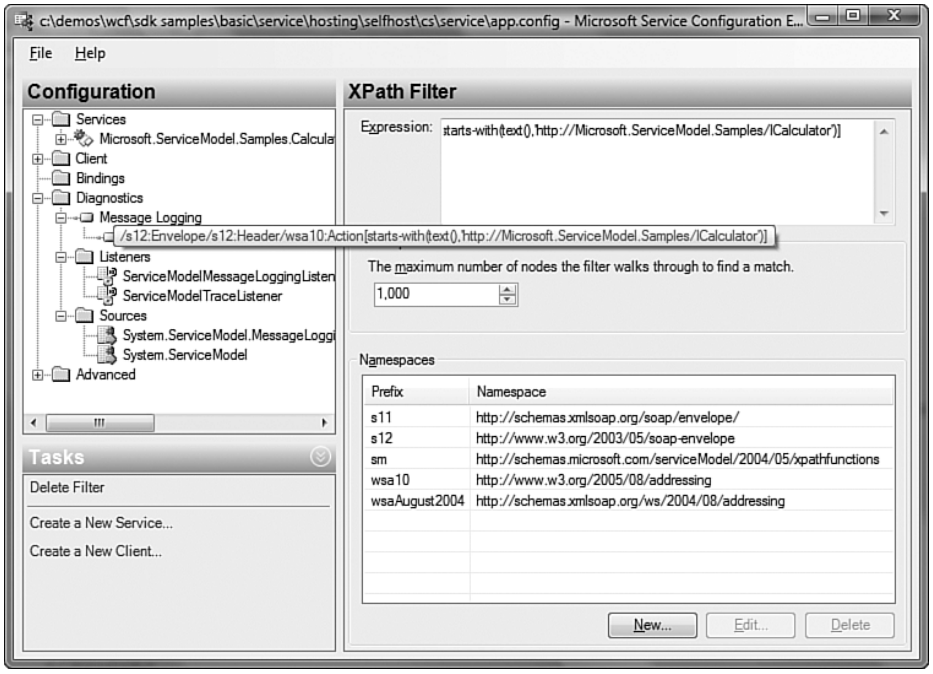

Рис. 9.3. Фильтр протоколирования сообщений в виде выражения XPath

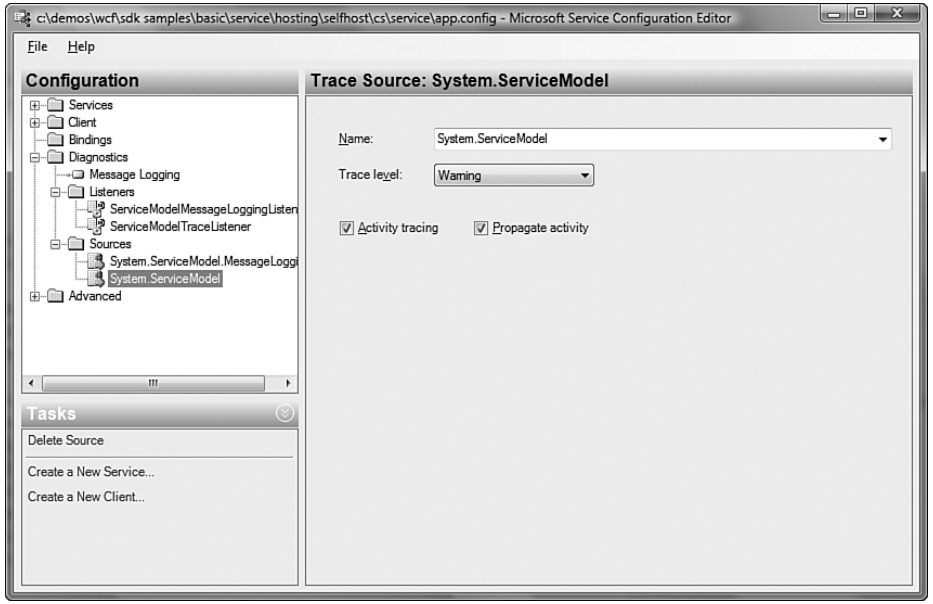

Рис. 9.4. Параметры источника трассировки

Здесь вы можете просмотреть и изменить уровень детализации, а также необходимость трассировки и распространения деятельностей. Отметим, что последнее доступно только для прослушивателей трассировки, но не для прослушивателей протоколирования сообщений.

Конфигурирование прослушивателей

Вернемся к главному окну диагностики (рис. 9.1). Щелкнув по имени прослушивателя (например, ServiceModelTraceListener), вы сможете увидеть все его параметры (рис. 9.5). С их помощью можно задать имя файла, в который прослушиватель записывает информацию, а также многочисленные детали, которые включаются в состав каждой трассировочной записи или протоколируемого сообщения.

Флажки соответствуют членам перечисления `System.Diagnostics.TraceOptions`. Из них наиболее интересны **Timestamp** (временной штамп), **Process ID** (идентификатор процесса), **Thread ID** (идентификатор потока), **Callstack** (стек вызовов) и **DateTime**. Флажок **Logical Operation Stack** (стек логических операций) позволяет включить корреляционный «стек» трассы, то есть скореллированную историю выполнения операций, которая необязательно совпадает со стеком вызовов.

Чтобы увидеть сводку параметров прослушивателя, раскройте узел Listeners на панели Configuration и щелкните по имени интересующего вас прослушивателя. Результат показан на рис. 9.6.

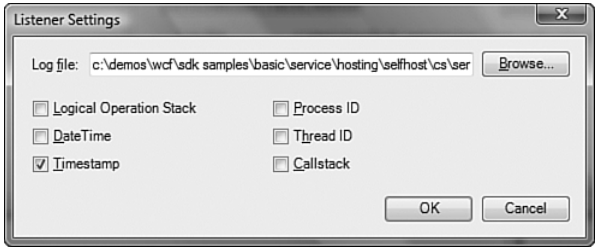

Рис. 9.5. Параметры прослушивателя

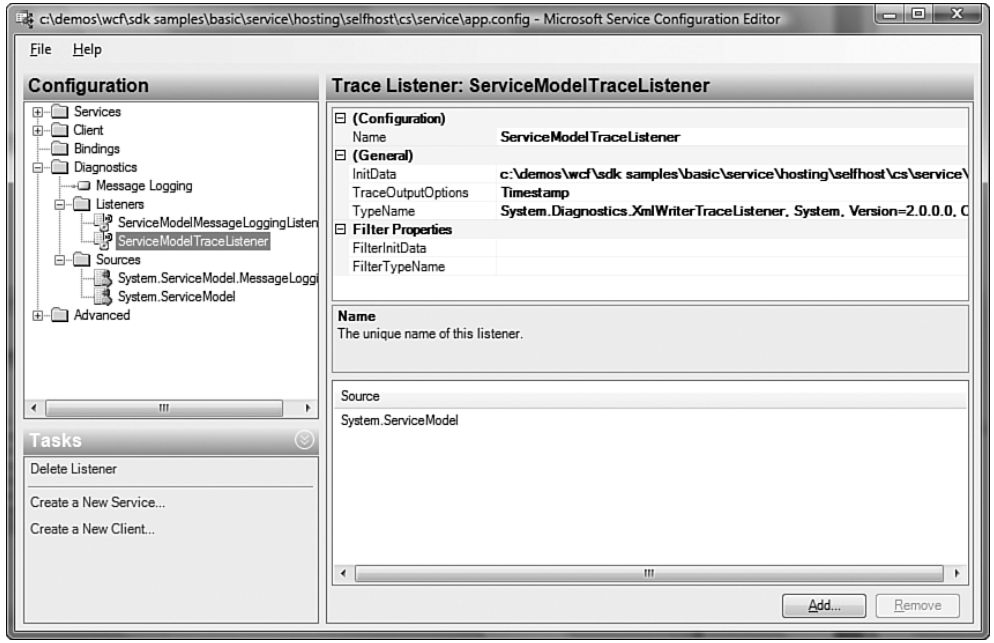

Рис. 9.6. Подробная конфигурация прослушивателя

Здесь вы видите сразу все параметры прослушивателя и можете изменить любой из них.

Инструмент просмотра трассы службы

Мы рассказали, как настраиваются различные аспекты трассировки и протоколирования сообщений. Но как эффективно воспользоваться этими диагностическими механизмами? Ведь даже за короткое время объем собранной информации может оказаться весьма велик.

В комплекте с WCF поставляется замечательный инструмент для анализа диагностических протоколов, который называется Service Trace Viewer. Им можно

воспользоваться для импорта трассы и протоколов сообщений, сформированных одним или несколькими компонентами распределенного приложения. После установки Windows SDK вы найдете программу Service Trace Viewer в меню **All Programs**, Microsoft Windows SDK, Tools.

Конфигурирование приложения SelfHost. Для этого раздела мы включили трассировку и протоколирование сообщений как для клиента, так и для службы. Чтобы сделать то же самое на своем компьютере, откройте редактор конфигурации службы и включите в обоих проектах трассировку и протоколирование сообщений, задав уровень детализации Information и оставив для остальных параметров значения по умолчанию. Закончив конфигурирование, запустите приложение, чтобы сгенерировать файлы протоколов для клиента и службы.

Воспользуемся программой Service Trace Viewer для анализа протоколов, сгенерированных приложением SelfHost. Запустив программу, выберите из меню **File** (Файл) пункт **Open** (Открыть). Найдите каталог SelfHost/client и выберите файлы трассировки (`app_trace.svclog`) и протокола сообщений (`messages.svclog`), удерживая клавишу **Shift** при щелчке по каждому из них. Обязательно выберите сразу оба файла, потому что последующая команда приведет к стиранию ранее загруженной информации. (Позже мы научимся пользоваться командой **File ⇒ Add** для слияния нескольких файлов.)

Режим просмотра деятельностей

Программа Service Trace Viewer умеет объединять несколько трасс и протоколов. На рис. 9.7 показано, как выглядит ее окно после загрузки файлов, сгенерированных клиентом службы SelfHost.

Здесь вы видите, как отображаются объединенные результаты по умолчанию. В левой панели находится список всех деятельностей, и для каждой показаны число трассировочных записей, продолжительность, время начало и окончания. Если выбрать одну или несколько деятельностей, то в правой верхней панели вы увидите отдельные трассировочные записи, относящиеся к выбранным деятельностям.

Предупреждения и исключения. Service Trace Viewer отображает деятельности, содержащие записи с предупреждениями, желтым цветом. Если же имеется запись об исключении, то деятельность отображается красным цветом.

Первая деятельность, 000000000000, – это специальная корневая деятельность, с которой связаны все остальные. Двигаясь вниз по списку, мы видим, какие деятельности обрабатывал клиент по ходу работы программы. Сначала был сконструирован и открыт объект `ChannelFactory`, что позволило начать взаимодействие со службой.

Каждый вызов службы отображается как деятельность Process action. В нашей трассе четыре таких вызова, соответствующие операциям `Add`, `Subtract`,

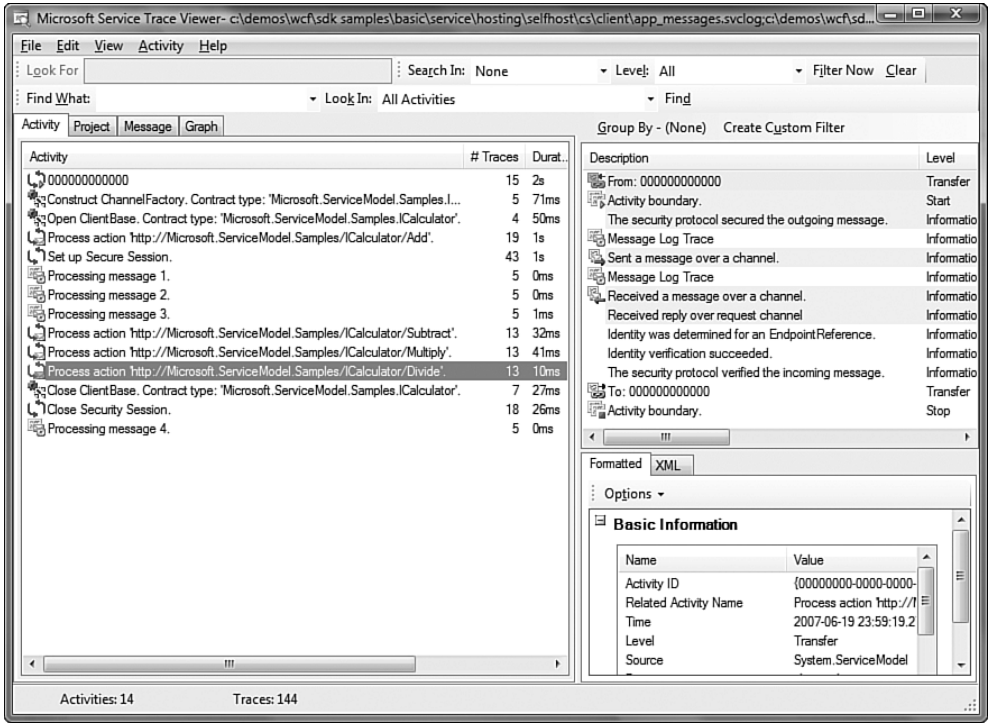

Рис. 9.7. Окно программы Service Trace Viewer после загрузки трассы и протокола сообщений, сгенерированных клиентом

`Multiply` и `Divide`. Клиент также вел переговоры об организации безопасного сеанса (Set Up Secure Session) в соответствии с требованиями привязки к службе.

Пощелкайте по разным деятельностям и посмотрите в левой панели, какие трассировочные записи им соответствуют. Показываются тип и краткое описание записи. Ниже мы познакомимся с другим способом просмотра этих записей.

Режим просмотра проекта

Для перехода в режим просмотра проекта щелкните по вкладке **Project** в левой панели. Проект в понимании программы Service Trace Viewer позволяет задать несколько файлов трасс и протоколов, которые следует загружать при открытии проекта. Особенно это полезно, когда есть несколько участников (например, клиент вызывает несколько служб), которых вы хотите отлаживать вместе. Загрузив интересующие вас файлы, выберите из меню **File** пункт **Save Project** (Сохранить проект).

В режиме просмотра проекта отображаются файлы, ассоциированные с текущим проектом. Вы можете создавать и модифицировать проекты, а также добавлять и удалять ассоциированные с проектом файлы.

Режим просмотра сообщений

В этом режиме выводится список всех запротоколированных сообщений независимо от скоррелированных деятельностей. Это полезно, когда нужно быстро найти конкретное сообщение, например посланное операции `Multiply`, и просмотреть его содержимое.

На рис. 9.8 показано окно программы в режиме просмотра сообщений, где сообщение, посланное клиентом для вызова операции службы `Divide`, выделено.

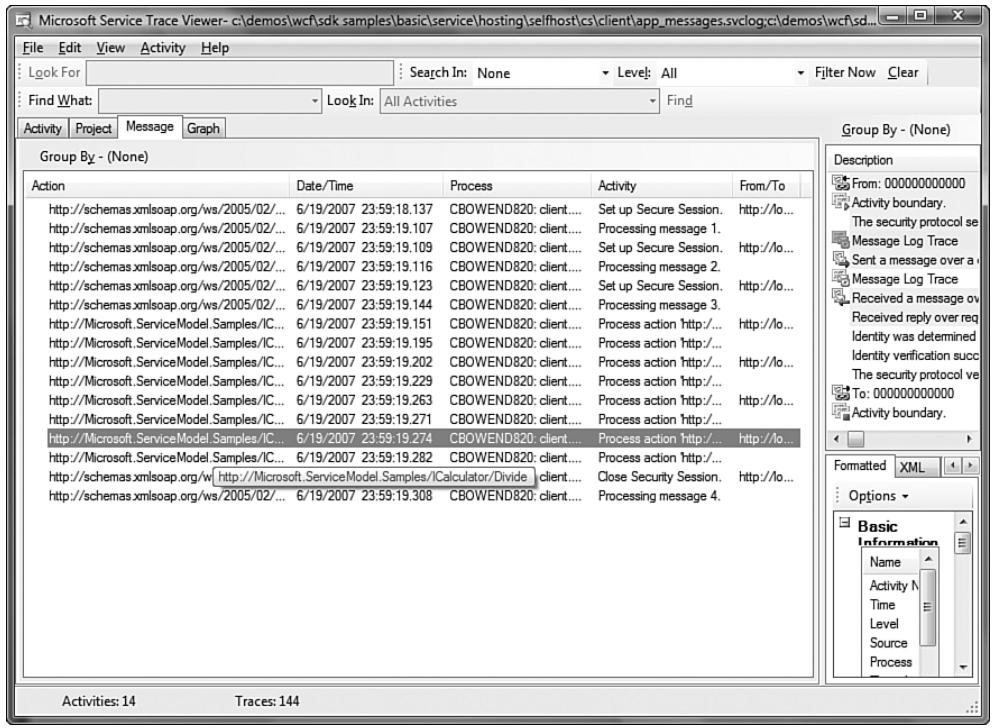

Рис. 9.8. Окно Service Trace Viewer в режима просмотра сообщений

Режим просмотра графа

Это самый сложный, хотя потенциально и наиболее полезный режим программы Service Trace Viewer. Чтобы войти в него, дважды щелкните по деятельности или сообщению в любом из ранее рассмотренных режимов или перейдите на вкладку **Graph**. Вид окна в этом режиме показан на рис. 9.9.

В этом режиме деятельности упорядочены вдоль верхнего края левой панели. На вертикальной «дорожке» показаны все трассировочные записи, входящие в состав деятельностей, причем скоррелированные деятельности соедине-

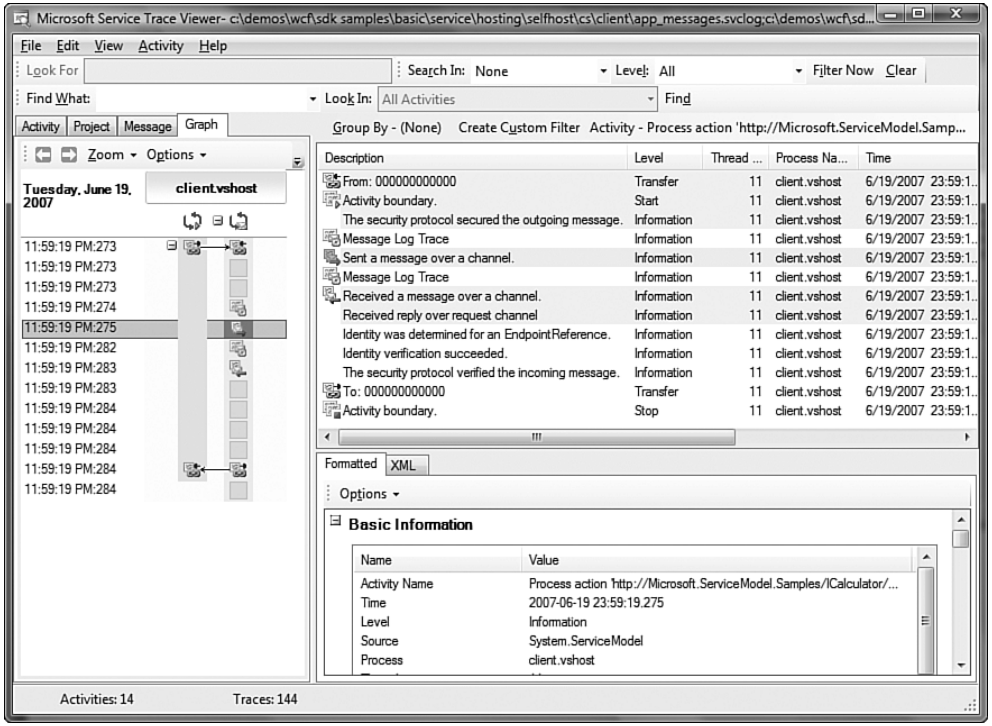

Рис. 9.9. Окно Service Trace Viewer в режима просмотра графа

ны стрелками. Если выбрать любую запись в левой панели, то в правой будут видны все записи, принадлежащие той же деятельности, а выбранная будет подсвечена.

В чем состоит главное удобство этого режима, станет ясно, когда мы включим трассировочные файлы, сгенерированные службой.

Инструмент Live Service Trace Viewer. Программа Service Trace Viewer очень удобна для анализа взаимодействий со службами постфактум. Но есть и другой инструмент – Live Service Trace Viewer, в котором реализован альтернативный подход. В этом приложении использован специальный класс `TraceListener` и написанный с помощью Windows Presentation Foundation (WPF) интерфейс для получения и отображения диагностической информации о событиях в момент их возникновения. Это может оказаться весьма полезно, особенно на этапе разработки, поскольку избавляет от необходимости вручную загружать файлы с протоколами после каждого прогона.

Отметим, что программа Live Service Trace Viewer не поддерживается корпорацией Microsoft, но может служить интересным примером расширения диагностических средств WCF.

Подробную информацию и код можно найти по адресу http://blogs.msdn.com/craigmcmurtry/archive/2006/09/19/762689.aspx.

Анализ протоколов из различных источников

Программа Service Trace Viewer полезна даже для просмотра протоколов работы одного клиента или службы, но по-настоящему мощь этого инструмента и механизма сквозной трассировки раскрывается, когда мы добавляем файлы, сгенерированные различными участниками распределенного приложения.

Чтобы убедиться в этом, выполните команду **File ⇒ Add** (в отличие от **File ⇒ Open**, она объединяет новые протоколы с загруженными ранее) и выберите файлы трассировки и протоколирования сообщений из проекта службы SelfHost. Сгенерированные службой протоколы будут импортированы и скоррелированы с загруженными ранее протоколами клиента, как показано на рис. 9.10.

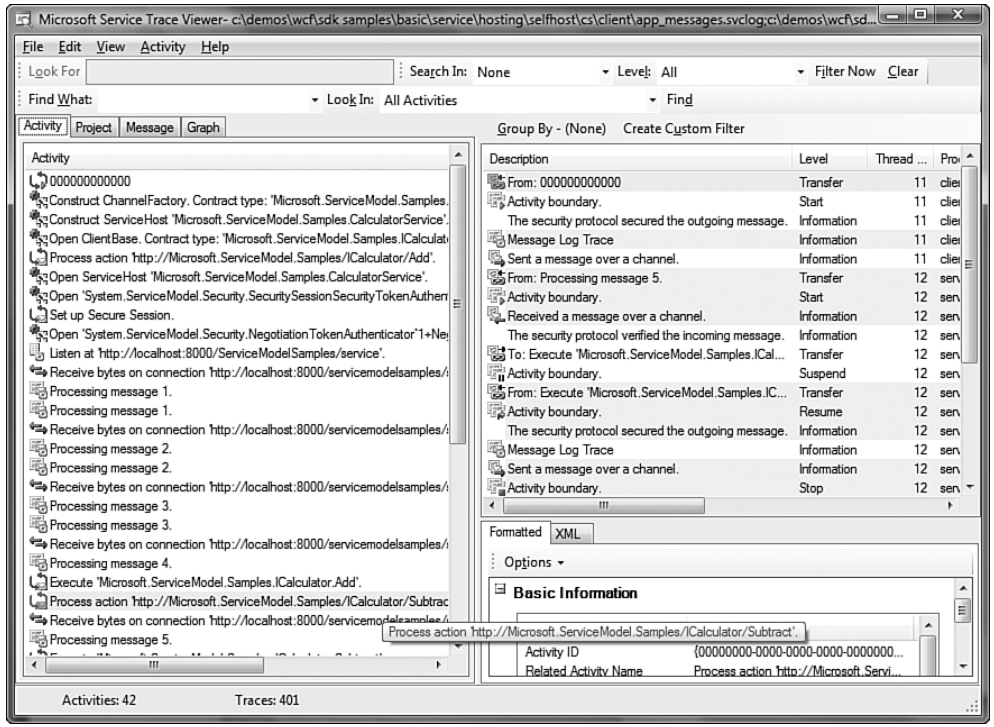

Рис. 9.10. После загрузки протоколов клиента и службы в Service Trace Viewer

Как видите, теперь нам доступно гораздо больше информации. В списке присутствуют деятельности из проектов клиента и службы.

Выберите деятельность **Process Action** для вызова `Subtract` службы и либо дважды щелкните по ней, либо перейдите на вкладку **Graph**. Вашему взору предстанет картина, изображенная на рис. 9.11.

Теперь ясно, как в графическом режиме визуализируются сложные взаимодействия между службами и клиентами. Вдоль верхнего края главной панели расположены деятельности разных участников, в нашем случае службы и одного ее клиента. Задержав курсор мыши над деятельностью, вы получите ее описание. Раскрывая трассировочные записи внутри деятельностей, вы будете видеть визуальные индикаторы корреляции между деятельностями.

Из рис. 9.11 видно, что клиент послал сообщение службе, служба обработала его, вызвав метод `Subtract`, создала ответное сообщение и отправила его клиенту. Визуализация возможна, потому что механизм сквозной трассировки предусматривает корреляцию деятельностей.

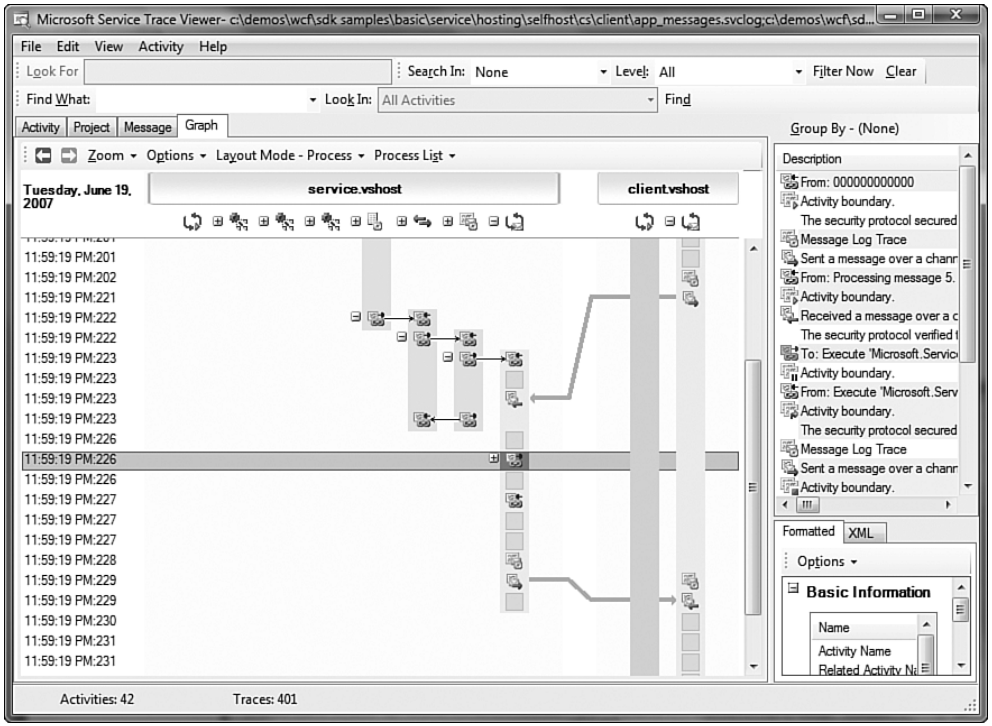

Рис. 9.11. Граф переходов от одной деятельности к другой

Щелчок по значку плюс слева от выделенной трассировочной записи раскрывает ее, показывая следующий уровень детализации. На рис. 9.12 все уровни раскрыты.

В блоке `service.vshost` мы видим новую деятельность – Execute “Microsoft. ServiceModel.Samples.ICalculator.Subtract”. Если бы возникли какие-то предупреждения или исключения, то в режиме просмотра графа они отображались бы в виде желтых треугольничков или красных кружочков. Раскрывая в Service Trace

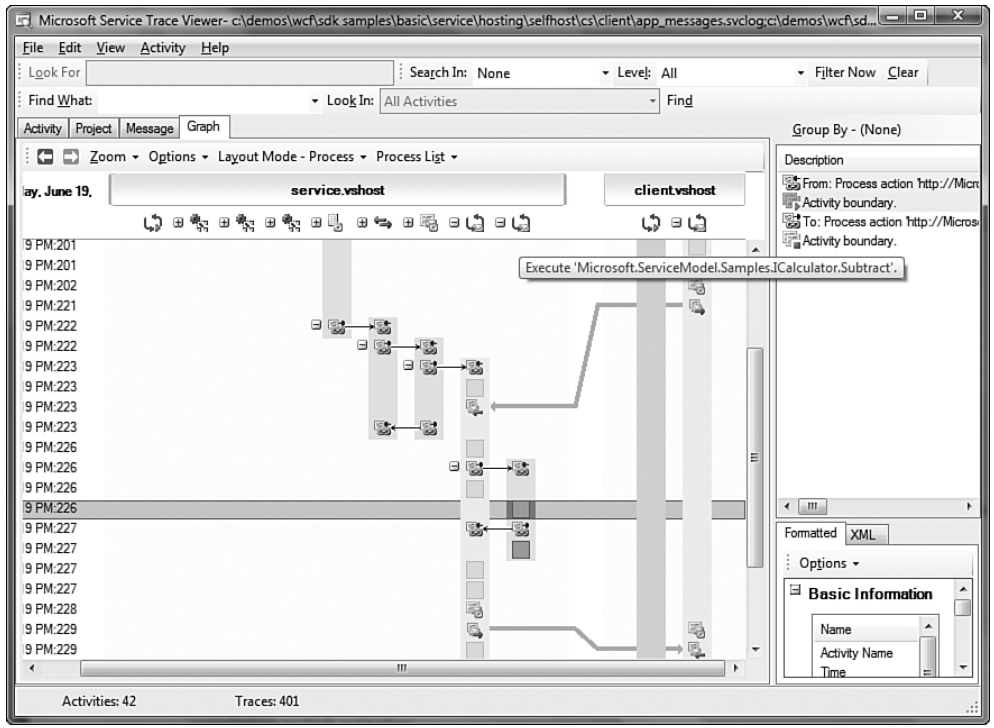

Рис. 9.12. Полностью раскрытые деятельности, связанные с вызовом службы

Viewer узлы для просмотра вложенных деятельностей и наблюдая за взаимодействием между деятельностями и машинами-участниками, разработчик или IT-специалист, расследующий проблему, сможет быстро отыскать источники неожиданного поведения.

Фильтрация результатов

Иногда, особенно при работе с файлами протоколов, сгенерированными промышленно эксплуатируемой системой, найти в них нужную информацию оказывается довольно сложно. Например, вы знаете, что в сеансе определенного пользователя имело место неожиданное поведение (предположим, что не было ни предупреждений, ни исключений, а только некорректные данные). Перспективы расследования этого инцидента пугающие, но Service Trace Viewer предлагает гибкие средства для поиска и фильтрации записей.

Поле **Find What** на панели инструментов позволяет быстро найти все записи, содержащие указанный текст. Наберите, к примеру, `Divide` и нажмите кнопку Find. В списке будут выделены трассировочные записи, содержащие это слово.

Поле **Look For** позволяет оставить в списке только записи, отвечающие указанному критерию. Щелкните по раскрывающемуся списку **Search In** (Искать в)

и выберите критерий (например, **Start Time**). Становится доступно поле **Look For**. Введите начальный момент времени и нажмите кнопку **Filter Now** (Отфильтровать). В списке деятельностей останутся только те, что начались не ранее указанного момента. Можно также выбирать сообщения с заданным уровнем (**Level**) серьезности (например, **Warning**). Кнопка **Clear** восстанавливает исходный нефильтрованный список.

Наиболее полезна возможность создавать и сохранять собственные фильтры. Нажмите кнопку **Create Custom Filter** сверху от списка трассировочных записей; появится диалоговое окно, изображенное на рис. 9.13.

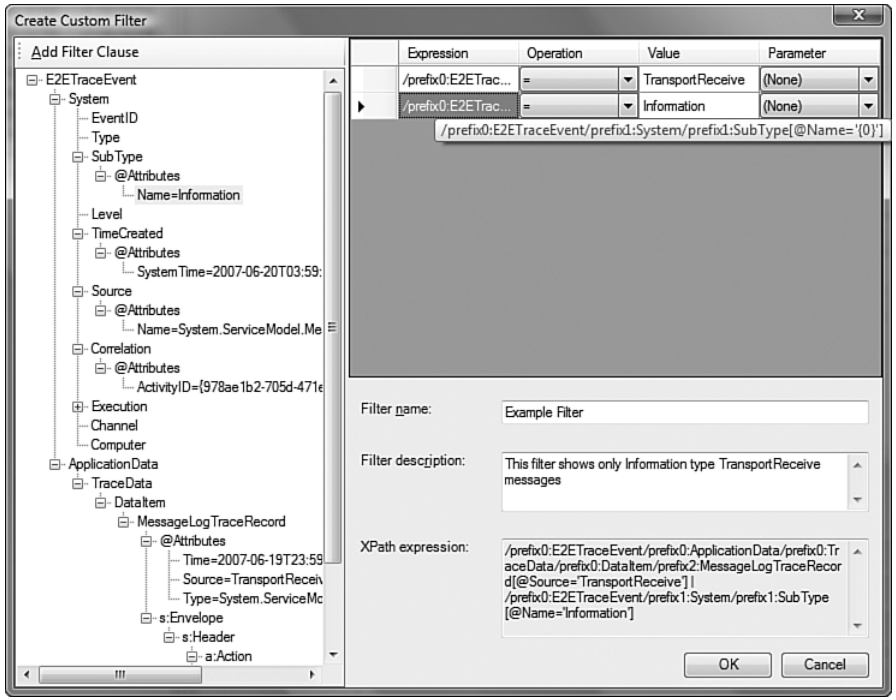

Рис. 9.13. Создание собственного фильтра в программе Service Trace Viewer

Этот редактор позволяет составлять фильтры из одного или нескольких выражений XPath. По умолчанию узлы и атрибуты в левой панели соответствуют тем деятельностям или записям, которые были выбраны в момент нажатия кнопки Create Custom Filter. Укажите атрибуты, которые вы хотите проверять, а затем введите имя и краткое описание фильтра. После нажатия OK вы сможете выбрать свой фильтр из списка Search In на панели инструментов для поиска соответствующих ему записей.

Резюме

В этой главе мы видели, как WCF использует встроенные в .NET Framework средства, чтобы помочь разработчикам и IT-специалистам в диагностике проблем, возникающих в распределенных приложениях.

Идея сквозной трассировки заключается в том, чтобы построить цепочку логически связанных действий, происходивших в различных частях приложения и, быть может, даже в разных системах. Тем самым упрощается воссоздание сценария развития событий по информации, хранящейся в протоколах. Корреляция выполняется за счет передачи уникальных идентификатор внутри одной оконечной точки и между разными оконечными точками WCF-служб, работающих в системе.

Трассировку и протоколирование легко включить и сконфигурировать, поскольку все это основано на знакомых классах из пространства имен `System.Diagnostics`. Трассировка помогает проникнуть в тайны действий, совершаемых внутри наших распределенных приложений. А протоколирование сообщений дает возможность заглянуть в реальные данные, которыми обмениваются клиенты и службы.

Редактор конфигурации служб – полезный инструмент, который входит в состав Windows SDK и позволяет разработчикам и администраторам быстро и уверенно просматривать и изменять конфигурационные параметры WCF, в том числе и настройки диагностики.

Наконец, мы познакомились еще с одной программой из Windows SDK – Service Trace Viewer, которая предназначена для визуализации и инспектирования больших объемов данных, собранных в ходе трассировки и протоколирования сообщений. Она особенно полезна, когда возникают предупреждения или исключения, а система состоит из многих машин (и даже компаний). Разработчики и администраторы с помощью Service Trace Viewer могут быстро изолировать источники неожиданного поведения.

Диагностические средства WCF просты в использовании, но позволяют эффективно сопровождать и расширять сложные распределенные приложения.

Глава 10. Обработка исключений

Неприглядная реальность разработки ПО свидетельствует о том, что даже тщательно написанные системы содержат ошибки и иногда ведут себя неожиданным образом. Хороший разработчик должен выдерживать баланс между предотвращением ошибок и их обработкой по мере возникновения. Распределенные системы на базе служб – не исключение. На самом деле, они даже усугубляют проблему, поскольку привносят такие факторы, как доступность серверов, условия в сети и совместимость версий служб.

Исключения – критически важный компонент надежной системы, поскольку они служат индикаторами различных ситуаций. Например, клиенту может быть предоставлена неправильная или неполная информация о службе, служба может столкнуться с проблемами при попытке завершить операцию, а сообщение может быть отформатировано для уже неподдерживаемой версии.

В этой главе мы поговорим о месте исключений в WCF и рассмотрим средства, которые WCF предоставляет для передачи и обработки исключений. Мы опишем различия между исключениями (exception) и отказами (fault), покажем, как создать отказ и послать его клиенту, и расскажем об обработке исключений на стороне службы и клиента. Наконец, будут продемонстрированы способы централизованной обработки исключений владельцем службы, перехвата неожиданных исключений и выполнения дополнительных действий над исключениями и отказами, например, протоколирования.

Введение в обработку исключений в WCF

Прежде чем переходить к подробному рассмотрению обработки исключений в службе WCF и ассоциированных с ней клиентских приложениях, взглянем, что происходит, когда исключения возбуждаются в системе с настройками, принимаемыми по умолчанию. Ведя разработку на платформе WCF, нужно отчетливо представлять себе, что случится, если не принимать исключения во внимание.

Обычно WCF-служба обертывает вызовы к библиотекам, содержащим бизнес-логику, а эти библиотеки, как и любой управляемый код, могут возбуждать стандартные исключения .NET. Исключение распространяется вверх по стеку вызовов, пока не будет кем-то перехвачено или не достигнет корневого контекста. В последнем случае оно обычно оказывается фатальным для вызывающего приложения, процесса или потока (в зависимости от вида приложения).

Хотя самой WCF необработанные исключения не страшны, все же WCF предполагает, что исключение – это признак серьезной проблемы, которая может помешать службе продолжать взаимодействие с клиентом. В таких случаях WCF

посылает отказ в канал службы, это означает, что все существующие сеансы (созданные, например, для обеспечения безопасности, надежной доставки или обобществления состояния) должны быть уничтожены. Если сеанс является частью обращения к службе, то клиентский канал становится бесполезным, и для продолжения взаимодействия клиент должен заново создать объект прокси-класса.

Передача исключений по протоколу SOAP

Исключения, возникающие в результате изъянов в реализации логики службы или внутри самого владельца, – это объекты классов, производных от типа `Exception`, встроенного в CLR. Поскольку службы должны поддерживать коммуникацию с любым клиентом вне зависимости от платформы, на которой он реализован, эти специфичные для .NET детали необходимо преобразовать в стандартизованный формат для обеспечения интероперабельности.

Интероперабельность достигается за счет сериализации зависящих от платформы деталей исключений согласно общей схеме, описанной в спецификации протокола Simple Object Access Protocol (SOAP). В ней упоминается элемент отказа (fault), который может присутствовать в теле сообщения SOAP.

В этой главе мы опишем несколько способов передачи исключений от службы клиенту в виде отказов. Детальное знакомство со схемой отказов в SOAP обычно не требуется, поскольку инфраструктура WCF абстрагирует эти подробности, предлагая различные варианты предоставления дополнительной информации, которая затем преобразуется в нужные элементы и атрибуты отказа SOAP.

Как минимум, в отказе SOAP должно быть задано два значения. Первое – это *причина*, то есть описание ошибки. Второе – *код* ошибки, который может быть либо специализированным индикатором, либо одним их предопределенных значений, перечисленных в спецификации SOAP. Позже мы еще вернемся к этому вопросу (когда будем обсуждать тип `FaultException`).

Дополнительную информацию о том, как трактуется управление отказами в спецификации SOAP, см. на сайте W3C по адресу www.w3.org/TR/2007/REC-soap12-part0-20070427/#L11549.

Пример необработанного исключения

Чтобы увидеть, как ведет себя WCF, когда необработанное исключение доходит до владельца службы, создадим простейшую службу и клиент для нее. Для демонстрации отказа канала нужно, чтобы служба включала сеансы. Для этого воспользуйтесь, например, привязкой `wsHttpBinding`, которая организует сеанс ради обеспечения безопасности.

В реализации службы создайте такую операцию, как показано в листинге 10.1.

Листинг 10.1. Пример контракта и его реализации

```
using System;
using System.ServiceModel;

namespace ServiceLibrary
```

```
{
   [ServiceContract()]
   public interface IService
   {
      [OperationContract]
      double Divide(double numerator, double denominator);
   }

   public class Service1 : IService
   {
      public double Divide(double numerator, double denominator)
      {
         if (denominator == 0)
               throw new ArgumentOutOfRangeException("denominator",
                        "Значение должно быть числом, отличным от нуля");

         return numerator/denominator;
      }
   }
}
```

Создайте клиент в виде Windows-приложения и с помощью операции Add Service Reference сгенерируйте прокси-класс для этой службы. Нарисуйте простую форму с двумя текстовыми полями TextBox и двумя кнопками Button. Пусть первая кнопка вызывает Web-службу `Divide`, передавая ей значения, введенные в текстовые поля, в качестве аргументов. Вторая кнопка пусть создает новый локальный экземпляр прокси-класса. Возможный код клиента приведен в листинге 10.2.

Листинг 10.2. Клиентское Windows-приложение

```
using System;
using System.ServiceModel;
using System.Windows.Forms;

namespace WindowsClient
{
   public partial class Form1 : Form
   {
      Service1.ServiceClient _serviceProxy = new
               WindowsClient.Service1.ServiceClient();

      public Form1() {InitializeComponent();}

      private void button1_Click(object sender, EventArgs e)
      {
         try
         {
            MessageBox.Show(_serviceProxy.Divide(
                        double.Parse(txtInputA.Text),
                        double.Parse(txtInputB.Text)).ToString());
         }
         catch (FaultException exp)
         {
            MessageBox.Show(exp.Code.Name + ": " +
```



```
                        exp.Message.ToString(),
                        exp.GetType().ToString());
         }
      }

      private void cmdNewProxy_Click(object sender, EventArgs e)
      {
         _serviceProxy = new WindowsClient.Service1.ServiceClient();
      }
   }
}
```

Запустите это приложение и введите параметры функции (например, 10 и 5). В предположении, что пример написан правильно, будет выведен результат деления (2). Теперь задайте знаменатель равным 0 и повторите вызов. Должно появиться окно, показанное на рис. 10.1.

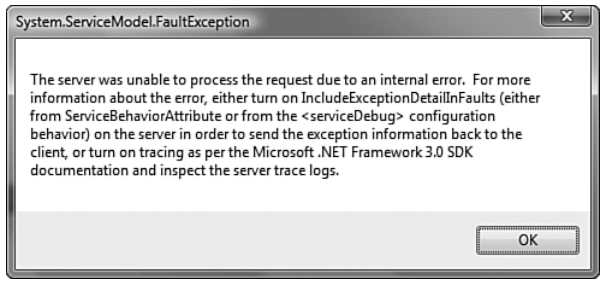

Рис. 10.1. Исключение `FaultException`, возвращенное службой при задании нулевого знаменателя

Снова введите ненулевой знаменатель и повторите вызов службы. Хотя этот вызов мог бы быть обработан нормально, на самом деле происходит ошибка – возвращается исключение `CommunicationObjectFaultedException` и появляется окно сообщения, изображенное на рис. 10.2.

Поскольку WCF получила необработанное исключение на уровне владельца службы, она сочла его признаком фатальной ошибки и перевела канал с сервером

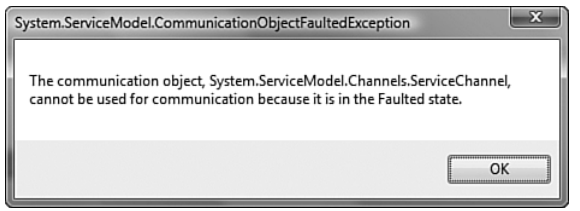

Рис. 10.2. Исключение `CommunicationObjectFaultedException`, возникающее из-за отказа в канале

в состояние отказа. В случае привязки `wsHttpBinding` это означает, что установленный для обеспечения безопасности сеанс уже недействителен, поэтому коммуникацию нужно инициализировать заново, создав новый экземпляр прокси-класса.

Односторонние операции и отказы. Односторонние операции по определению не получают от службы ответных сообщений независимо от того, закончился вызов успешно или неудачно. Поскольку ответа нет, то у клиента нет никакой информации о том, что произошел отказ.

Кроме того, если отказ был вызван необработанным исключением, то канал с сервером перейдет в состояние отказа, но клиент об этом не узнает. Если для взаимодействия необходим сеанс, то все последующие вызовы будут завершаться неудачно (с исключением `CommunicationObjectFaultException`), пока не будет создан новый экземпляр прокси-класса. Не забывайте об этой особенности односторонних операций, проектируя логику клиента и службы.

Обнаружение и восстановление отказавшего канала

Обнаруживать отказ канала может и должен клиент, который обязан проверять состояние канала после каждого отказа и выяснять, вызван ли он ошибкой в самом канале. Для этого клиент опрашивает свойство `State` канала в блоке обработки исключения, как показано в листинге 10.3.

Листинг 10.3. Проверка состояния канала

```
private void button1_Click(object sender, EventArgs e)
{
   try
   {
      MessageBox.Show(_serviceProxy.Divide(double.Parse(txtInputA.Text),
                      double.Parse(txtInputB.Text)).ToString());
   }
   catch (FaultException exp)
   {
      MessageBox.Show(exp.Code.Name + ": " + exp.Message.ToString(),
exp.GetType().ToString());

      if (_serviceProxy.State == CommunicationState.Faulted)
      {
         MessageBox.Show("Отказ коммуникационного канала. Пробую
➥восстановиться.");
         cmdNewProxy_Click(null, null);
      }
   }
   catch (CommunicationException exp)
   {
      MessageBox.Show("Ошибка связи: " +
                      exp.Message.ToString(), exp.GetType().ToString());
   }
   catch (Exception exp)
   {
      MessageBox.Show("Ошибка общего характера: " +
                      exp.Message.ToString(),exp.GetType().ToString());
   }
}
```

Если обнаружен отказ канала, следует запротоколировать состояние и причину, попытаться создать новый экземпляр прокси-класса и продолжить. Если это невозможно, например потому, что был установлен сеанс, который вы не можете пересоздать вручную, то следует уведомить пользователя и прекратить дальнейшую работу с данным прокси.

Передача информации об исключении

В предыдущем примере мы вызывали службу, которая возбудила исключение, дошедшее до уровня владельца, и клиент получил минимальную информацию, показанную в листинге 10.1. По умолчанию WCF передает только это сообщение, а не детали исключения, чтобы не раскрывать клиенту важные сведения о реализации или инфраструктуре службы.

Если необходимо передать клиенту информацию об исключении, то можно воспользоваться свойством `IncludeExceptionDetailInFaults` поведения `ServiceDebugBehavior`. Чтобы включить это поведение, модифицируйте файл `app.config`, как показано в листинге 10.4.

Листинг 10.4. Включение параметра IncludeExceptionDetailsInFaults в конфигурационном файле

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="ServiceLibrary.Service1"
behaviorConfiguration="MetadataAndExceptionDetail">
            <endpoint contract="ServiceLibrary.IService"
binding="wsHttpBinding"/>
            <endpoint contract="IMetadataExchange" binding="mexHttpBinding"
address="mex" />
         </service>
      </services>
      <behaviors>
         <serviceBehaviors>
            <behavior name="MetadataAndExceptionDetail" >
               <serviceMetadata httpGetEnabled="true" />
               <serviceDebug includeExceptionDetailInFaults="false" />
            </behavior>
         </serviceBehaviors>
      </behaviors>
   </system.serviceModel>
</configuration>
```

Если теперь снова запустить службу и задать нулевой знаменатель, то появится сообщение, показанное на рис. 10.3.

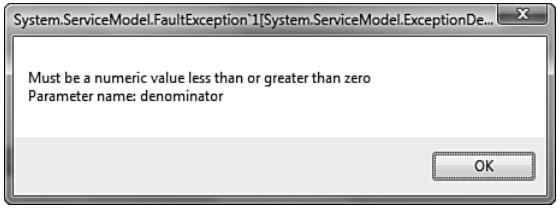

Рис. 10.3. Подробная информация об исключении после изменения поведения ServiceDebugBehavior

Отметим, что задать такое поведение можно также с помощью атрибута `ServiceBehaviorAttribute` в определении службы. Например, можно было бы разрешить передачу отладочной информации, следующим образом модифицировав код Service.cs.

Листинг 10.5. Использование атрибута ServiceBehaviorAttribute с параметром IncludeExceptionDetailsInFaults

```
using System;
using System.ServiceModel;

namespace ServiceLibrary
{
   [ServiceBehavior(IncludeExceptionDetailInFaults=true)]
   public class Service1 : IService
   {
      public double Divide(double numerator, double denominator)
      {...}
   }
}
```

Отметим, что такое использование атрибута для разрешения передачи информации об исключении отменяет все прочие настройки в конфигурационном файле, которые, возможно, отключали этот режим. Поэтому в приложениях, предназначенных для промышленной эксплуатации, мы рекомендуем пользоваться конфигурацией, оставляя этот атрибут только для этапа разработки.

Какой бы способ вы ни выбрали, не забывайте отключать передачу деталей исключения, если нет жесткой необходимости. Это предотвратит раскрытие сведений о реализации службы.

Управление исключениями в службе с помощью класса FaultException

В нашем первом примере мы продемонстрировали, что происходит, когда необработанному исключению позволяют добраться до уровня владельца службы. На рис. 10.1 видно, что клиент получает исключение типа `FaultException`. Этот класс, производный от `Exception`, используется в WCF для интеграции со спецификацией отказов в SOAP.

По умолчанию исключения, достигшие владельца службы и не являющиеся производными от `FaultException`, считаются признаком потенциально фатальной ошибки. В этом случае исключение заменяется на `FaultException`, а детали исходного исключения опускаются, если только не задан параметр `IncludeExceptionDetailInFaults`. Затем `FaultException` сериализуется в виде отказа SOAP для передачи клиенту (если только вызванная операция не односторонняя). Повторим, что если вызов был частью сеанса, то этот сеанс уничтожается и должен быть создан заново.

Фатальную ошибку, возникающую из-за необработанных исключений, можно предотвратить, перехватив исключение еще до того, как оно достигнет владельца хоста и возбудив исключение `FaultException` вручную. В классе `FaultException` определено несколько конструкторов и свойств, с помощью которых можно задать различные обязательные и дополнительные значения, относящиеся к представлению отказа по спецификации SOAP.

Рекомендуется перехватывать в службе все исключения и включать представляющие интерес детали в новый экземпляр `FaultException`. В листинге 10.6 приведен соответствующий пример.

Листинг 10.6. Перехват исходного исключения и возбуждение вместо него FaultException

```
public bool ApproveInvoice(int invoiceId)
{
   try
   {
      Invoice invoice = InvoiceSystem.GetInvoice(invoiceId);
      return invoice.Approve();
   }
   catch (ArgumentException exp)
   {
      // Протоколировать детали и контекст возникновения ArgumentException

      // Возбудить FaultException для передачи клиенту
      throw new FaultException(invoiceId + " некорректная накладная.");
   }
   catch (Exception exp)
   {
      // Протоколировать детали и контекст возникновения Exception

      // Возбудить FaultException для передачи клиенту
      throw new FaultException("Ошибка при обработке накладной: " + exp.Message);
   }
}
```

Использование FaultCode и FaultReason для расширения FaultException

Код в листинге 10.5 неплох в качестве отправной точки, но он не включает дополнительную информацию, которая могла бы оказаться полезной клиенту.

Конструктору объекта FaultException можно передать также аргументы FaultReason и FaultCode, которые используются для формирования соответственно элементов reason и code в отказе SOAP.

FaultCode может принимать три значения. Код Sender означает ошибку в сообщении, которое отправил клиент. Если при создании FaultException не было указано иное, то это значение принимается по умолчанию. Код Receiver означает, что обработку не удалось завершить из-за проблем, с которыми столкнулась служба. И, наконец, может быть указан нестандартный код.

Класс FaultReason полезен, когда нужно создать несколько сообщений о причине на разных языках. Спецификация SOAP 1.2 допускает наличие нескольких элементов reason с различными localeID например, en-US для американского диалекта английского языка). Конструктор FaultReason принимает набор объектов типа FaultReasonText, каждый из которых содержит текстовую строку на одном языке вместе с идентификатором соответствующей локали.

В листинге 10.7 приведен пересмотренный код из листинга 10.6, в котором классы FaultCode и FaultReason используются для включения дополнительной информации об отказе в сериализованный документ, передаваемый клиенту.

Листинг 10.7. Расширение FaultException с помощью FaultCode и FaultReason

```
public bool ApproveInvoice(int invoiceId)
{
   try
   {
      Invoice invoice = InvoiceSystem.GetInvoice(InvoiceId);
      return invoice.Approve();
   }
   catch (ArgumentException exp)
   {
      throw new FaultException(invoiceId + " некорректная накладная.");
   }
   catch (InvoiceNotFoundException exp)
   {
      throw new FaultException(invoiceId + " накладная не найдена.",
FaultCode.CreateReceiverFaultCode(new FaultCode("GetInvoice")));
   }
   catch (Exception exp)
   {
      List<FaultReasonText> frts = new List<FaultReasonText>();
      frts.Add(new FaultReasonText("Ошибка при обработке накладной"));
      frts.Add(new FaultReasonText("<перевод на французский>",
               new CultureInfo("fr-FR")));
      frts.Add(new FaultReasonText("<перевод на чешский>",
               new CultureInfo("cs-CZ")));
      throw new FaultException(new FaultReason(frts),
               FaultCode.CreateReceiverFaultCode(
                 new FaultCode("ApproveInvoice")));
   }
}
```


Поскольку исключение ArgumentException свидетельствует о том, что службе было передано недопустимое значение, то мы не задаем FaultCode, оставляя подразумеваемое по умолчанию значение Sender. В случае исключения InvoiceNotFoundException мы хотим указать, что ошибка возникла в работе самой службы, поэтому вызываем статический метод CreateReceiverFaultCode() для создания кода Receiver.

В блоке, где перехватываются все остальные исключения, мы демонстрируем технику работы с другим конструктором класса FaultReason, который позволяет передать клиенту сообщения об ошибке на разных языках. Сначала создается обобщенный список List объектов типа FaultReasonText, куда помещаются тексты сообщений и соответствующие коды культуры. Затем из этого списка конструируется объект FaultReason и передается конструктору FaultException.

Чтобы клиент мог получить доступ к переведенным сообщениям, в классе FaultException определено свойство Reason, раскрывающее метод GetMatchingTranslation(). Чтобы автоматически получить перевод, соответствующий культуре текущего потока, вызовите этот метод без аргументов: <exception>.Reason.GetMatchingTranslation(). Можно также передать аргумент типа CultureInfo, если вас интересует перевод на конкретный язык.

Ограничения класса FaultException

Базовый класс FaultException дает простой способ предотвратить распространение необработанного исключения до владельца службы, что может привести к уничтожению сеанса и недействительности клиентского прокси-объекта. Но у этого класса есть серьезный недостаток – отсутствие механизма идентификации ошибок. Если ваша служба возвращает только объекты FaultException, то разработчикам клиентских приложений будет трудно написать надежный алгоритм обработки исключений. Рассмотрим пример в листинге 10.8.

Листинг 10.8. Обработка нетипизированных объектов FaultException

```
public double GetPrice(int itemId)
{
   try
   {
      // Вызвать операцию службы
   }
   catch (FaultException exp)
   {
      // Обработать исключения WCF
      // Проанализировать Reason, Code, Message и т.д., чтобы выбрать
      // подходящее действие
   }
   catch (Exception exp)
   {
      // Обработать все остальные исключения, в том числе, возникшие
      // на стороне клиента
   }
}
```

Проблема в том, что нет подклассов FaultException, которые можно было бы обработать в отдельных блоках catch. Ниже предлагается более разумный подход, который позволит разработчику клиентского приложения извлечь дополнительную информацию из свойств объекта исключения и на ее основе выбрать подходящий способ восстановления.

Создание и обработка строго типизированных отказов

Как мы только что отметили, класс FaultException не позволяет построить логику обработки исключений на основе его типа в блоке try/catch/finally на стороне клиента. Чтобы разрешить эту проблему, попробуем воспользоваться обобщенным классом FaultException< >.

Класс FaultException< > параметризуется типом, который определяет структуру сериализуемых данных об исключении. Это может быть любой тип, допускающий сериализацию при передаче, но чтобы клиент мог им воспользоваться, ему необходимо знать определение типа.

Например, мы могли бы в листинге 10.6 возбудить исключение типа FaultException<ArgumentException>. Если клиент – это приложение .NET, то такой подход сработает, так как обе стороны знают о деталях определения типа ArgumentException. А что, если к службе обратится клиент, написанный с помощью иной технологии, например на Java? Язык Java ничего не знает о типе ArgumentException, поэтому написанный на Java прокси-класс не сможет получить типизированный доступ к деталям отказа, возвращенного нашей службой.

Для обеспечения интероперабельности в WSDL-документ, описывающий службы, необходимо включить структуру типа, которым параметризован класс FaultException<>. Для этого предназначен контракт об отказах.

Объявление отказов с помощью класса FaultContract

Напомним, что контракт о данных определяет структуры данных, представимые с помощью WSDL-описания службы. За счет этого клиент имеет полную информацию о типах данных, которые он должен посылать службе или принимать от нее. Ту же идею контракта можно применить для описания структуры данных, содержащих информацию об отказе.

Любую операцию службы можно снабдить одним или несколькими атрибутами FaultContract. Они говорят WCF о том, что в WSDL-описание следует включить информацию об отказе, который потенциально может произойти в данной операции. Зная это, инструменты генерации прокси-классов способны создать строго типизированные представления классов, несущих информацию об отказах. А, имея определения таких прокси-классов, разработчик сумеет написать надежное клиентское приложение для работы со службой. Отметим, что коль скоро детали представлены в стандартном WSDL-формате, можно сгенерировать прокси для любой платформы, а не только для .NET.

Давайте определим структуру данных для передачи информации об ошибке вызывающей стороне. Создайте класс TrackedFault, как показано в листинге 10.9.

Листинг 10.9. Создание DataContract для использования совместно с FaultContract

```
using System;
using System.Runtime.Serialization;

namespace ServiceLibrary
{
   [DataContract]
   public class TrackedFault
   {
      Guid _trackingId;
      string _details;
      DateTime _dateTime;

      [DataMember]
      public Guid TrackingId
      {
         get { return _trackingId; }
         set { _trackingId = value; }
      }

      [DataMember]
      public string Details
      {
         get { return _details; }
         set { _details = value; }
      }

      [DataMember]
      public DateTime DateAndTime
      {
         get { return _dateTime; }
         set { _dateTime = value; }
      }

      public TrackedFault(Guid id, string details, DateTime dateTime)
      {
         _trackingId = id;
         _details = details;
         _dateTime = dateTime;
      }
   }
}
```

Определение контракта об отказе

Определив один или несколько контрактов о данных, позволяющих передать детали исключения клиентам, снабдите операцию атрибутом FaultContract, указав в нем имена соответствующих контрактов о данных.

Например, операция `ApproveInvoice` в листинге 10.10 может возбуждать исключение `FaultException` параметризованное типом контракта `TrackedFault`.

Листинг 10.10. Уточнение определения операции с помощью атрибута FaultContract

```
[OperationContract]
[FaultContract(TrackedFault)]
public bool ApproveInvoice(int invoiceId)
{
   ...
}
```

Односторонние операции и контракты об отказах. Мы уже говорили, что односторонние операции не возвращают своим клиентам никаких сообщений, поэтому уведомить клиента об отказе невозможно. Раз так, то на этапе загрузки службы возбуждается исключение `InvalidOperationException`, если есть хотя бы одна односторонняя операция с атрибутом `FaultContract`.

Возбуждение исключения FaultException<>, параметризованного контрактом об отказе

Итак, мы сообщили WCF, что операция `ApproveInvoice` может возбуждать исключение, сериализуемое в соответствии с контрактом о данных `TrackedFault`. Осталось лишь заполнить этот контракт и возбудить исключение. В листинге 10.11, являющемся уточнением листинга 10.7, показано, как это делается.

Листинг 10.11. Возбуждение FaultException<>, параметризованного FaultContract

```
[OperationContract]
[FaultContract(typeof(TrackedFault))]
public bool ApproveInvoice(int invoiceId)
{
   try
   {
      Invoice invoice = InvoiceSystem.GetInvoice(invoiceId);
      return invoice.Approve();
   }
   catch (ArgumentException exp)
   {
      throw new FaultException(invoiceId + "is not valid.");
   }
   catch (InvoiceNotFoundException exp)
   {
      TrackedFault tf = new TrackedFault(
                           Guid.NewGuid(),
                           invoiceId + " накладная не найдена.",
                           DateTime.Now);

      throw new FaultException<TrackedFault>(
                           tf,
                           new FaultReason("InvoiceNotFoundException"),
                           FaultCode.CreateReceiverFaultCode(new
                              FaultCode("GetInvoice")));
   }
   catch (Exception exp)
   {
      List<FaultReasonText> frts = new List<FaultReasonText>();
      frts.Add(new FaultReasonText("Error processing invoice"));
      frts.Add(new FaultReasonText("<French translation>",
                new CultureInfo("fr-FR")));
      frts.Add(new FaultReasonText("<Czech translation>",
                new CultureInfo("cs-CZ")));
      throw new FaultException(new FaultReason(frts),
                FaultCode.CreateReceiverFaultCode(
                   new FaultCode("ApproveInvoice")));
   }
}
```


Существует ряд стратегий описания контрактов об отказах служб. Можно создать разделяемую библиотеку контрактов, используемую во всех службах, разработанных компанией. Можно задавать контракты для отдельной службы или приложения, содержащие специфичные для нее детали. Или создать контракт для особого класса ошибок, характерных только для ваших служб.

Мы рекомендуем в каждом новом проекте WCF-службы всюду применять контракты об отказах. Снабжая операции атрибутом `FaultContract`, описывающим ожидаемые исключения, вы даете возможность клиентскому приложению обрабатывать исключения на основе их типов. Какую бы стратегию вы ни выбрали, применяйте ее последовательно, это упростит сопровождение клиентов и сделает их более удобными для применения.

Реализация обработчиков отказов на стороне клиента

Если вы снабдили операции службы атрибутами `FaultContract`, определив контракты данных для каждого отказа, то в WSDL-описание службы будет включена вся информация из этих контрактов. Это позволит вам или другим разработчикам писать на стороне клиента обработчики, имеющие полный доступ к деталям и типам контрактов.

Продолжая начатый ранее пример, мы в листинге 10.12 демонстрируем, как можно структурировать обработку исключения `FaultException`, параметризованного контрактом `TrackedFault`.

Листинг 10.12. Обработка типизированного исключения FaultException

```
public double GetPrice(int itemId)
{
   try
   {
      // Вызвать операцию службы
```

```
   }
   catch (FaultException<TrackedFault> tfexp)
   {
      // Полный, строго типизированный доступ к TrackedFault

      // Свойства TrackedFault доступны через свойство Detail
      // типа FaultException<>.

      MessageBox.Show("Обнаружена и запротоколирована ошибка.
➥При обращении в службу поддержки укажите Id " +
➥tfexp.Details.TrackingId.");
   }
   catch (FaultException fexp)
   {
      // Обработать остальные исключения WCF.
      // Проанализировать Reason, Code, Message и т.д., чтобы выбрать
➥подходящее действие.
   }
   catch (Exception exp)
   {
      // Обработать все остальные исключения, в том числе, возникшие
➥на стороне клиента
   }
}
```

Раскрытие информации о типе `TrackingFault` в WSDL-описании службы позволяет инструменту генерации прокси-класса создать строго типизированные свойства для каждого члена `TrackingFault`. Обработчик исключений на стороне клиента может получить доступ к информации из вложенного контракта о данных с помощью свойства `Details` объекта `FaultException<>`. В листинге 10.11 в сообщении пользователю указывается идентификатор ошибки, полученный из свойства `Details.TrackingId` типизированного экземпляра исключения.

Напомним, что при обработке любых исключений в .NET важен порядок блоков `catch`. Поскольку `FaultException<TrackedFault>` – это самый специфичный тип, его обработчик должен быть первым. Так как тип `FaultException` более специфичен, чем `Exception` (которому он наследует), то его обработчик должен быть следующим, а последним идет обработчик исключения `Exception`.

Код в этом примере не полон, но можно было бы включить в него различные дополнительные действия, например: протоколирование подробной информации об исключении, уведомление пользователя и аварийный выход или попытку повторить вызов службы.

Прикладной блок обработки ошибок

Группа обобщения практики применения (Patterns and Practices) в корпорации Microsoft занимается созданием руководств и инструментов, помогающих воплотить имеющиеся технологии в реальные приложения с помощью зарекомендовавших себя приемов. Enterprise Library представляет собой библиотеку бесплатно распространяемых классов – *прикладных блоков*, которыми можно воспользоваться для применения передового опыта в собственных программах.

В последнюю версию Enterprise Library (на настоящий момент 3.1) включены новые средства, интегрированные с некоторыми функциями .NET Framework 3.0 и 3.5. Один из прикладных блоков называется *Exception Handling Application Block* (прикладной блок обработки исключений), он позволяет определить политику обработки исключений в приложении. Например, можно создать политику, согласно которой все исключения, относящиеся к данным, протоколируются и замещаются исключением типа `Exception`.

Еще о библиотеке Enterprise Library. Полное описание библиотеки Enterprise Library и ее связей с каркасом .NET 3.x выходит за рамки настоящей книги, но дополнительную информацию можно найти на сайте http://msdn.com/practices.

Экранирование исключений

В последней версии Enterprise Library в прикладной блок обработки исключений добавлена новая возможность: *экранирование исключений* (exception shielding). Частью ее является новый атрибут `ExceptionShielding`, который можно употреблять, чтобы применить политику исключений при обращении к службе. В сочетании с классом `FaultContractExceptionHandler` это позволяет преобразовать исключения .NET, возбуждаемые операциями службы, в экземпляры строго типизированных классов `FaultException<>`, параметризованных контрактом о данных.

Вместе с другими обработчиками, осуществляющими обертывание, замещение и протоколирование, класс `FaultContractExceptionHandler` в зависимости от конфигурации может выполнять отображение между полями объекта-исключения и контракта о данных, заданного в атрибуте `FaultContract`.

Экранирование исключений можно рассматривать как способ перехватить и преобразовать все или только некоторые типы исключений, которые не были обработаны ранее с помощью методов, описанных выше в этой главе.

Дополнительную информацию и примеры применения Enterprise Library к экранированию исключений WCF можно найти в MSDN по адресу http://msdn2.microsoft.com/en-us/library/aa480591.aspx.

Резюме

В этой главе мы рассмотрели, как в WCF обрабатываются исключения и как можно повлиять на способ обработки. Самое важное – понимать, что происходит, когда вы не предпринимаете никаких действий. По умолчанию необработанное исключение может привести к разрыву сеанса и сделать клиентские прокси недействительными.

Чтобы организовать эффективную стратегию обработки исключений и избежать потенциальных проблем, необходимо использовать класс `FaultException`. Он помогает абстрагировать детали схемы отказов SOAP, которая применяется, чтобы сериализовать информацию об исключении и передать ее клиенту. Вы ви-

дели, что класс `FaultException` можно расширить, включив в него подробные сведения о кодах и причинах ошибки и даже переведенные на разные языки сообщения, что необходимо для локализации клиентских приложений.

От рассмотрения необработанных исключений и базового класса `FaultException` мы перешли к понятию строго типизированных исключений, основанных на использовании обобщенного класса `FaultException<>`. Применяя контракты о данных для передачи подробных сведений об исключении, клиентское приложение может организовать обработку на основе типа и выбрать подходящее действие, не прибегая к анализу отдельных экземпляров `FaultException` во время выполнения. Атрибут `FaultContract` позволяет инструментам генерации прокси-классов проанализировать WSDL-описание службы и создать строго типизированные представления исходных контрактов о данных, не зависящие от технологии разработки клиента.

Наконец, мы познакомились с идеями, заложенными в прикладной блок обработки исключений в библиотеке Enterprise Library. Они позволяют без труда создавать политики обработки исключений, пересекающих границы службы.

Невозможно переоценить важность внедрения стратегии обработки исключений, которая четко определена и одинаково понимается всеми членами команды, группы или целой компании. Создавая WCF-службы и клиентские приложения для них, продумывайте механизмы обеспечения предсказуемого поведения службы и одновременно надежной работы клиентов, которых к тому же должно быть удобно сопровождать.

Глава 11. Потоки работ

Дойдя до этого места, вы уже понимаете, что смысл WCF – это службы: их определение, создание и обеспечение безопасности. Службы имеют четко определенные границы, которые формально описаны в контрактах, но их внутреннее устройство абсолютно непрозрачно. WCF мало что говорит о реализации службы; она лишь предоставляет интерфейсы, с помощью которых службы могут надежно и безопасно взаимодействовать с клиентами.

Windows Workflow Foundation (WF) – технология, дополняющая WCF. Ее назначение – определить и выполнить операции (activity)[1] многошагового процесса (иначе говоря, потока работ). WF позволяет моделировать как последовательные, так и событийно-управляемые потоки работ. Исполняющая среда WF выполняет операции с помощью организации условий, циклов, разветвлений и последующих слияний. Поток работ может быть совсем коротким или выполняться длительное время. WF не делает почти никаких предположений об интерфейсе с потоком работ, поэтому может применяться в самых разнообразных приложениях.

Новое в .NET 3.5. WCF и WF поставлялись в составе .NET 3.0. Описанная в этой главе интеграция между ними появилась только в .NET 3.5. и в Visual Studio 2008.

Сочетание WCF и WF образует стабильную платформу для определения многошаговых процессов, которые раскрываются клиенту надежным и безопасным способом. Существует встроенная модель активации, позволяющая запускать потоки работ в ответ на полученное сообщение. Имеется также встроенная модель сохранения состояния потока работ между выполнением отдельных операций. А поддержка со стороны диспетчера обеспечивает возможность одновременного выполнения нескольких экземпляров потока работ и доставку входящих сообщений нужному экземпляру.

Важно: для понимания излагаемого материала вы должно быть хорошо знакомы с WF. Эта глава не является учебником по WF, а посвящена интеграции WCF и WF с помощью средств, имеющихся в Visual Studio и .NET 3.5. Отличным руководством по WF может служить книга Dharma Shukla and Bob Schmidt «*Essential Windows Workflow Foundation*» (Addison-Wesley; ISBN 0-321-39983-8)[2]. Кроме

[1] Терминология здесь неоднозначна. В WCF под *activity* понимается логическая группировка трассировочных записей, и в этой книге переводится как *деятельность*. В WWF *activity* – это логическая операция процесса и переводится как *операция*. В тех случаях, когда контекст не позволяет однозначно понять смысл термина, приводится англоязычный оригинал. (Прим. перев.)

[2] Имеется русский перевод Дхарма Шукла, Боб Шмидт «Основы Windows Workflow Foundation», ДМК-Пресс, 2008.

того, поскольку интеграция WCF и WF опирается на привязки и поведения, вы должны быть знакомы с содержанием глав 4 и 5 этой книги.

Точки интеграции

Описать интеграцию между WCF и WF можно двояко. С точки зрения WCF, технология WF позволяет «реализовать службу в виде потока работ». А с точки зрения WF, WCF позволяет вам «представить поток работ в виде службы». Это просто разные способы выражения одной и той же мысли: комбинируя WCF и WF, вы моделируете и реализуете логику, раскрываемую с помощью стандартизованного интерфейса. Логика может быть совместима со многими стандартами, включая SOAP, JSON и X.509, и размещена в IIS, WAS, Windows Service или любой поддерживаемой WCF среде размещения. Все инструменты трассировки, диагностики и автономного тестирования, имеющиеся в Visual Studio, в полной мере применимы к WCF и WF.

WF интегрируется с WCF за счет механизмов расширения WCF. Таким образом, WF знает о WCF, тогда как WCF ничего не знает о WF. В .NET 3.5 интеграция достигается за счет глубокого внедрения WF в точки расширения WCF. Инструменты визуального моделирования WF, включенные в Visual Studio, модернизированы, и добавлена поддержка WCF со стороны исполняющей среды.

Для интеграции WCF с WF необходимы три элемента. Во-первых, нужен способ моделирования взаимодействий со службой. WF прекрасно справляется с определением потока работ в виде набора операций, поэтому остается лишь распространить операции на интероперабельные службы. После этого можно использовать WF для моделирования взаимодействий между службами. Во-вторых, требуется где-то разместить инфраструктуру активации, чтобы поток работ можно было представить в виде службы. При этом необходимо поддержать событийно-управляемые модели, транзакции и сохранение состояния, так чтобы поток мог «пережить» перезагрузку системы. В-третьих, требуется поддержать корреляцию между клиентами и службами, чтобы клиент мог взаимодействовать с нужным ему экземпляром службы в условиях, когда одновременно работают тысячи служб.

Механизм интеграции, встроенный в .NET 3.5 и Visual Studio 2008, поддерживает все эти требования. Ниже мы дадим его краткий обзор, а в последующих разделах рассмотрим детали более подробно.

- В конструктор WF добавлены операции Send и Receive.
- Класс `WorkflowServiceHost` обертывает класс владельца WCF.
- Новые привязки и поведения добавляют в канал контекстную информацию, необходимую для поддержки корреляции и протяженных потоков работ.

Операция Send применяется для отправки сообщения оконечной точке WCF-службы. На этапе проектирования конструктор WF отображает типы входящих и исходящих сообщений в сгенерированном WCF прокси-классе на переменные WF. На этапе исполнения WF пользуется прокси-классом для коммуникации с оконечными точками службы. Операция Receive противоположна Send. Она не

является клиентом существующей службы, а раскрывает поток работ как самостоятельную службу. На этапе проектирования конструктор WF используется для описания оконечной точки и контрактов об операциях (operation) этой службы потока работ, в том числе и для описания получаемых и отправляемых сообщений. На этапе выполнения, когда оконечная точка получает сообщение, происходит одно из двух: либо создается новый экземпляр потока работ, либо сообщение доставляется уже работающему потоку.

Класс `WorkflowServiceHost` наследует классу `ServiceHost`, являющемуся частью WCF, и с точки зрения интеграции WF с WCF является лишь вершиной айсберга. Для решения специфичных для WF задач корреляции и управления созданием экземпляров он пользуется специальными привязками и поведениями. Кроме того, он предоставляет доступ к исполняющей среде WF, размещенной внутри владельца службы, так чтобы провайдер сохранения (persistence provider) мог организовать выполнение протяженных процессов в WCF-службе без запоминания состояния.

В главе 5 отмечалось, что поведения являются очень гибкой точкой расширения в архитектуре WCF. Для действий с сообщениями и экземплярами WCF применяет поведения службы и операций (рис. 11.1).

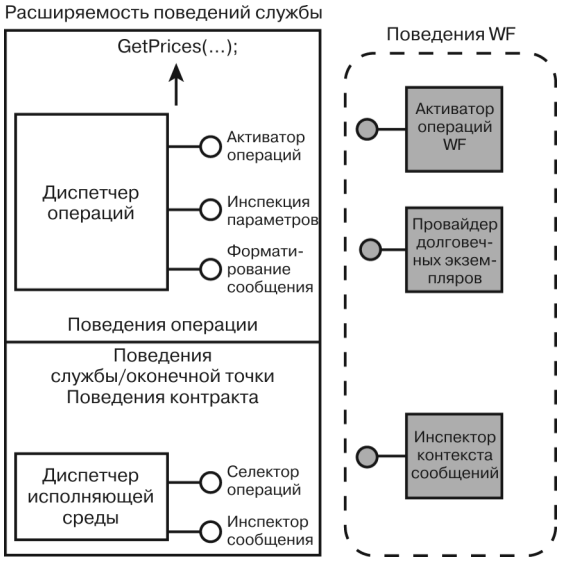

Рис. 11.1. Поведения WF

Операции WF – это кирпичики, из которых составляются потоки работ. Обычно одна операция представляет одну задачу, например, вызов метода класса, вызов Web-службы или запуск другой программы. В комплект поставки WF входят несколько готовых операций, а разработчики могут создавать дополнительные, наследуя одному из базовых классов операций.

Существует по меньшей мере четыре способа вызвать службу из WF: воспользоваться операцией Send, написать заказную операцию, воспользоваться операцией InvokeWebService или операцией Code. Для вызова WCF или других интероперабельных Web-служб оптимальна операция Send, именно для этих целей она и включена в состав .NET 3.5. Заказные операции – это облегченный механизм инкапсуляции кода, допускающего повторное использование в разных потоках работ. Операция InvokeWebService полезна для вызова ASMX-служб, но по сравнению с Send не имеет никаких преимуществ. Операция Code – это, как следует из самого названия, просто код. В этой главе мы рассмотрим только операцию Send и написание заказной операции. Сначала мы обратимся к Web-службе с помощью Send, а затем напишем вместо нее совсем простую заказную операцию.

В листинге 11.1 приведена служба-заглушка, к которой мы будем обращаться в этой главе. Метод `GetPrice` принимает на входе значение простого типа, а возвращает объект составного типа `StockPrice`.

Создайте проект в Visual Studio 2008, воспользовавшись шаблоном Sequential Workflow Console Application (Консольное приложение с последовательным потоком работ) в папке Workflow. Шаблон включает файл класса (`Workflow1.cs`), в котором реализован поток работ, и главную программу (`Program.cs`), которая инициализирует исполняющую среду WF и запускает экземпляр потока работ.

Листинг 11.1. Служба StockService, вызываемая из WF

```
[DataContract]
class StockPrice
{
   [DataMember] public double price;
   [DataMember] public int volume;
}

[ServiceContract]
public class StockService
{
   [OperationContract]
   private StockPrice GetPrice(string ticker)
      {...}
}
```

Использование операции Send

Send – одна из операций, встроенных в Visual Studio 2008. Она предназначена для вызова Web-службы с помощью прокси-класса WCF. Среда разработки позволяет задать общие свойства, необходимые для конфигурирования обращения к Web-службе, в том числе и информацию об оконечных точках. Для связывания переменных WF с параметрами операции службы также используется страница свойств и конструктор. Кроме того, операция WF может переопределить URI-адрес службы. Перед тем как вызывать Web-службу, необходимо включить в проект прокси-класс для нее. Его можно сгенерировать инструментом Add Service Reference в Visual Studio или с помощью утилиты `svcutil.exe`.

Чтобы воспользоваться операцией Send, перетащите ее из инструментария на поверхность конструирования потока работ так же, как любую другую операцию WF. На рис. 11.2 показан конструктор потока работ, в который уже добавлена операция Send.

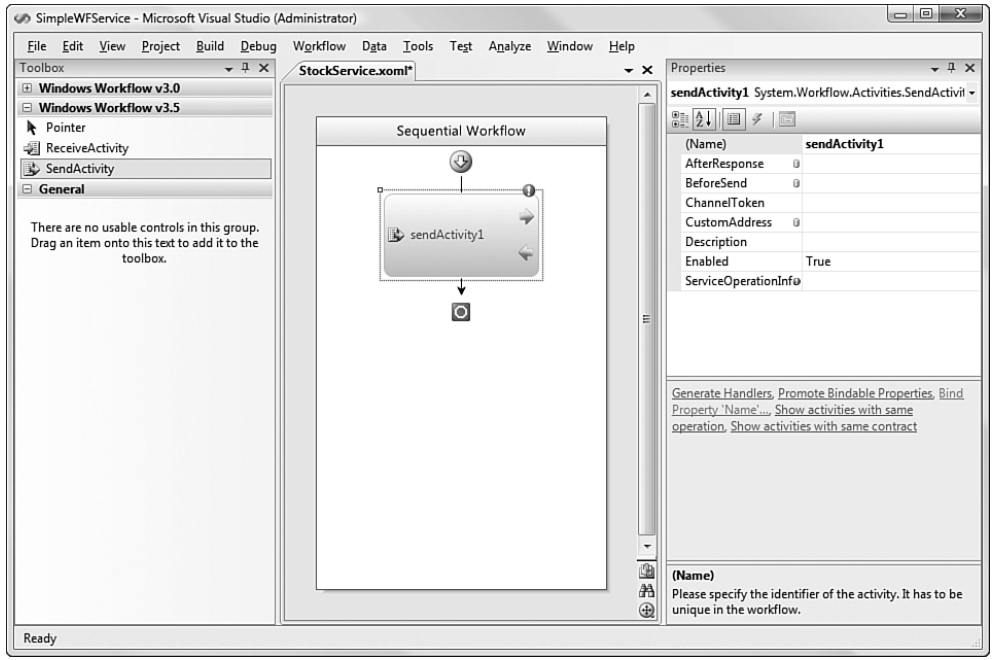

Рис. 11.2. Добавление операции Send на поверхность конструирования потока работ

Операцию Send необходимо сконфигурировать, указав вызываемую операцию службы, переменные WF, через которые мы будем получать входящие сообщения и отправлять исходящие, а также имя оконечной точки.

Первым делом определим операцию службы, задав свойство `ServiceOperation`. После щелчка по многоточию на странице свойств появляется диалоговое окно, в котором вы выбираете операцию службы. Если вы впервые обращаетесь к службе из данного потока работ, то понадобится импортировать в WF типы из сгенерированного WCF прокси-класса. Нажмите кнопку Import и найдите в проекте файл прокси-класса. На рис. 11.3 показано, как выглядит диалог после импорта типа прокси из проекта WF. В данном случае мы выбрали операцию `GetPrice`. Когда операция службы становится известна, в окне свойств появляются ее параметры и возвращаемое значение.

Следующий шаг – связать переменные WF с вызовом прокси. Если щелкнуть по многоточию рядом с именем переменной (в данном случае `Ticker` и `ReturnValue`), то откроется окно, в котором можно будет выбрать или создать переменные WF

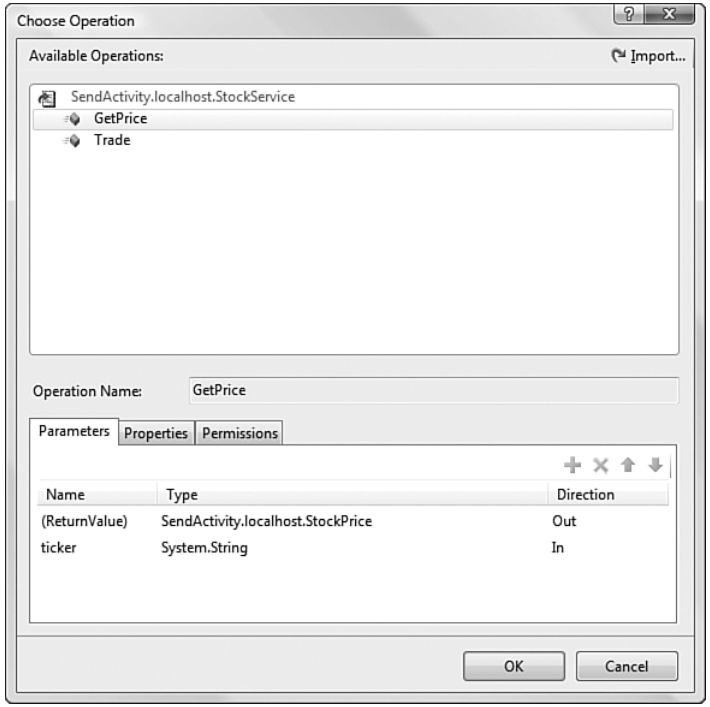

Рис. 11.3. Тип прокси-класса импортирован, можно выбирать операцию службы

подходящего типа. Допускается как поле, так и свойство WF; поле локально по отношению к классу потока работ, а у свойства более широкая область видимости в среде разработки WF.

Свойство или поле? Свойство – это поле, которое можно инициализировать в момент запуска экземпляра потока работ. Можно отправить потоку работ значение, определив объект-словарь и передав его в качестве необязательного второго параметра методу `workflowRuntime.CreateWorkflow`. Если вы определяете свойство, то переданное потоку работ значение можно передавать и далее при вызовах служб или других операций WF (activity).

На рис. 11.4 показана переменная WF `sendActivity1_ReturnValue_1`, связанная со значением, которое возвращает операция службы `GetPrice`.

Наконец, вы должны сконфигурировать свойства оконечных точек службы, чтобы операция Send могла отформатировать и послать сообщение в нужное место. Для этого следует задать конфигурацию свойства `ChannelToken`, которое состоит из трех компонентов: `Name`, `EndpointName` и `OwnerActivityName`. Компонент `OwnerActivityName` определяет контекст `ChannelToken` и выбирается из списка. Значение `EndpointName` должно совпадать с именем оконечной точки

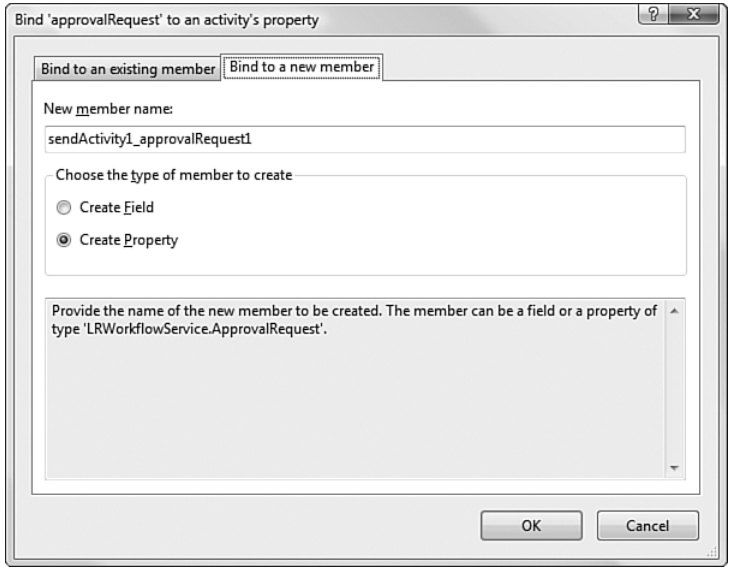

Рис. 11.4. Связывание переменной WF с параметрами операции службы

в файле `app.config`, который был сгенерирован с помощью Add Service Reference или `svcutil.exe`.

Написание заказной операции WF

В WF заказные операции применяются для инкапсуляции бизнес-функций. Позаботившись о надлежащем уровне абстракции и гранулярности заказных операций, разработчик может моделировать приложение, сочетая их между собой. Хотя операция Send очень удобна для вызова любой интероперабельной Web-службы, для ее применения разработчик WF должен знать, что необходимая ему бизнес-функция в действительности является Web-службой. Заказные операции инкапсулируют это знание, позволяя моделировать само приложение, отвлекаясь от его инфраструктуры.

В простейшем случае заказная операция представляет собой класс .NET, производный от `System.Workflow.ComponentModel.Activity`. Ему наследуют многие подклассы, чтобы специализировать его поведение для последовательной или основанной на конечном автомате модели либо для порождения составных операций. Необходимо переопределить единственный метод этого класса – `Execute`. Он возвращает одно из значений, определенных в перечислении `ActivityExecutionStatus`. Если оно равно `Closed`, значит, операция WF завершилась. В противном случае WF управляет экземпляром операции до тех пор, пока та не известит исполняющую среду WF о том, что завершилась.

Заказные операции WF могут раскрывать свои свойства конструктору WF. Свойства можно задать в Visual Studio на этапе проектирования и связать их с переменными WF, доступными на этапе выполнения. Свойства – это, по существу, интерфейс с заказной операцией. Метод `Execute` требует в качестве параметра только контекст, поскольку интерфейс определяется свойствами. Чтобы создать заказную операцию, воспользуйтесь шаблоном Workflow Activity Library в Visual Studio 2008. Он создает операции типа `SequenceActivity`, производного от `Activity`.

В листинге 11.2 приведен код заказной операции. Отметим, что, помимо конструктора, у нее имеется всего один метод `Execute`. Именно сюда вставляется код для вызова WCF-службы. В данном примере мы создаем объект прокси-класса и вызываем его метод. Одновременно с генерацией прокси-класса WCF создала в локальном проекте файл `app.config`. На этапе выполнения, когда вызывается операция WF, WCF ищет в конфигурационном файле приложения секцию `<service.model>`. Поэтому всю эту секцию необходимо скопировать в файл `app.config`, ассоциированный с владельцем потока работ.

В классе определены два свойства: `ticker` и `price`. Оба доступны конструктору WF в Visual Studio и программе во время выполнения.

Листинг 11.2. Реализация заказной операции WF

```
namespace MyActivityLibrary
{
   public partial class GetPriceActivity: Activity
   {
      public GetPriceActivity(
      {
         InitializeComponent();
      }
      protected override ActivityExecutionStatus
                Execute(ActivityExecutionContext context)
         {
            localhost.StockServiceClient proxy =
               new ActivityLibrary1.localhost.StockServiceClient();
            price = proxy.GetPrice(ticker);
            return ActivityExecutionStatus.Closed;
      }

   public static DependencyProperty tickerProperty =
                 DependencyProperty.Register("ticker",
                 typeof(System.String),
                 typeof(GetPriceActivity));
   [DescriptionAttribute("Please specify a stock symbol ")]
   [DesignerSerializationVisibilityAttribute
         (DesignerSerializationVisibility.Visible)]
   [BrowsableAttribute(true)]
    public string ticker
      {
         get
         {
            return ((String)
```



```
                  (base.GetValue(GetPriceActivity.tickerProperty)));
         }
         set
         {
            base.SetValue(GetPriceActivity.tickerProperty, value);
         }
      }

      public static DependencyProperty priceProperty =
                           DependencyProperty.Register("price",
                           typeof(localhost.StockPrice),
                           typeof(GetPriceActivity));
      [DescriptionAttribute("tradePrice")]
      [DesignerSerializationVisibilityAttribute
               (DesignerSerializationVisibility.Visible)]
      [BrowsableAttribute(true)]
      public localhost.StockPrice price
      {
         get
         {
            return ((localhost.StockPrice)
                  (base.GetValue(GetPriceActivity.priceProperty)));
         }
         set
         {
            base.SetValue(GetPriceActivity.priceProperty, value);
         }
      }
   }
}
```

После создания библиотеки на операцию можно ссылаться из проекта WF. Поток работ ничего не знает о том, что задействована WCF, так как вся инфраструктура инкапсулирована в заказной операции. На рис. 11.5 показан поток работ, в который включена операция `GetPrice`.

Раскрытие службы из WF

Обычно WF применяется для написания *реактивных программ*. Программа запускается, делает что-то полезное, ожидает входных данных, снова что-то делает, опять ждет данных и т.д. В какой-то детерминированной точке программа завершается. Реактивные программы могут работать очень длительное время, в течение которого клиентский или серверный компьютер могут не раз перезагружаться. Кроме того, одновременно могут работать несколько экземпляров потока работ, и у каждого из них должен быть уникальный адрес, иначе экземпляр не сможет получить входные данные. Такие характеристики присущи не только WF, но так или иначе они требуют особого внимания.

Чтобы удовлетворить этим ключевым требованиям, среда разработки и исполнения WF подключается к точкам расширения WCF. Она обрабатывает протяженные потоки работ, «переживающие» сбой системы. Она коррелирует входящие сообщения с существующими потоками работ, так что масштабируемый

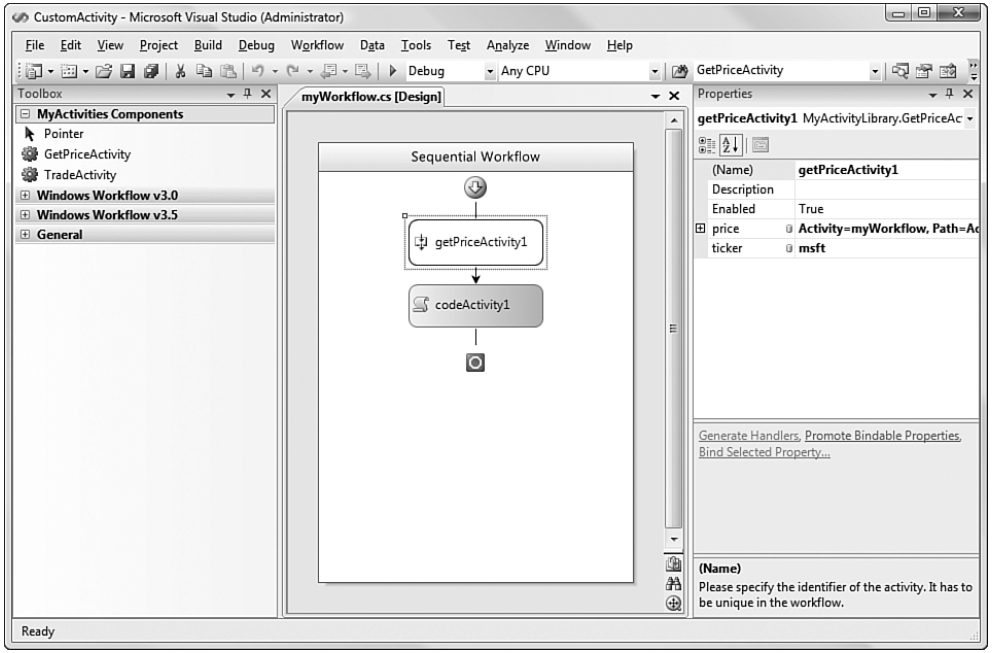

Рис. 11.5. Включение заказной операции в поток работ

владелец способен поддержать одновременное исполнение многих экземпляров. Она раскрывает оконечные точки службы из потока работ, чтобы для коммуникации можно было применять стандартизованные сообщения.

В этом и последующих разделах мы рассмотрим, каким образом WF раскрывает потоки работ в виде служб. Начнем с простого потока. Мы определим интерфейс на языке C# и импортируем его в конструктор WF. Затем мы сконфигурируем операцию Receive, которая будет раскрывать поток работ в виде оконечной точки службы, обладающей единственной операций типа запрос-ответ. Далее мы сделаем этот поток многошаговым и на одном из шагов будем ожидать поступления сообщений. Одновременно мы определим вторую операцию (operation) службы, соответствующую второй операции (activity) Receive. Мы покажем, как клиент может направлять сообщения конкретному экземпляру WF, и напоследок встроим в поток поддержку долговечности, позволяющую ему пережить отказ системы.

Определение интерфейса

Чтобы построить поток работ, реализующий конкретный интерфейс, мы создали в Visual Studio новый проект по шаблону *Sequential Workflow Service Library*, который находится в папке WCF. И проект, и решение мы назвали SimpleService. Мы удалили файлы `IWorkflow1.cs` и `Workflow1.cs` и определили интерфейс

`IStockService`, как показано в листинге 11.3. Отметим, что пространство имен в этом файле названо `SimpleService`. В файле `app.config` этого проекта необходимо в качестве имени контракта для оконечной точки употреблять полное имя интерфейса `SimpleService.IStockService`.

В контракте о службе есть две операции: `InitiateTrade` и `Approval`. `InitiateTrade` размещает фиктивный заказ на покупку акций и возвращает номер подтверждения в структуре `TradeRequestStatus`. Операция `Approval` вызывается после того, как вызывающий подозрения заказ был рассмотрен и одобрен. Она возобновляет поток работ, который все это время ожидает внешнего события.

Показанный в листинге 11.3 интерфейс находится в файле `IStockService.cs`. Именно его раскрывает служба данного потока работ. Можно было определить этот интерфейс и в конструкторе WF, если вам ближе стиль «код сначала». Начав с определения интерфейса, мы могли бы опубликовать его, обсудить с другими разработчиками, а потом использовать в качестве отправной точки для реализации службы. С чего бы ни начинать – с интерфейса или с конструктора, – результат будет одинаков, но нам кажется, что более ответственный подход к созданию служб – начинать с контракта.

Листинг 11.3. Интерфейс службы, раскрываемый из WF

```
namespace SimpleService {
   [ServiceContract]
   public interface IStockService
   {
      [OperationContract]
      TradeRequestStatus InitiateTrade(TradeRequest tradeRequest);

      [OperationContract]
      void Approval(ApprovalRequest approvalRequest);
   }

   [DataContract]
   public class TradeRequest
   {
      [DataMember] public string account;
      [DataMember] public string action;
      [DataMember] public string ticker;
      [DataMember] public double price;
   }

   [DataContract]
   public class ApprovalRequest
   {
      [DataMember] public string tradeNumber;
      [DataMember] public string approval;
      [DataMember] public string reason;
   }

   [DataContract]
   public class TradeRequestStatus
   {
```

```
        [DataMember] public string confirmationNumber;
        [DataMember] public string status;
    }
}
```

Операция Receive

Операция (activity) Receive моделирует контракт об операции (operation). Поскольку в листинге 11.3 определены два контракта об операции, то в потоке работ должно быть две операции Receive. Свойствами Receive будут входящие и исходящие сообщения, определенные в контракте об операции.

Чтобы реализовать поток работ, мы добавили новый элемент, пользуясь шаблоном Sequential Workflow (в режиме отделения кода – Code Separation). В этом режиме мы можем посмотреть представление потока на языке XOML; это диалект XML, на котором описывается поток работ и составляющие его операции. На рис. 11.6 показан внешний вид конструктора WF после того, как мы открыли файл StockService.xoml и поместили на поверхность конструирования операцию Receive.

Выберем свойство `ServiceOperationInfo` и щелкнем по значку многоточия, откроется диалог для выбора или определения контракта. При первом запуске этого диалога еще не определено ни одного контракта об операции. После нажа-

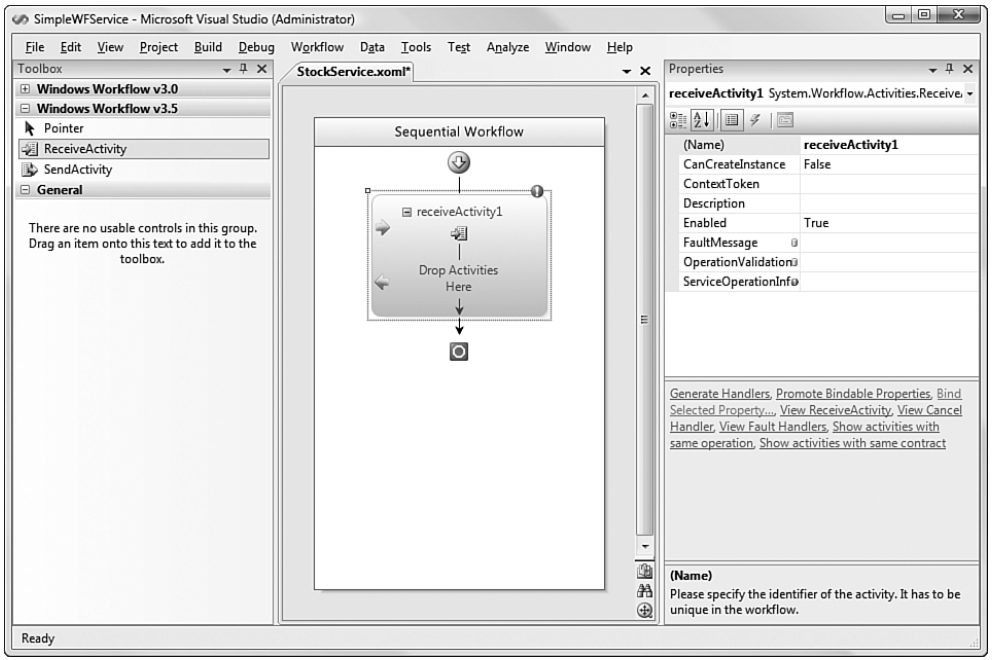

Рис. 11.6. Добавление операции Receive на поверхность конструирования потока работ

тия кнопки Import (рис. 11.7) будут показаны все включенные в проект классы и сборки с атрибутом `[ServiceContract]`, на которые проект ссылается. Импортировать из них можно только методы, помеченные атрибутом `[OperationContract]`. Ввести определение контракта можно непосредственно в этом диалоге, а можно выбрать уже существующий в проекте класс. Именно для этой цели мы ранее включили интерфейс `IStockService`; на рис. 11.7 показано, как теперь выглядит диалоговое окно. Как видите, в список попали две операции, которым в качестве аргументов передаются составные типы .NET.

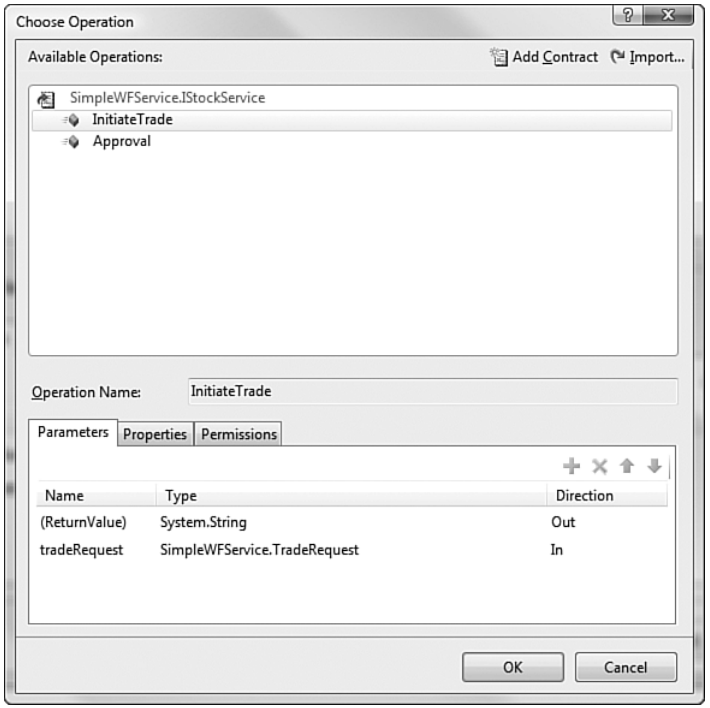

Рис. 11.7. Ввод или импорт интерфейса для операции Receive

WF-программа – это самая обычная программа, поэтому превращение ее в службу посредством интеграции с WCF наделяет ее всеми возможностями WCF, доступными любой другой программе. Когда WCF получает сообщение, она десериализует его, преобразуя в тип .NET, и передает подходящему методу класса. В WF-программе значение этого типа присваивается локальной переменной потока работ.

Любой параметр, определенный в контракте об операции, должен быть связан с каким-то полем или свойством WF. Для этой цели предназначен диалог, где вы можете выбрать существующую либо создать новую переменную (поле или свойство) для связывания с параметром. Этот диалог открывается в результате щелч-

ка по многоточию справа от имени параметра на странице свойств операции Receive (рис. 11.8).

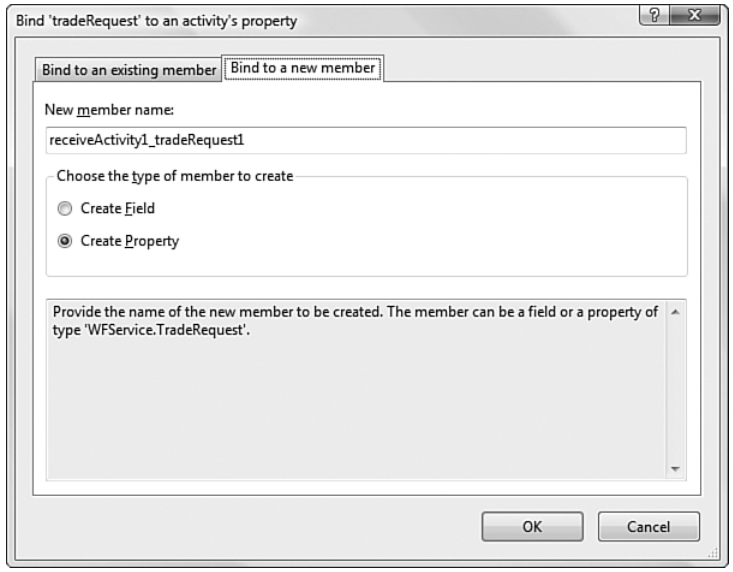

Рис. 11.8. Связывание параметров контракта об операции с переменными потока работ

Теперь интерфейс службы полностью определен, хотя сама служба еще ничего не умеет делать. Следующий шаг – изменить операцию Receive, включив в нее операцию Code, которая будет возвращать осмысленный ответ клиенту. Операцию `Approval` мы реализуем ниже. В данном примере значению, которое должно быть возвращено в соответствии с контрактом об операции, присваивается некий текст, служащий номером подтверждения. Отметим, что Receive – это составная операция, поэтому в нее можно поместить любую другую операцию, и та будет выполнена перед отправкой ответа клиенту. Оставляя в Receive минимум кода, мы стремимся к тому, чтобы операция службы как можно быстрее возвращала ответы клиентам.

Окончательная конфигурация потока работ показана на рис. 11.9. Обратите внимание, что возвращаемое значение и входной параметр операции `InitiateTrade` связаны с переменными. Также заметьте, что свойство `CanCreateInstance` установлено в `true`. Это говорит владельцу, что необходимо запускать новый экземпляр службы в исполняющей среде WF всякий раз, как поступает сообщение, не ассоциированное с имеющимися экземплярами.

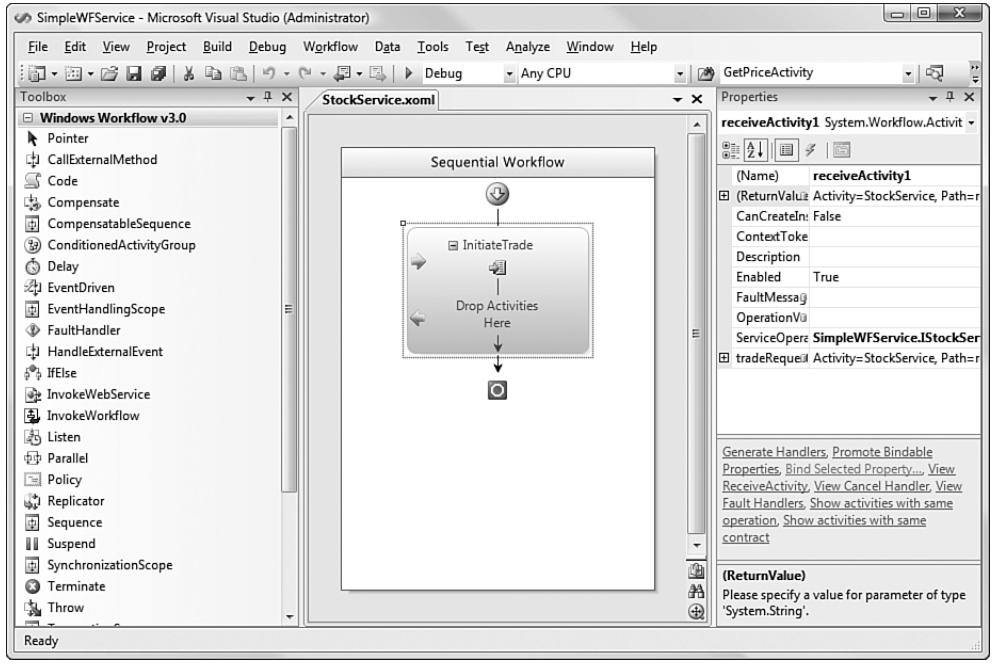

Рис. 11.9. Рабочий поток с единственной операцией Receive

Задание конфигурации в файле app.config

Как и в большинстве WCF-служб, информация об оконечных точках, безопасности и поведениях вынесена из программы и хранится в файле `app.config` или `web.config`.

Создавая проект по шаблону Sequential Workflow Service, Visual Studio включила в него файл `app.config`. Но поскольку в шаблоне предполагались имена `IWorkflow1` и `Workflow1` для интерфейса и класса соответственно, а мы заменили их на `IStockService` и `StockService`, то файл `app.config` придется подправить. В листинге 11.4 приведены модифицированные настройки WCF в файле `app.config`. На некоторые параметры стоит обратить особое внимание.

Прежде всего, само имя службы. Оно должно соответствовать имени класса, который реализует службу. Поскольку в файле `StockService.XMOL.cs` мы заменили полное имя класса `SimpleWFService.`**`Workflow1`** на `SimpleWFService.`**`StockService`**, то нужно подправить и имя службы в `app.config`. То же относится к имени контракта в оконечной точке, которое должно соответствовать полному имени интерфейса в файле `IStockService.cs`.

И еще не забудьте про привязку для оконечной точки. По умолчанию используется привязка `wsHttpContextBinding`. Она содержит элемент `ContextBindingElement` и поддерживает сеансы, что необходимо для протяженных потоков работ. Элемент привязки `ContextBindingElement` поддерживается тремя привязками, поставляемыми в комплекте с WF: `wsHttpContextBinding`, `basicHttpContextBinding` и `netTcpContextBinding`. Но этот элемент можно включать и в заказные привязки, чтобы они были в состоянии взаимодействовать с потоками работ, выступающими в роли служб. Отметим, что `ContextBindingElement` не поддерживает односторонние операции, поэтому в привязки к MSMQ он входить не может. Дополнительную информацию о создании заказных привязок см. в главе 4.

Листинг 11.4. Файл app.config для потока работ, наделенного возможностями службы

```
<system.serviceModel>
   <services>
      <service name="SimpleWFService.StockService"
            behaviorConfiguration="SimpleWFService.Workflow1Behavior">
         <host>
            <baseAddresses>
               <add baseAddress="http://localhost:8080/Workflow1" />
            </baseAddresses>
         </host>
         <endpoint address=""
                   binding="wsHttpContextBinding"
                   contract="SimpleWFService.IStockService" />
         <endpoint address="mex"
                   binding="mexHttpBinding"
                   contract="IMetadataExchange" />
      </service>
   </services>
   <behaviors>
      <serviceBehaviors>
         <behavior name="SimpleWFService.Workflow1Behavior">
            .
            .
            .
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

Размещение потока работ, наделенного возможностями службы

Размещать WCF-службы можно по-разному. Подробно эта тема рассматривается в главе 7, а здесь лишь напомним, что владельцем службы может быть любой процесс, в котором создается экземпляр класса `ServiceHost`. Чаще всего в качестве владельца служб используется IIS. В WCF поддержка сервера IIS встроена,

поэтому размещать в нем службы и управлять ими сравнительно несложно. В операционных системах Vista и Windows Server 2008 появилась подсистема Windows Activation Services (WAS), которая обобщает многие механизмы IIS на протоколы, отличные от HTTP. Поэтому размещение служб в WAS – не менее надежный и удобный способ. WCF-службы нередко размещают и в службах Windows (иначе говоря, NT-службах), поскольку системным администраторам хорошо знаком интерфейс для работы с ними. Для тестирования и в разного рода необычных ситуациях WCF-службы можно размещать также в консольных или в Windows-приложениях.

Вне зависимости от среды размещения, класс `ServiceHost` добавляет оконечные точки в описание службы в момент ее запуска. На этапе выполнения `ServiceHost` настраивает прослушиватели каналов, описанные в привязках, так чтобы они ожидали поступления входящих сообщений. `ServiceHost` получает описание службы из разных источников: элемента `<servicemodel>` в конфигурационном файле; атрибутов, заданных в коде (`[ServiceContract]`, `[ServiceBehavior]`, `[OperationContract]` и т.д.) и из определений классов. Экземпляром `ServiceHost` можно манипулировать и программно. Программа-владелец тоже может добавлять в службу поведения, управляющие тем, что происходит в момент получения или отправки сообщений, а также стратегией создания и уничтожения экземпляров службы. Сообщения, полученные прослушивателем канала, доставляются описанным в контракте операциям службы для обработки. Значительная часть этого алгоритма реализована в виде поведений (детали см. в главе 5).

Размещение потока работ, наделенного возможностями службы, происходит точно так же, как и для любой другой службы. Для размещения таких потоков WF предоставляет новый класс `WorkflowServiceHost`, производный от `ServiceHost`, в котором учтены особенности WF. Если конструктор `ServiceHost` принимает в качестве аргумента объект любого класса, помеченного атрибутом `[ServiceContract]`, то конструктору `WorkflowServiceHost` необходима еще информация, относящаяся к созданию потока работ (типы, производные от `Activity` или потока/файла XAML). Класс `WorkflowServiceHost` добавляет в службу три поведения: `WorkflowServiceBehavior`, `WorkflowOperationBehavior` и `WorkflowRuntimeBehavior`, а также `DurableInstanceProvider` и `MessageContextInspector`. Вместе они управляют созданием экземпляров службы и занимаются диспетчеризацией – доставкой входящих сообщений конкретным потокам работ, а исходящих – клиентам. Кроме того, `WorkflowServiceHost` проверяет, что привязки, указанные для оконечной точки службы, включают элемент `ContextBindingElement`.

Авторазмещение потока работ, наделенного возможностями службы

В листинге 11.5 приведен код минимального консольного приложения с авторазмещением потока работ, наделенного возможностями службы. Сравнив его с листингом 1.2 из главы 1, вы убедитесь, что они практически идентичны. Един-

ственное различие состоит в том, что вместо класса ServiceHost используется WorkflowServiceHost.

Листинг 11.5. Авторазмещение потока работ, наделенного возможностями службы

```
WorkflowServiceHost serviceHost =
           new WorkflowServiceHost(typeof(StockService));

serviceHost.Open();

Console.WriteLine("Службы готовы. Для завершения нажмите <ENTER>.");
Console.ReadLine();

serviceHost.Close();
```

В листинге 11.6 приведен конфигурационный файл для авторазмещаемого потока работ. Единственное отличие от листинга 1.5 в главе 1 – это включение в поведение аутентификационной информации, которая необходима для обеспечения безопасности службы потока работ. О безопасности мы поговорим ниже в этом разделе.

Замечание: файл app.config создан Visual Studio в проекте типа Sequential Workflow Library. Его необходимо перенести из проекта библиотеки в проект консольного приложения, показанного в листинге 11.5, поскольку WorkflowServiceHost будет искать конфигурационный файл в собственной папке.

Листинг 11.6. Конфигурация службы потока работ

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="SimpleWFService.StockService"
                  behaviorConfiguration="SimpleWFService.Workflow1Behavior">
            <host>
               <baseAddresses>
                  <add baseAddress="http://localhost:8000/EffectiveWCF" />
               </baseAddresses>
            </host>
            <endpoint address=""
                      binding="basicHttpContextBinding"
                      contract="SimpleWFService.IStockService" />

            <endpoint address="mex"
                      binding="mexHttpBinding"
                      contract="IMetadataExchange" />
         </service>
      </services>
      <behaviors>
         <serviceBehaviors>
            <behavior name="SimpleWFService.Workflow1Behavior">
               <serviceMetadata httpGetEnabled="true" />
```



```
               <serviceCredentials>
                  <windowsAuthentication
                     allowAnonymousLogons="false"
                     includeWindowsGroups="true" />
               </serviceCredentials>
            </behavior>
         </serviceBehaviors>
      </behaviors>
   </system.serviceModel>
</configuration>
```

Размещение потока работа, наделенного возможностями службы, в IIS

Поток работ с возможностями службы размещается в IIS, как и любая другая служба. Шаги, необходимые для размещения службы в IIS, описаны в главе 1. Понадобится приложение IIS, SVC-файл с описанием способа создания экземпляра службы, файл web.config, в который включена секция <servicemodel>, и файл реализации в папке /bin виртуального корня приложения.

В листинге 11.7 показан минимальный SVC-файл, ассоциированный со службой потока работ. Он очень похож на файл из листинга 1.9 в главе 1. Единственное отличие – это элемент Factory, который говорит IIS о необходимости использовать класс WorkflowServiceHostFactory вместо ServiceHostFactory при создании экземпляра ServiceHost.

Листинг 11.7. SVC-файл для размещения в IIS потока работа, наделенного возможностями службы

```
<%@ ServiceHost
      Service="SimpleWFService.StockService"
      Factory="System.ServiceModel.Activation.WorkflowServiceHostFactory" %>
```

В файл web.config включена секция <servicemodel>, содержащая ту же информацию, что в листинге 11.6, за исключением элемента <host>, поскольку базовый адрес службы в IIS определяется виртуальным корнем приложения.

Корреляция и долговечность

Потоки работ часто применяются для моделирования бизнес-транзакций. Как и в реальном мире, транзакция может занимать несколько секунд (перевод денег с помощью банкомата) или куда более длительное время (покупка чего-то на сайте eBay, оплата, получение товара и оставление отзыва). Транзакция моделируется один раз, а одновременно исполняться могут тысячи ее экземпляров. На протяжении времени работы транзакции клиент, сервер или сеть могут оказываться недоступными или перезагружаться.

Для поддержки протяженных транзакций необходимы две вещи: корреляция и долговечность. Корреляция – это способ, с помощью которого клиент указывает тот экземпляр рабочего потока, с которым хочет общаться. Долговечность позво-

ляет экземпляру рабочего потока «пережить» отказ системы, а исполняющей среде – эффективно распоряжаться памятью и процессором. Класс `WorkflowServiceHost` реализует расширения, необходимые для поддержки корреляции и долговечности. Для этого он использует два элемента: класс контекста, который помещается в канал между клиентом и службой, и провайдера долговечных экземпляров, который может «сдуть» (записать из памяти на диск) или «надуть» (прочитать с диска в память) экземпляр рабочего потока.

Контекст, передаваемый между клиентом и службой, однозначно идентифицирует экземпляр рабочего потока. Когда клиент посылает сообщение службе рабочего потока, та проверяет в нем наличие контекста. Если контекста нет, то в исполняющей среде WF создается новый экземпляр рабочего потока. В противном случае сообщение передается уже существующему экземпляру. Исполняющая среда WF смотрит, загружен ли данный экземпляр в память, и, если нет, вызывает провайдера долговечных экземпляров, чтобы тот прочитал экземпляр с диска. Затем сообщение десериализуется в тип .NET, передается исполняющей среде WF и доставляется нужному экземпляру.

Отметим, что исполняющая среда WF целиком погружена в единственный экземпляр WCF-службы и отвечает за создание и сохранение экземпляров потока работ. Иными словами, если одновременно выполняются 50 экземпляров потока работ, то посылаемые им сообщения поступают в единственный экземпляр исполняющей среды WF, у которой есть собственный внутренний механизм корреляции и ведения очередей сообщений, предназначенных каждому экземпляру. Чтобы гарантировать присутствие нужного экземпляра в памяти, используется хранящийся в контексте идентификатор экземпляра (Instance ID).

Протяженный поток работ

В этом разделе мы смоделируем бизнес-процесс, который принимает и исполняет заказы на покупку/продажу акций. На ранней стадии проверяется, не является ли заказ криминальным. Если сделка вызывает подозрения, она направляется аналитику для более детального изучения. Если аналитик дает добро, заказ принимается, иначе отвергается. Большинство экземпляров потока работ в этом примере завершаются в течение нескольких секунд, поскольку, как правило, сделки честные. Но, если попадается запрос, требующий ручного анализа, то процедура может занимать от нескольких минут до нескольких часов. В течение оживленного биржевого дня могут потребовать анализа сотни подозрительных сделок. Для каждой сделки процедура одна и та же, но детали (входные и выходные данные) различаются.

Предположим, что воображаемый трейдер пользуется Web-приложением или «толстым» клиентом, который вызывает метод `InitiateTrade` для регистрации новой сделки. Если в ответ получен статус `executed`, сделка завершена. Если же статус оказался `review`, то сделка не завершена, и трейдер ожидает получить в ближайшее время то или иное уведомление (в нашем примере уведомление не присутствует, но это могло бы быть сообщение, отправленное по электронной почте, голосом, в виде SMS или по интернет-пейджеру). В случае статуса `review` находящийся где-то финансовый аналитик получает извещение о том, что такая-то сделка требует анализа. Аналитик открывает приложение, в котором просматривает запрос, и в конечном итоге вызывает метод `Approval`, указывая, должна ли система выполнить его.

Вспомните созданный для этого сценария интерфейс (листинг 11.3). В нем описаны форматы трех сообщений – `TradeRequest`, `TradeRequestStatus`, `ApprovalRequest` – и две операции службы. Клиент посылает сообщение `TradeRequest`, когда хочет инициировать новую сделку, и сообщение `ApprovalRequest` – чтобы одобрить или отвергнуть подозрительную сделку. Все взаимодействие клиента с потоком работ, начиная с создания экземпляра и до завершения, исчерпывается этими сообщениями. В интерфейсе описаны также две операции службы: `InitiateTrade` и `Approval`, которые посылают сообщения.

На рис. 11.10 показано начало потока работ. Операция Receive названа `InitiateTrade` и является составной операцией, содержащей три других, из которых две погружены в конструкцию `IfThenElse`. Операция `CheckFraud` вызыва-

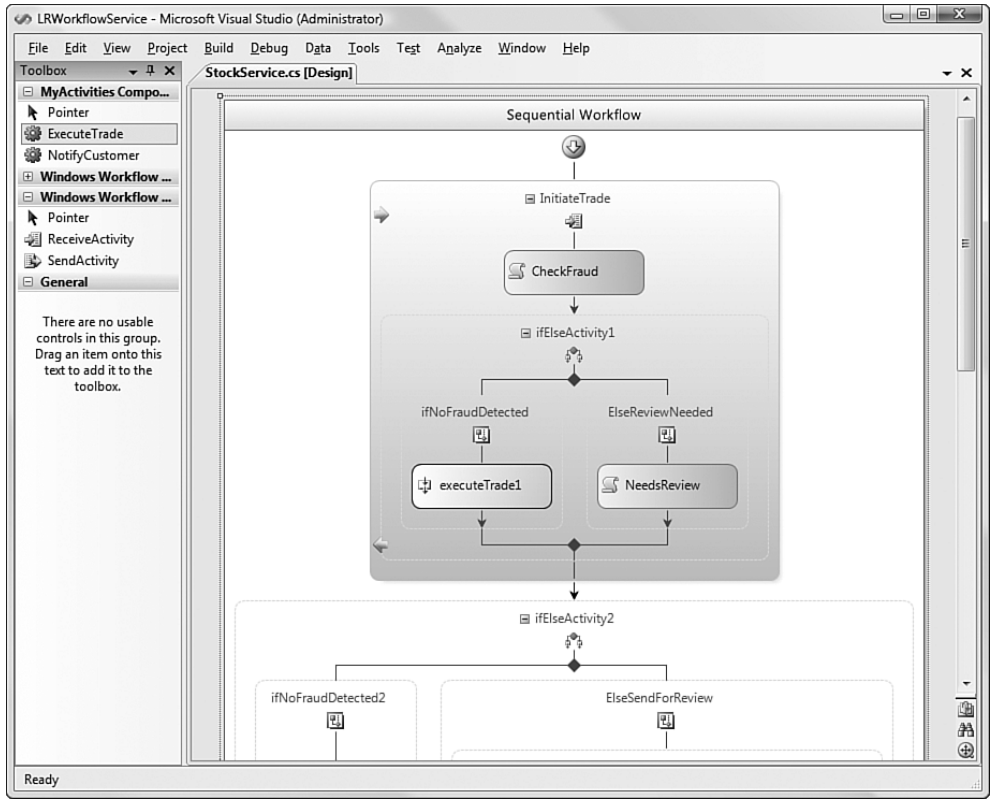

Рис. 11.10. Составная операция Receive в протяженном потоке работ

ет подпрограмму, которая оценивает подозрительность сделки. Если все выглядит нормально, то вызывается операция `ExecuteTrade`, которая выполняет запрос и возвращает структуру `RequestTradeStatus`, содержащую номер подтверждения и `status = "executed"`. Если сделка подозрительная, она помещается в очередь работ (внешнюю по отношению к данному потоку работ), а в возвращаемой структуре будет номер подтверждения и `status = "review"`. Все три операции выполняются синхронно (и, вероятно, быстро) внутри операции Receive, поэтому, по какой бы ветке ни пошло исполнение, ответ будет отправлен с незначительной задержкой.

Конфигурационный файл для этого потока работ показан в листинге 11.6. Как и для всех потоков работ, наделенных возможностями службы, в канале присутствует элемент `ContextBinding`; в данном случае мы воспользовались привязкой `wsHttpContextBinding`. Когда служба получает сообщение `TradeRequest`, заказное поведение проверяет, есть ли в нем контекст. Если нет, то `WorkflowRuntime` создает новый экземпляр потока работ. Когда операция Receive синхронно возвращает клиенту сообщение `TradeRequestStatus`, заказное поведение проштамповывает его контекстом, указывая идентификатор экземпляра данного потока работ.

Если запрос о сделке не выполняется немедленно, то поток работ приостанавливается в ожидании результатов ручного анализа. На рис. 11.11 показана оставшаяся часть потока. Операция Listen состоит из двух ветвей: Delay и еще одна операция Receive. Если операция Receive не будет вызвана до момента, указанного в операции Delay, то поток работ возобновляется без входных данных. Как и исходная операция `InitiateTrade`, операция `Approval` также является составной, но в нее погружена только одна программная операция. `ReviewReceived` анализирует сообщение `ApprovalRequest`, выясняя, одобрена ли сделка. Если да, то поднимается флаг, говорящий о том, что ее следует выполнить. Независимо от того, выполнена ли сделка в `ExecuteTrade2`, операция `NotifyCustomer` посылает клиенту, который инициировал сделку, сообщение по электронной почте.

В потоке работ, изображенном на рис. 11.10 и 11.11, есть две операции Receive, помеченные `InitiateTrade` и `Approval`. Свойство `CanCreateInstance` операции `InitiateTrade` установлено в `true`, следовательно, эту операцию можно вызывать, не передавая идентификатор экземпляра в контексте. Когда WF получает это сообщение, она создает новый экземпляр WF-программы. Свойство `CanCreateInstance` операции `Approval` сбрасывается в `false`, то есть ее нельзя вызывать, не поместив в контекст идентификатор экземпляра. Если клиент попытается это сделать, то получит в ответ отказ SOAP.

В листинге 11.8 приведена реализация описанного потока работ. Программная операция `codeCheckFraud_ExecuteCode` используется для вызова внутренней процедуры контроля и установки значения, возвращаемого Receive. Еще одна программная операция `codeNeedsReview_ExecuteCode_1` служит для вызова внутренней процедуры сохранения контекста. Третья операция `ReviewReceived_ExecuteCode` инспектирует отправленное сообщение о результатах ручного анализа и соответственно выставляет признак проверки. Внутренняя процедура

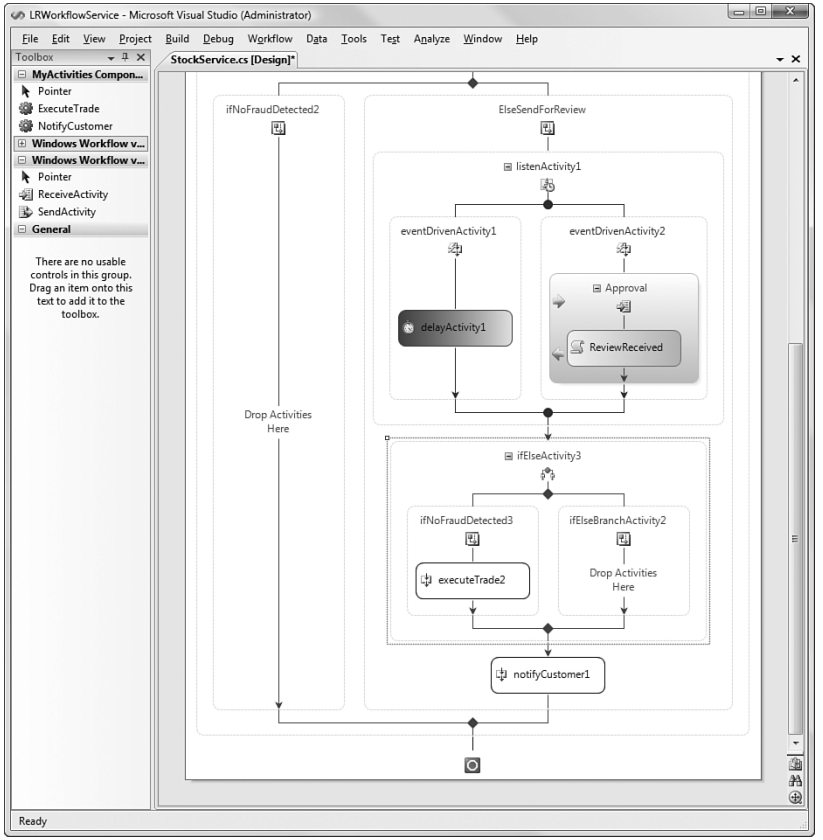

Рис. 11.11. Завершение протяженного потока работ

`saveContext` в этом примере совсем проста и просто записывает идентификатор экземпляра в файл. В промышленной системе она могла бы сохранять пару ключ/значение в базе данных или отправлять ее Web-службе, доступной как клиенту, так и службе.

Листинг 11.8. Реализация протяженного потока работ

```
public sealed partial class StockService: SequentialWorkflowActivity
{
   public StockService()
   {
      InitializeComponent();
      }

      public bool bValidation;
      public TradeRequestStatus receiveActivity1__ReturnValue_1;
      public ApprovalRequest receiveActivity2_approvalRequest1;
```

```
    public TradeRequest receiveActivity1_tradeRequest1;

    private void codeCheckFraud_ExecuteCode(object sender, EventArgs e)
    {
       wfHelper h = new wfHelper();
       bValidation = h.Evalute(receiveActivity1_tradeRequest1);
       receiveActivity1__ReturnValue_1 = new TradeRequestStatus();
       receiveActivity1__ReturnValue_1.confirmationNumber = "123";
       if (bValidation)
           receiveActivity1__ReturnValue_1.status = "OK";
       else
           receiveActivity1__ReturnValue_1.status = "Review";
    }

    private void codeNeedsReview_ExecuteCode_1(object sender, EventArgs e)
    {
       wfHelper h = new wfHelper();
       h.SaveContext
                (receiveActivity1__ReturnValue_1.confirmationNumber,
                 this.WorkflowInstanceId.ToString());
    }

    private void ReviewReceived_ExecuteCode(object sender, EventArgs e)
    {
       wfHelper h = new wfHelper();
       bValidation = h.Evalute(receiveActivity2_approvalRequest1);
    }
}

public class wfHelper
{
   public const string INSTANCEFILENAME = "c:\temp\instanceID.txt";

   public bool Evaluate(TradeRequest tradeRequest)
   {
      if (tradeRequest.account == "000") return (false);
      else return (true);
   }

   public bool Evaluate(ApprovalRequest approvalRequest)
   {
      if (approvalRequest.approval == "OK") return (true);
      else return (false);
   }

   // Работает только в отладочном или демонстрационном приложении, когда
   // клиент и служба имеют доступ к папке, в которой хранится ключ и
   // идентификатор экземпляра. В реальности это должна быть Web-служба,
   // к которой могут обратиться клиент и служба.
   public void SaveContext(string key, string instanceId)
   {
      if (File.Exists(INSTANCEFILENAME))
            File.Delete(INSTANCEFILENAME);
      string txt = string.Format("{0},{1}", key, instanceId);
      File.WriteAllText(INSTANCEFILENAME, txt);
   }
}
```


Обработка контекста

В протяженном потоке работ может быть несколько операций Receive, как показано на рис. 11.10 и 11.11. После того как первая операция Receive вызвана и создан экземпляр потока работ, все последующие обращения к этому экземпляру должны присоединять к каналу контекст, иначе не удастся выполнить корреляцию. Это означает, что отслеживать контекст экземпляров потока работ должен клиент, желающий с ними взаимодействовать. Чтобы облегчить клиенту решение этой задачи, в WF используется элемент привязки `ContextBinding`.

По умолчанию клиент имеет доступ к контексту в канале. Интерес для него представляет свойство `InstanceId`, которое однозначно идентифицирует поток работ. Клиент может получить `InstanceId` из контекста, сохранить его в памяти или в локальном файле и присоединить к каналу при последующих обращениях. Этот способ работает, коль скоро начальный и все последующие вызовы делает один и тот же клиент, которые не перезагружается между вызовами. Такая последовательность показана на рис. 11.12.

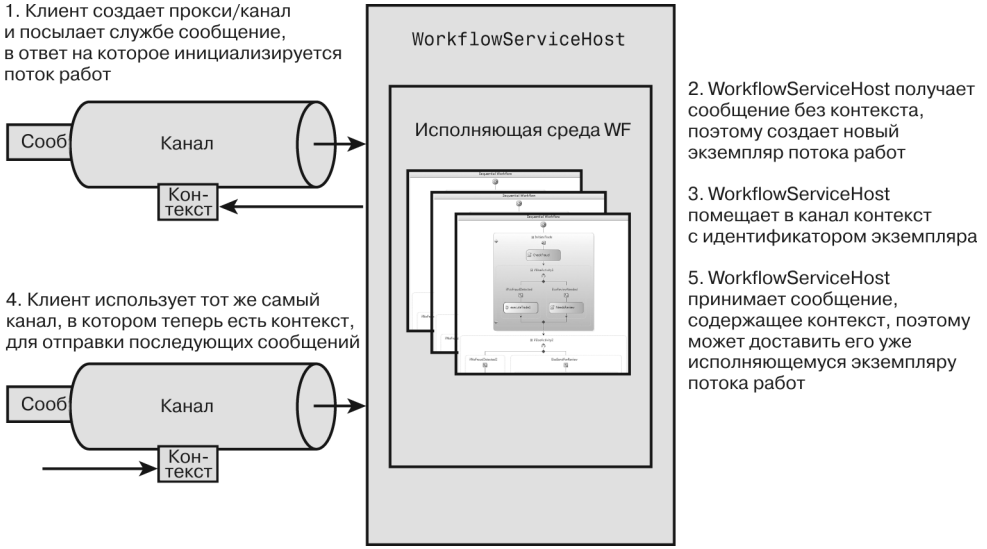

Рис. 11.12. Единственный клиент, сохраняющий контекст между вызовами к экземпляру потока работ

Однако часто бывает так, что к одному экземпляру потока работ обращаются несколько клиентов. Это могут быть различные Web-сайты или разные люди. Как бы то ни было, клиент, выполняющий любой вызов, кроме первого, должен получить контекст от первого клиента.

В случае, когда к одному экземпляру потока работ обращаются несколько клиентов, контекст, или `InstanceId`, следует хранить отдельно от клиента и эк-

экземпляра. Кроме того, с `InstanceId` необходимо ассоциировать понятное имя, чтобы клиентам не приходилось работать с внутренними сгенерированными артефактами. Как правило, это не составляет проблемы, поскольку в бизнес-транзакции уже присутствует какой-то номер подтверждения или уникальный идентификатор транзакции.

На рис. 11.13 показано, как можно организовать доступ нескольких клиентов к одному экземпляру потока работ. Здесь операция Receive явно сохраняет идентификатор экземпляра в таком месте, откуда клиент позже может его достать. Вместе с идентификатором хранится понятное клиенту имя, например, номер подтверждения, по которому клиент сможет его отыскать. Этот номер, вероятно, был получен от начальной операции Receive или включен в URL.

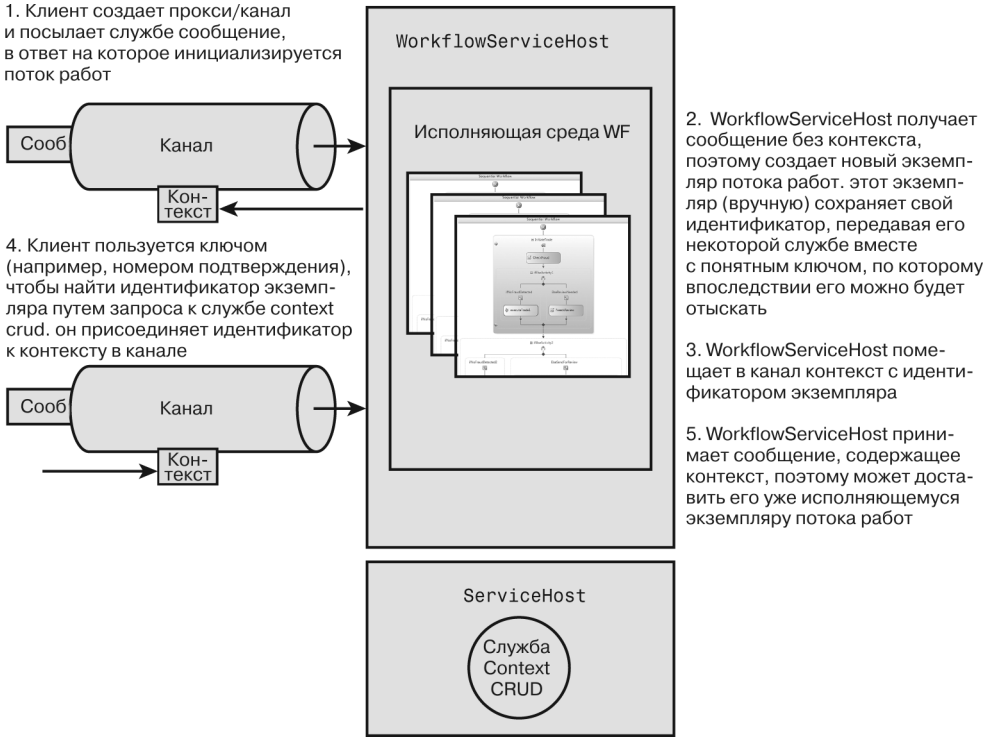

Рис. 11.13. Несколько клиентов, сохраняющих контекст при вызовах одного и того же экземпляра потока работ

Независимо от того, как клиент получает `InstanceId`, он должен создавать контекст и помещать его в канал для всех последующих вызовов. В листинге 11.9 приведен код клиента, который это делает. Он вызывает метод `GetFromSomewhere`, чтобы получить идентификатор экземпляра, а затем помещает его в контекст канала и передает прокси-классу.

Листинг 11.9. Клиент присоединяет идентификатор экземпляра потока работ к каналу

```
instanceID = GetFromSomewhere(myConfirmationNumber);
IContextManager cm =
          proxy.InnerChannel.GetProperty<IContextManager>();
IDictionary<string, string> context = new Dictionary<string,string>();
context.Add("InstanceId",instanceID);
cm.SetContext(context);
```

Сохранение состояния потока работ на сервере

Состояние протяженного потока работ сохраняется в классе `WorkflowRuntime`. Внешние события активируют операции в потоке работ, а `WorkflowRuntime` эти операции выполняет.

Когда поток работ простаивает в ожидании события, `WorkflowRuntime` может записать его состояние во внешнее хранилище, чтобы не расходовать зря системные ресурсы, например, память и процессорное время. Важнее, однако, тот факт, что в процессе останова исполняющей среды `WorkflowRuntime`, состояние всех активных потоков работ обязательно должно быть записано во внешнее хранилище, иначе экземпляры потоков работ не смогут возобновить выполнение, когда исполняющая среда восстановится.

Для решения этой задачи исполняющая среда WF пользуется службой сохранения, обращаясь к ней, когда потоки работ простаивают, и в процессе останова. Провайдер сохранения регистрируется в исполняющей среде еще до запуска каких-либо потоков работ и в дальнейшем используется для сериализации экземпляров и записи их во внешнее хранилище. Ничего относящегося к WCF в службе сохранения нет, но мы упомянули ее, чтобы был понятен следующий пример.

Службу сохранения можно зарегистрировать программно или в конфигурационном файле. Для авторазмещаемой WCF-службы с потоком работ владелец может зарегистрировать службу сохранения в своем коде. В листинге 11.10 показано, как зарегистрировать службу сохранения в базе данных SQL.

Листинг 11.10. Программная регистрация службы сохранения

```
WorkflowServiceHost serviceHost =
      new WorkflowServiceHost(typeof(StockService));

WorkflowPersistenceService persistenceService =
      new SqlWorkflowPersistenceService(
      "Initial Catalog=WFPersistence;Data Source=localhost;
                     Integrated Security=SSPI;",
      false,
      new TimeSpan(1, 0, 0),
      new TimeSpan(0, 0, 5));

WorkflowRuntime runtime =
   serviceHost.Description.Behaviors.Find
                        <WorkflowRuntimeBehavior>().WorkflowRuntime;
```

```
runtime.AddService(persistenceService);

serviceHost.Open();
```

Если доступа к объекту `WorkflowServiceHost` у вас нет, то службу сохранения необходимо зарегистрировать в конфигурационном файле. Делается это с помощью поведения службы. В листинге 11.11 приведен конфигурационный файл службы. Он похож на файл из листинга 11.6, но в секцию `<behaviors>` мы добавили описание службы сохранения.

Листинг 11.11. Регистрация службы сохранения в файле web.config

```
<behaviors>
   <serviceBehaviors>
      <behavior name="Durable.Workflow1Behavior">
         <serviceMetadata httpGetEnabled="true" />
         <serviceDebug includeExceptionDetailInFaults="true" />
         <serviceCredentials>
            <windowsAuthentication
                  allowAnonymousLogons="false"
                  includeWindowsGroups="true" />
         </serviceCredentials>
         <workflowRuntime name="WorkflowServiceHostRuntime">
            <services>
               <add type="System.Workflow.Runtime.Hosting.
                              SqlWorkflowPersistenceService,
                           System.Workflow.Runtime,
                           Version=3.0.00000.0, Culture=neutral,
                           PublicKeyToken=31bf3856ad364e35"
                  connectionString="Initial Catalog=WFPersistence;
                           Data Source=localhost;
                           Integrated Security=SSPI;"
                           LoadIntervalSeconds="1" UnLoadOnIdle= "true" />
            </services>
         </workflowRuntime>
      </behavior>
   </serviceBehaviors>
</behaviors>
```

Управление доступом к потокам работ, наделенным возможностями службы

Интеграция WF и WCF производится в основном за счет поведений. Поведения могут просматривать и изменять сообщения до того, как они поступят в исполняющую среду WF (`WorkflowRuntime`), или сразу после того, как покинут ее. У поведений есть полный доступ к сообщениям, включая и заголовки SOAP. В зависимости от модели безопасности, заданной в описании службы, клиент передает службе различную информацию в заголовке SOAP.

В примере из листинга 11.6 используется элемент `<windowsAuthentication>`. Он говорит WCF, что в заголовке SOAP нужно послать (в зашифрованном виде) сериализованное представление информации, необходимой для аутентификации средствами Windows. Атрибут `includeWindowsGroups=true` означает, что WCF должна включить все группы Windows, которым принадлежит текущий пользователь. Совместно эти настройки позволяют WF-программе принимать решения на основе идентификатора пользователя и его членства с группах.

В операцию Receive встроено два механизма управления доступом, специфичных для потоков работ. Во-первых, операцию Receive можно сконфигурировать так, чтобы доступ был разрешен только определенным пользователям или группам. Это делается декларативно. Во-вторых, в операции Receive может быть метод Operation Validation, который устанавливает флаг, разрешающий или запрещающий доступ. Это делается программно. В настоящем разделе мы рассмотрим оба способа.

Декларативный контроль доступа

Есть много способов проконтролировать доступ к WCF-службам. В главе 8 мы подробно рассмотрели роли ASP.NET, сертификаты, использование системы Kerberos и прочие. Помимо встроенных в WCF средств, WF предлагает дополнительные механизмы на уровне операции Receive.

Напомним, что в WF операция Receive раскрывает контракт об операциях службы. Когда в проект добавляется контракт об операциях, открывается диалоговое окно для задания параметров, свойств и разрешений. Это окно показано на рис. 11.14. Можно ввести имя домена и пользователя в нем (domain\username) в поле Name либо имя домена и группы (domain\group) в поле Role. На этапе выполнения, после того как сообщение получено службой, но до его передачи `WorkflowRuntime` поведение проверяет разрешения на основе членства. Для этого исследуется заголовок сообщения и затребованные в нем разрешения сравниваются с теми, что были заданы в диалоговом окне. Если сообщение поступило от разрешенного пользователя или группы, операция выполняется. В противном случае возвращается исключение подсистемы безопасности.

Программный контроль доступа

Любая операция Receive может определить метод для авторизации доступа. Этот метод вызывается перед обращением к операции службы и может разрешить или запретить доступ. Имя метода хранится в свойстве `OperationValidation` операции Receive и может быть сгенерировано конструктором WF. На этапе выполнения этот метод вызывается поведением WCF до обращения к операции службы. Ему передается объект, содержащий все разрешения, затребованные клиентом. Тем самым авторизация на основе требований клиента реализуется сравнительно легко.

Добавим еще одно требование в наш пример с торговлей акциями: лицо, инициировавшее сделку, не может совпадать с лицом, одобрившим ее. Для его реализации нам потребуется сделать две вещи. В первой операции Receive (рис. 11.10) метод Operation Validation находит и сохраняет имя пользователя, инициировавшего сделку. Во второй операции Receive (рис. 11.11) метод Operation Validation

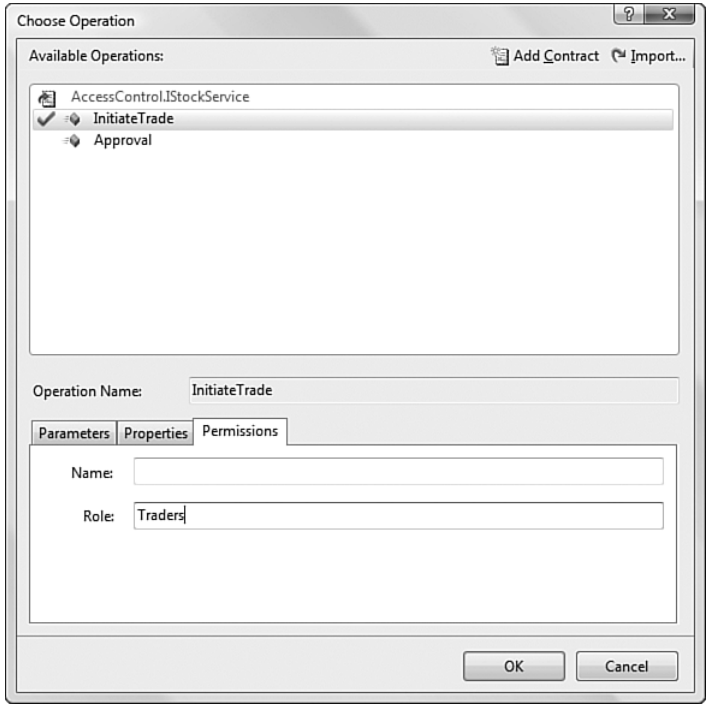

Рис. 11.14. Декларативная авторизация в операции Receive

сравнивает пользователя, рассматривавшего сделку на предмет одобрения, с тем, кто ее инициировал. Если это один и тот же пользователь или любой из них не задан, то в авторизации будет отказано.

В листинге 11.12 приведена небольшая функция для определения имени пользователя на основе объекта `ClaimsSet`, отправленного службе. Это функция работала бы надежнее с более предсказуемыми запросами, например, такими как в сертификатах, но для иллюстрации программного контроля доступа к операциям службы ее достаточно.

Листинг 11.12. Функция для поиска имени пользователя, запросившего разрешение, на основе ClaimsSet

```
private string findClaimName(OperationValidationEventArgs e)
{
   string claimName="";
   if (e.ClaimSets.Count == 1)
   {
      IEnumerable myClaims = e.ClaimSets[0].
                             FindClaims("http://schemas.xmlsoap.org/
                                    ws/2005/05/identity/claims/name",
                                    Rights.PossessProperty);
      foreach (Claim c in myClaims) while (claimName =="")
```



```
         claimName = c.Resource.ToString();
   }
   return claimName;
}
```

В листинге 11.13 демонстрируется применение функции `findClaimName`. Конструктор WF генерирует две функции: `receive1_OpValidation` и `receive2_OpValidation`. Первая сохраняет имя пользователя в переменной в области действия класса потока работ. Если она не может найти имя пользователя, то выставляет флаг **`IsValid = false`**, отвергая вызов. Вторая сравнивает текущее имя пользователя с предыдущим. Если они совпадают, то выставляется флаг `IsValid = false`, то есть вызов снова отвергается. В любом случае, увидев, что `IsValid = false`, Receive не станет вызывать операцию службы.

Листинг 11.13. Методы контроля доступа к операции службы

```
private void receive1_OpValidation(object sender,
                                   OperationValidationEventArgs e)
{
   initiatedBy = findClaimName(e);
   if (initiatedBy == "") e.IsValid = false;
}

private void receive2_OpValidation(object sender,
                                   OperationValidationEventArgs e)
{
   string reviewedBy = findClaimName(e);
   if (reviewedBy == initiatedBy)
         e.IsValid = false;
}
```

Резюме

Windows Workflow Foundation (WF) – это технология, дополняющая WCF. Если WCF определяет и реализует интерфейс со службой, то WF моделирует и реализует бизнес-логику службы. Visual Studio 2008 и .NET 3.5 обеспечивают глубокую интеграцию обеих технологий.

Для вызова служб из WF можно пользоваться операцией Send или Code. В любом случае сначала необходимо добавить ссылку на службу, воспользовавшись инструментом Add Service Reference или утилитой `svcutil.exe`.

Операция Receive используется в конструкторе WF для раскрытия потока работ в виде службы. Будучи правильно сконфигурирована, WF определяет атрибут `[ServiceContract]` для службы и атрибут `[OperationContract]` для каждой операции Receive. Можно указать на операцию Receive при обращении к прокси-классам, импортированным или включенным в проект, либо определить интерфейсы с помощью конструктора WF. Операция Receive связывает переменные WF с операцией службы. Конструктор WF позволяет выбрать или создать переменные WF, связываемые с каждым входным параметром операции службы.

Класс `WorkflowServiceHost` служит для создания экземпляра исполняющей среды WF. Этот класс используется вместо `ServiceHost` для потоков работ, наделенных возможностями службы. При авторазмещении его можно использовать непосредственно. При размещении в IIS для достижения того же результата необходимо задать фабрику класса в SVC-файле.

При раскрытии потока работ, наделенного возможностями службы, необходимо использовать одну из контекстных привязок: `basicHttpContextBinding`, `wsHttpContextBinding` или `netTcpContextBinding`. В них встроен элемент `ContextBindingElement`, позволяющий включить в канал информацию о контексте. Этот элемент можно использовать и в заказных привязках. Контекст необходим для корреляции входящих сообщений с существующими экземплярами потоков работ.

Многие WF-программы, моделирующие бизнес-процессы, работают в течение дней, недель или месяцев. За это время клиент может отключиться от сети, служба – перезагрузиться, а сеть – пропасть и снова появиться. Чтобы поддержать протяженный характер таких процессов, необходимо служба сохранения. Когда экземпляр потока работ простаивает или останавливается исполняющая среда, служба сохранения «сдувает» экземпляр, записывая его состояние во внешнее хранилище. Получив сообщение для «сдутого» экземпляра, WF обращается к службе сохранения с просьбой «надуть» его.

Глава 12. Пиринговые сети

Многие разработчики распределенных приложений рассматривают только клиент-серверные или n-ярусные модели. Но есть еще один подход, на который часто не обращают внимания, – одноранговые или пиринговые (peer-to-peer – P2P) сети. Он применяется в ряде наиболее популярных Интернет-приложений: интернет-пейджерах, играх и файлообменных сетях. В отличие от других типов, пиринговое приложение не предполагает наличия централизованной инфраструктуры, то есть между клиентом и сервером нет никаких различий. Это заметно усложняет задачу проектирования подобных приложений. Большинство разработчиков шарахаются от пиринговых приложений, опасаясь встающих на пути трудностей. Но если подойти к делу с умом, то пиринговые приложения обладают несомненными достоинствами в плане масштабируемости и надежности. В этой главе мы будем говорить о построении пиринговых приложений на платформе WCF и Windows Vista. Мы рассмотрим, что предлагает в этом отношении WCF и какие новые возможности появились в .NET Framework 3.5.

Подходы к построению распределенных приложений

Большинство современных распределенных приложений построены на базе одной из трех архитектур: клиенты-серверные, n-ярусные и пиринговые. В этом разделе мы сравним различные подходы, чтобы разобраться, зачем вообще нужны пиринговые приложения.

Клиент-серверные приложения

В прошлые годы многие распределенные приложения были клиент-серверными. В этой модели выделяются клиент и сервер, каждый со своей ролью. Клиент отправляет запрос, а сервер отвечает на него. К числу самых распространенных сегодня клиент-серверных приложений относятся Web-браузеры, в частности Internet Explorer. Пользователь посылает запрос, набирая URL в поле адреса, а сервер отвечает. URL содержит как собственно запрос, так и адрес Web-сервера, которому этот запрос предназначен. Web-сервер, например Internet Information Services (IIS), обрабатывает входящий запрос и посылает ответ клиенту. На рис. 12.1 изображена клиент-серверная модель.

Рис. 12.1. Клиент-серверная модель

N-ярусные приложения

Позже распределенные приложения из двухъярусных (клиент и сервер) превратились в n-ярусные. Наиболее распространена трехъярусная модель, в которой пользовательский интерфейс, бизнес-логика и уровень доступа к данным разнесены по разным физическим ярусам. На рис. 12.2 изображена трехъярусная модель. У нее есть немало преимуществ. Чаще всего в качестве обоснования приводят тот факт, что инкапсуляция бизнес-логики в отдельном физическом ярусе позволяет обеспечить ее безопасность. Кроме того, так построенные приложения проще масштабируются, правда, за счет дополнительного оборудования.

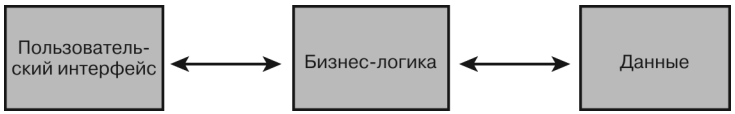

Рис. 12.2. Трехъярусная модель

Пиринговые приложения

Еще один вид распределенных приложений – пиринговые. В чисто пиринговом приложении каждый участник (*узел*) выступает по отношению к остальным участникам как в роли клиента, так и в роли сервера. Каждый может инициировать запросы и отвечать на них. Такую архитектуру часто ассоциируют с совместной работой. Примером может служить Gnutella – Интернет-приложение, обеспечивающее общий доступ к файлам. Другие примеры: интернет-пейджеры, программы презентации, открытые «доски» (whiteboard) и программы для совместного доступа к файлам. На рис. 12.3 изображена модель пирингового приложения с тремя узлами.

Сравнение подходов к построению распределенных приложений

Клиент-серверные и n-ярусные приложения обычно гораздо проще построить, чем пиринговые. У них немало достоинств, в частности: простота разработки, централизованное управление и безопасность. К недостаткам следует отнести сложность масштабирования и недостаточную надежность. Первая проблема решается путем вертикального (установка более мощного и дорогого оборудования) или горизонтального (увеличение числа серверов) масштабирования. Повысить надежность можно за счет добавления избыточного оборудования. В любом случае необходима дорогостоящая аппаратура, что существенно повышает общую стоимость решения. Пиринговые приложения – прямая противоположность. Многие достоинства пиринговых приложений одновременно являются недостатками клиент-серверных и наоборот. Например, можно добавлять новые узлы, в качестве которых выступают компьютеры для массового потребителя, и тем самым повысить масштабируемость и надежность, не тратясь на дорогостоящие серверы. Но при этом нет централизованной системы управления и обеспечения безопасности, а, значит, развертывание, сопровождение и защита пиринговых приложений оказываются сложнее. В общем, ни один подход не лучше другого, и выбор того или иного диктуется требованиями к конкретной системе. В некоторых случаях можно применить смешанный подход. Далее в этой главе мы будем говорить только о построении пиринговых приложений с помощью WCF.

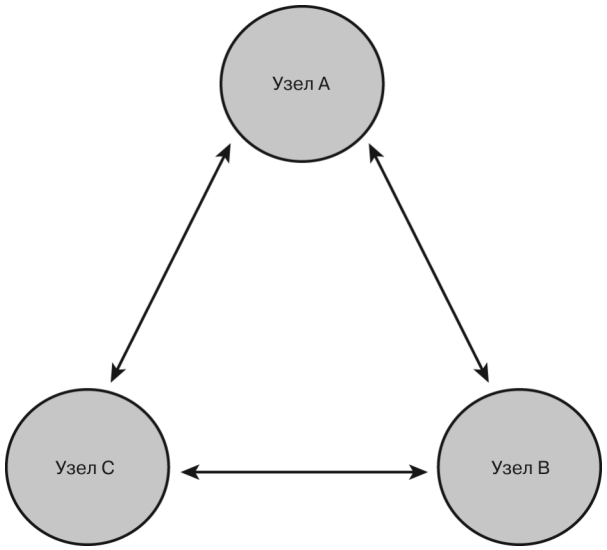

Рис. 12.3. Пиринговая модель с тремя узлами

Пиринговые приложения

В этом разделе мы увидим, какую поддержку WCF предоставляет для создания пиринговых приложений. Но сначала рассмотрим различные способы коммуникации в пиринговых сетях.

Ячеистые сети

В пиринговых приложениях взаимодействие основано на идее *ячеистой сети* (mesh network). Ячеистая сеть – это группа равноправных узлов (пиров), связан-

ных между собой. Каждый узел представляет собой один экземпляр пирингового приложения. *Полносвязной* называется ячеистая сеть, в которой каждый узел связан с каждым. Пример полносвязной сети показан на рис. 12.4.

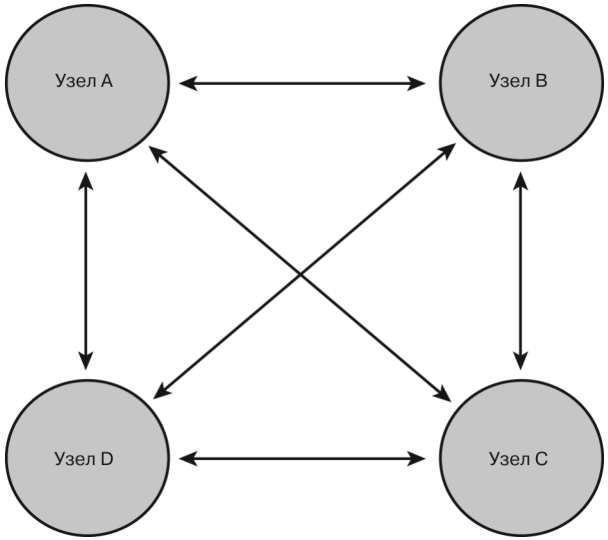

Рис. 12.4. Полносвязная ячеистая сеть

Полная связность имеет смысл только для очень небольших сетей. Невозможно построить полносвязную сеть, в которой столько узлов, сколько пользователей Интернета. Поэтому в большинстве случаев ячеистые сети *частично связны*. В частично связной сети узлы соединены только со своими соседями. На рис. 12.5 приведен пример частично связной сети. В таких сетях объем необходимых на каждом узле ресурсов меньше, поэтому масштабируемость выше. Масштабируемость ячеистой сети измеряется количеством участников. Недостаток частичной связности состоит в том, что невозможно отправить сообщение сразу всем узлам сети. Приходится передавать сообщение от соседа к соседу, пока его не получат все участники. Сообщение циркулирует в сети, пока не дойдет до каждого узла или не будет превышена заданная глубина распространения.

Разрешение имен в ячеистой сети

В пиринговых приложениях для идентификации узлов ячеистой сети применяются имена ячеек. Имя ячейки – это логическое сетевое имя, используемое приложением для адресации узлов. В какой-то момент имени ячейки должен быть сопоставлен набор сетевых адресов, с которыми можно установить соединение. Эта процедура называется разрешением имени. Обычно для этого необходимо соединиться с другими участниками и обменяться с ними информацией об известных узлах. Существует много способов, позволяющих пиринговому приложению обнаружить другие узлы сети, как то: протоколы с групповым вещанием (например, UDP) или хорошо известные серверы, которые называются стартовыми (bootstrap servers). В WCF используется протокол Peer Network Resolution Protocol (PNRP), реализованный в Windows XP SP2 и Windows Vista. В нем применяется сочетание различных приемов обнаружения, пригодных в разных сетях. Дополнительную информацию см. в разделе «Обнаружение участников с помощью протокола PNRP».

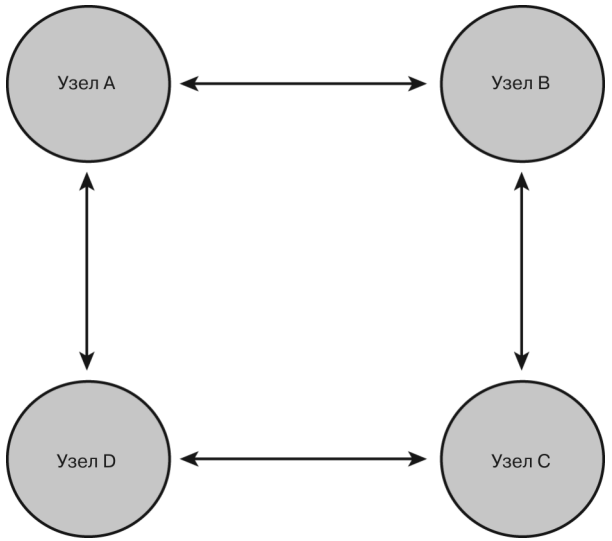

Рис. 12.5. Частично связная ячеистая сеть

Массовое и направленное вещание

Для коммуникаций в ячеистой сети применяется массовое (message flooding) либо направленное (directional messaging) вещание. При массовом вещании сообщения рассылаются всем узлам сети. Распространение сообщения по сети достигается за счет того, что каждый узел переправляет его своим соседям. При направленном вещании делается попытка послать сообщение конкретному узлу. Узел-отправитель посылает сообщение одному из своих соседей, тот – своему соседу, и так, пока сообщение не достигнет адресата. В любом случае применяются разнообразные приемы, чтобы ограничить общее число сообщений в сети.

WCF поддерживает пиринговые приложения с массовым вещанием. Готового решения для направленного вещания нет, но его можно надстроить над имеющимися средствами с помощью механизмов расширения WCF.

Создание пиринговых приложений

WCF поддерживает создание пиринговых приложений с помощью привязки `netPeerTcpBinding`, которая обеспечивает многостороннюю коммуникацию по пиринговому транспортному протоколу. Кроме того, WCF определяет средства

обнаружения участников в ячеистой сети. По умолчанию для этого используется протокол PNRP. Эта технология встроена в операционные системы Windows, начиная с Windows XP SP2. Подробнее протокол PNRP будет рассмотрен ниже в этой главе.

Привязка netPeerTcpBinding

Привязка `netPeerTcpBinding` и соответствующий ей элемент привязки `PeerTransportBindingElement` обеспечивает поддержку пиринговых коммуникаций в WCF. В ней используется транспортный протокол TCP и двоичный кодировщик сообщений.

Привязке `netPeerTcpBinding` соответствует следующая схема адресации:

```
net.peer://{meshname}[:port]/{service location}
```

По умолчанию подразумевается порт 0, то есть транспортный протокол выберет номер порта случайным образом, но можно указать и конкретный порт.

В таблице 12.1 перечислены свойства привязки `netPeerTcpBinding` и их значения по умолчанию.

Таблица 12.1. Свойства привязки netPeerTcpBinding

Имя атрибута	Описание	Значение по умолчанию
`closeTimeout`	Максимальное время ожидания закрытия соединения	00:01:00
`listenIPAddress`	Порт прослушивателя пирингового транспортного протокола. 0 означает, что номер порта выбирается случайным образом	0
`maxBufferSize`	Максимальный объем памяти, отведенной для хранения (в байтах)	65 536
`maxConnections`	Максимальное число входящих или исходящих соединений. Входящие и исходящие соединения считаются порознь	10
`maxReceivedMessageSize`	Максимальный размер одного входящего сообщения	65 536
`name`	Имя привязки	Нет
`openTimeout`	Максимальное время ожидания завершения операции открытия соединения	00:01:00
`readerQuotas`	Определяет сложность подлежащих обработке сообщений (например, размер)	Нет
`receiveTimeout`	Максимальное время ожидания завершения операции приема	00:01:00
`security`	Параметры безопасности привязки	Нет

Таблица 12.1. Свойства привязки netPeerTcpBinding (окончание)

Имя атрибута	Описание	Значение по умолчанию
`sendTimeout`	Максимальное время ожидания завершения операции отправки	00:01:00
`resolver`	Распознаватель, служащий для регистрации и разрешения имен других участников в ячеистой сети	Нет

В листинге 12.1 показана минимальная конфигурация, необходимая для раскрытия службы с помощью привязки `netPeerTcpBinding`.

Листинг 12.1. Конфигурация службы для привязки netPeerTcpBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <services>
         <service name="EssentialWCF.HelloWorld">
            <endpoint binding="netPeerTcpBinding"
                      contract="EssentialWCF.IHelloWorld"
                      address="net.peer://MyMeshName/HelloWorld/" />
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

В листинге 12.2 показана минимальная конфигурация, необходимая для потребления службы с помощью привязки `netPeerTcpBinding`.

Листинг 12.2. Конфигурация клиента для привязки netPeerTcpBinding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <client>
         <endpoint binding="netPeerTcpBinding"
                   contract="EssentialWCF.IHelloWorld"
                   address="net.peer://MyMeshName/HelloWorld/" />
      </client>
   </system.serviceModel>
</configuration>
```

Обнаружение участников с помощью протокола PNRP

PNRP – это распределенный протокол разрешения имен, предназначенный для публикации и обнаружения информации о пиринговых ресурсах. В WCF этот протокол применяется для обнаружения узлов в ячеистой сети, что позволяет участникам находить соседей, с которыми они могут общаться. Уже сравнительно

давно этот протокол можно было загрузить отдельно для Windows XP, а в Windows XP SP2 он был включен штатно. В Windows Vista версия PNRP v2.0 устанавливается автоматически, если вы включаете поддержку IPv6.

Как получить PNRP v2.0 для Windows XP SP2. Версия PNRP v2.0 поставляется в комплекте с Windows Vista. Она *не* работает с предыдущими версиями PNRP-клиентов, например, Windows XP SP2. Microsoft предлагает бесплатно загрузить последнюю версию со своего сайта (http://support.microsoft.com/kb/920342). В результате PNRP-клиенты, работающие на платформе Windows XP SP2, смогут общаться с клиентами в Windows Vista.

Протокол PNRP построен поверх IPv6, поэтому поддержка IPv6 должна быть установлена. Но предоставляемыми WCF средствами организации пиринговых сетей вы можете воспользоваться даже без поддержки IPv6, если напишете нестандартный распознаватель, работающий по специальному протоколу вместо PNRP. О том, как подойти к решению этой задачи, рассказано в разделе «Реализация нестандартного распознавателя» ниже.

Принцип работы PNRP основан на публикации информации о пиринговом ресурсе, так чтобы другие участники сети могли его обнаружить. Обычно в состав этой информации входит список клиентов и IP-адресов их оконечных точек, привязанных к именам в ячеистой сети. PNRP можно использовать для хранения различной информации; однако мы сосредоточимся только на пиринговом транспортном канале и на том, как он задействует средства PNRP. С помощью PNRP транспортный канал публикует информацию о том, как общаться с другими пиринговыми приложениями в той же ячеистой сети, а именно: имя ячейки и оконечные точки служб, ассоциированных с каждым узлом. Когда очередное пиринговое приложение запускается, оно по протоколу PNRP находит другие работающие сейчас в сети приложения.

Процедура начальной загрузки PNRP

Для начальной загрузки PNRP в ячеистой сети (ее называют еще PNRP-облаком) выполняется несколько шагов, в ходе которых участники регистрируются и в дальнейшем могут обнаружить друг друга. Кроме того, многошаговая процедура позволяет PNRP масштабироваться как в изолированных сетях (например, в корпоративной локальной сети), так и в Интернете, поскольку делается все возможное для минимизации сетевого трафика, необходимого для присоединения к сети.

1. **Проверить ранее заполненный кэш**. PNRP поддерживает локальный кэш оконечных точек ресурсов для каждого узла. Если клиент ранее регистрировался в какой-то ячейке, то PNRP попытается использовать хранящуюся в кэше запись, чтобы повторно присоединиться к той же ячейке.
2. **Простой протокол обнаружения служб (Simple Service Discovery Protocol – SSDP)**. SSDP – это часть спецификации Universal Plug-n-Play (UPnP), он позволяет совместимым с ней устройствам обнаруживать друг друга в локальной сети. Та же техника применима и для взаимного обнаружения пиринговых узлов в локальной сети.

3. **Затравочные узлы PNRP**. PNRP-клиентов можно сконфигурировать так, чтобы они искали некий узел, который можно использовать для вхождения в PNRP-облако. Такие узлы часто называют *затравочными* (seed node). Microsoft предоставляет публичный затравочный узел в Интернете по адресу pnrpv2.ipv6.microsoft.com. Пользователи при желании могут организовать собственные затравочные узлы в своей сети.

Имена компьютеров в Windows Internet

В Windows Vista можно ассоциировать с компьютером имя, которое будет опубликовано по протоколу PNRP. Это позволяет обойтись без добавления имени в базу данных DNS-сервера. Такое имя называется Windows Internet Computer Name (WICN). Иногда еще WICN-имена называют «именами PNRP-пиров». Есть два типа WICN-имен: защищенные и незащищенные. Незащищенное имя обычно предназначено для запоминания человеком, например: richshomecomputer.pnrp.net. На рис. 12.6 показано применение команды `netsh` для опроса WICN-имени компьютера. Она имеет вид `netsh p2p pnrp peer show machine name`.

Рис. 12.6. Получение WICN-имени

Защищенные имена зашифрованы и для доказательства права владения необходим закрытый ключ. Такие имена генерируются с помощью свертки открытого ключа. Чтобы опубликовать защищенное имя, вы должны располагать закрытым ключом, парным к открытому ключу, который хранится внутри имени. Вот пример защищенного имени:

```
p9962c5876ab48521cab41350013457109c1b0219
```

Интересно, что WICN-имена можно запрашивать с помощью DNS API. Ниже показано то же самое имя в формате, пригодном для запроса к DNS:

```
p.p9962c5876ab48521cab41350013457109c1b0219.pnrp.net
```

Класс PnrpPeerResolver

В привязку `netPeerTcpBinding` встроена возможность задать распознаватель, с помощью которого будут обнаруживаться другие участники сети. По умолчанию, если не задано ничего другого, подразумевается класс `PnrpPeerResolver`, который реализует абстрактный класс `PeerResolver`. Оба эти класса находятся в пространстве имен `System.ServiceModel.Channels`.

Аутентификация в ячеистой сети

Ячеистые сети можно защитить, задав пароль или сертификат X.509. Чтобы приложение могло стать участником сети, оно должно предъявить пароль. Пароли позволяют приложениям регистрироваться в сети и обнаруживать других участников. Но пароль ничего не говорит о том, является ли участник аутентифицированным пользователем. Пароль следует задавать как на сервере, так и на клиенте. В листинге 12.3 показано, как задать пароль в объекте `ServiceHost`:

Листинг 12.3. Задание пароля доступа к ячеистой сети в объекте ServiceHost

```
host = new ServiceHost(typeof(PeerChatService), baseAddresses);
host.Credentials.Peer.MeshPassword = meshPassword.ToString();
host.Open();
```

Соответствующий код клиента показан в листинге 12.4. Здесь мы применили подход на основе класса `ChannelFactory`, описанный в главе «Каналы».

Листинг 12.4. Код для задания доступа к ячеистой сети на стороне клиента

```
using (ChannelFactory<IPeerChat> cf =
               new ChannelFactory<IPeerChat>(binding, ep))
{
   cf.Credentials.Peer.MeshPassword = meshPassword.ToString();

   IPeerChat chat = cf.CreateChannel();
   chat.Send(from, message);
}
```

Регистрация имен по протоколу PNRP

WCF может использовать протокол PNRP для обнаружения других участников в ячеистой сети. Пиринговый канал в WCF абстрагирует применение PNRP, поэтому приложению не нужно работать с этим протоколом напрямую. Однако у некоторых пиринговых приложений может возникнуть потребность публиковать и разрешать имена самостоятельно в обход канала WCF. К сожалению, до выхода .NET Framework 3.5 не было способа зарегистрировать PNRP-имя из управляемого кода. В .NET Framework 3.5 появилось новое пространство имен `System.Net.Peer`, устраняющее эту проблему.

Пространство имен System.Net.Peer

Мы уже говорили, что протокол PNRP позволяет публиковать и разрешать имена участников пиринговой сети. Чтобы опубликовать имя, нужно сначала создать экземпляр класса `PeerName`. В этом объекте задается идентификатор (имя участника) и признак защищенности имени. Далее мы пользуемся классом `PeerNameRegistration`, чтобы зарегистрировать имя. Для этого следует задать свойства `PeerName` и `Port`, а затем вызвать метод `Start`. Метод `Stop` отменяет регистрацию имени. В листинге 12.5 приведен пример регистрации имени участника пиринговой сети.

Имена участников – собственность приложений. Именем участника владеет зарегистрировавшее его приложение. Если по какой-то причине приложение завершается, то регистрация имени отменяется. Следовательно, для того чтобы участника можно было обнаружить, приложение должно работать.

Листинг 12.5. Публикация имени участника

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.PeerToPeer;
using System.Text;

namespace PublishName
{
   class Program
   {
      static void Main(string[] args)
      {
         PeerName peerName =
                        new PeerName("PeerChat", PeerNameType.Unsecured);
         PeerNameRegistration pnReg = new PeerNameRegistration();
         pnReg.PeerName = peerName;
         pnReg.Port = 8080;
         pnReg.Comment = "My registration.";
         pnReg.Data = Encoding.UTF8.GetBytes("Данные, сопровождающие
➥регистрацию.");

         pnReg.Start();

         Console.WriteLine("Для завершения нажмите [Enter].");
         Console.ReadLine();
         pnReg.Stop();
      }
   }
}
```

В листинге 12.6 показано, как разрешить имя участника, зарегистрированного программой в листинге 12.5. Мы пользуемся классом `PeerNameResolver`, чтобы получить набор объектов типа `PeerNameRecord`. Затем мы обходим этот набор и выводим информацию, хранящуюся в каждой записи.

Листинг 12.6. Разрешение имени участника

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.PeerToPeer;
using System.Text;

namespace ResolveName
{
   class Program
   {
      static void Main(string[] args)
      {
         PeerNameResolver resolver = new PeerNameResolver();
         PeerName peerName = new PeerName("0.PeerChat");

         PeerNameRecordCollection results = resolver.Resolve(peerName);

         PeerNameRecord record;

         for (int i=0; i<results.Count; i++)
         {
            record = results[i];

            Console.WriteLine("Запись #{0}", i);
            if (record.Comment != null)
                  Console.WriteLine(record.Comment);

            Console.Write("Данные: ");
            if (record.Data != null)
                  Console.WriteLine(Encoding.ASCII.GetString(record.Data));
            else
                  Console.WriteLine();

            Console.WriteLine("Оконечные точки:");

            foreach (IPEndPoint endpoint in record.EndPointCollection)
               Console.WriteLine("Оконечная точка:{0}", endpoint);

            Console.WriteLine();
         }

         Console.WriteLine("Для завершения нажмите [Enter].");
         Console.ReadLine();
      }
   }
}
```

Реализация нестандартного распознавателя

Пиринговый транспортный канал оставляет разработчикам возможность реализовать собственный алгоритм обнаружения участников, указав нестандартный распознаватель. Есть много причин пользоваться нестандартным распознавателем вместо встроенного в протокол PNRP. PNRP зависит от протокола IPv6, а для того чтобы клиенты на платформах Windows XP SP2 и Vista могли работать вместе, необходимо загружать дополнительное программное обеспечение. Нестандартный же распознаватель может опираться на уже существующую инфраструктуру IPv4, что упрощает развертывание. В Windows SDK имеется пример реализации на основе WCF-службы. Мы приведем похожий пример, в котором тоже используется служба, но с хранилищем в базе данных SQL Server 2005. Во многих приложениях это решение можно использовать для организации совместной работы нескольких компьютеров в сети.

Для реализации нового распознавателя вы должны создать класс, наследующий абстрактному базовому классу `PeerResolver`. В этом классе есть ряд методов, позволяющих регистрировать, обновлять и отменять регистрацию клиента в ячеистой сети. Кроме того, имеется метод для обнаружения других участников сети. В листинге 12.7 приведен код класс `SqlPeerResolver` и ассоциированных с ним классов, относящихся к конфигурированию.

Листинг 12.7. Класс SqlPeerResolver

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Configuration;
using System.ServiceModel;
using System.Runtime.Serialization;

namespace EssentialWCF.PeerNetworking
{
   public class SqlPeerResolver : PeerResolver
   {
      private static object dalLock = new object();
      private static SqlPeerResolverDatabase dal;

      private static SqlPeerResolverDatabase DAL
      {
         get
         {
            if (dal == null)
            {
               lock (dalLock)
               {
                  if (dal == null)
                        dal == new SqlPeerResolverDatabase();
               }
            }

            return dal;
         }
      }

      public override bool CanShareReferrals
      {
```

```
        get { return true; }
    }

    public override object Register(string meshId,
        PeerNodeAddress nodeAddress, TimeSpan timeout)
    {
        MaskScopeId(nodeAddress.IPAddresses);
        int registrationId = DAL.Register(meshId, nodeAddress);
        return registrationId;
    }

    public override void Unregister(object registrationId,
        TimeSpan timeout)
    {
        DAL.Unregister((int)registrationId);
    }

    public override void Update(object registrationId,
        PeerNodeAddress updatedNodeAddress, TimeSpan timeout)
    {
        MaskScopeId(updatedNodeAddress.IPAddresses);
        DAL.Update((int)registrationId, updatedNodeAddress);
    }

    public override ReadOnlyCollection<PeerNodeAddress>
        Resolve(string meshId, int maxAddresses, TimeSpan timeout)
    {
        PeerNodeAddress[] addresses = null;

        addresses = DAL.Resolve(meshId, maxAddresses);

        if (addresses == null)
              addresses = new PeerNodeAddress[0];

        return new ReadOnlyCollection<PeerNodeAddress>(addresses);
    }

    void MaskScopeId(ReadOnlyCollection<IPAddress> ipAddresses)
    {
        foreach (IPAddress address in ipAddresses)
        {
           if (address.AddressFamily == AddressFamily.InterNetworkV6)
               address.ScopeId = 0;
        }
    }
}

public class SqlPeerResolverBindingElement : PeerResolverBindingElement
{
    PeerReferralPolicy peerReferralPolicy = PeerReferralPolicy.Share;
    static SqlPeerResolver resolverClient = new SqlPeerResolver();

    public SqlPeerResolverBindingElement() { }
    protected SqlPeerResolverBindingElement(SqlPeerResolverBindingElement
➥: base(other) { }
```



```
    public override PeerReferralPolicy ReferralPolicy
    {
        get { return peerReferralPolicy; }
        set { peerReferralPolicy = value; }
    }

    public override BindingElement Clone()
    {
        return new SqlPeerResolverBindingElement(this);
    }

    public override IChannelFactory<TChannel>
➥BuildChannelFactory<TChannel>(BindingContext context)
    {
        context.BindingParameters.Add(this);
        return context.BuildInnerChannelFactory<TChannel>();
    }

    public override bool
➥CanBuildChannelFactory<TChannel>(BindingContext context)
    {
        context.BindingParameters.Add(this);
        return context.CanBuildInnerChannelFactory<TChannel>();
    }

    public override IChannelListener<TChannel>
➥BuildChannelListener<TChannel>(BindingContext context)
    {
        context.BindingParameters.Add(this);
        return context.BuildInnerChannelListener<TChannel>();
    }

    public override bool
➥CanBuildChannelListener<TChannel>(BindingContext context)
    {
        context.BindingParameters.Add(this);
        return context.CanBuildInnerChannelListener<TChannel>();
    }

    public override PeerResolver CreatePeerResolver()
    {
        return resolverClient;
    }

    public override T GetProperty<T>(BindingContext context)
    {
        return context.GetInnerProperty<T>();
    }
}

public class SqlPeerResolverConfigurationBindingElement :
                BindingElementExtensionElement
{
    public override Type BindingElementType
    {
```

```
        get { return typeof(SqlPeerResolverBindingElement); }
      }

      protected override BindingElement CreateBindingElement()
      {
         return new SqlPeerResolverBindingElement();
      }
   }
}
```

Ограничение количества передач сообщения

В пиринговых сетях, где применяется массовое вещание, обычно есть механизм ограничения расстояния, которое сообщение может проходить по сети. Под расстоянием понимают количество передач сообщения от одного узла другому. Читатели, знакомые с программированием на уровне сокетов, увидят здесь аналогию с параметром «время жизни» (Time-to-Live – TTL) в протоколе TCP, определяющего максимальное число маршрутизаторов, через которые может пройти пакет, прежде чем будет отброшен. На рис. 12.7 эта идея проиллюстрирована на примере нескольких взаимосвязанных узлов. Сообщение, отправленное узлом A, достигнет узла B после трех передач. Количество передач может быть существенно в сетях масштаба Интернета. Нам необходимо как-то ограничить его в таких ситуациях.

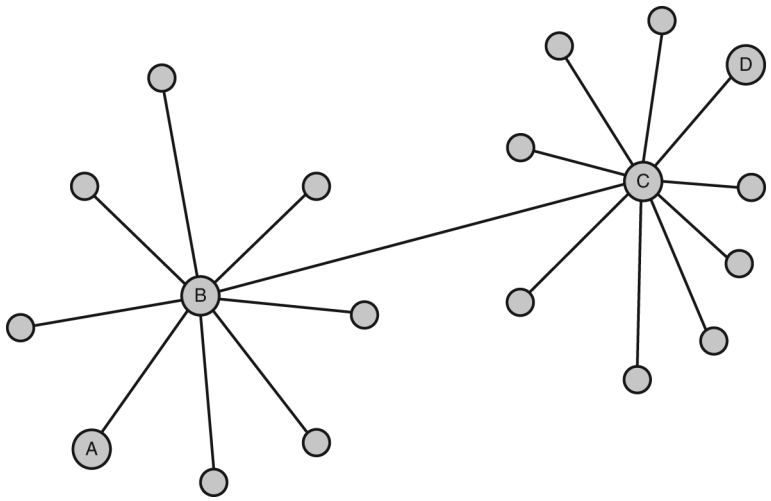

Рис. 12.7. Количество передач между узлами

В WCF имеется очень простой способ ограничить количество передач на уровне контракта о сообщении. В листинге 12.8 показан контракт о сообщении `SearchRequest` для отправки запроса на поиск в ячеистой сети. Один из членов контракта снабжен атрибутом `PeerHopCount`, с которым можно ассоциировать це-

лое значение. Это значение уменьшается на единицу всякий раз, как сообщение передается другому узлу. Когда счетчик обращается в нуль, сообщение перестает передаваться соседним узлам. Тем самым мы ограничиваем число узлов, посещаемых одним сообщением. В нашем примере максимальное количество передач равно 3.

Листинг 12.8. Контракт о сообщении с атрибутом PeerHopCount

```
[MessageContract]
public class SearchRequest
{
   [PeerHopCount]
   private int _hopCount;

   [MessageBodyMember]
   private string _query;

   public string Query
   {
      get { return _query; }
      set { _query = value; }
   }

   [MessageBodyMember]
   private PeerInstance _participant;

   public PeerInstance Participant
   {
      get { return _participant; }
      set { _participant = value; }
   }

   public SearchRequest()
   {
   }

   public SearchRequest(string query)
   {
      _hopCount = 3;
      _query = query;
      _participant = new PeerInstance();
   }

   public SearchRequest(int hopCount, string query)
               : this(query)
   {
      _hopCount = hopCount;
   }
}
```

WCF предоставляет инфраструктуру для создания пиринговых приложений в ячеистых сетях. Но не существует никаких средств для обнаружения и спонтанного взаимодействия с другими узлами в сети. Для этого нам понадобятся новые возможности, встроенные в Windows Vista и .NET Framework 3.5. Речь идет о технологиях People Near Me (Кто рядом со мной) и Windows Contacts and Invitations (Контакты и приглашения). Ниже мы рассмотрим эти функции и продемонстри-

руем работу с ними с помощью классов из нового пространства имен `System.Net.PeerToPeer`.

Технология People Near Me

В Windows Vista имеется возможность обнаружить людей, подключенных к одной с вами локальной подсети, и пригласить их к совместной работе. В Windows XP такой функции нет. Функцию People Near Me можно сконфигурировать на панели управления или с помощью ассоциированного приложения в лотке. На рис. 12.8 изображено диалоговое окно для конфигурирования People Near Me.

Вы можете задать информацию о собственном присутствии: имя и картинку. Кроме того, можно указать, кому разрешено посылать вам приглашения. По умолчанию посылать приглашение может кто угодно, но можно дать это право лишь доверенным контактам или запретить вовсе. На рис. 12.9 показаны возможные варианты.

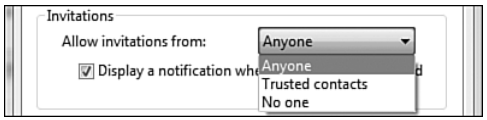

Рис. 12.9. Кому разрешено посылать приглашение

Технология Windows Contacts

В Windows Vista можно отслеживать свои контакты с помощью новой технологии Windows Contacts. Windows Contacts – это место для централизованного хранения списка контактов. На 12.10 показана папка Windows Contacts со списком контактов.

По умолчанию для каждого пользователя создается контакт, содержащий имя и картинку. Часто этот контакт называется «Me» (Я). Технологию Windows Contacts можно использовать для совместной работы с людьми, включенными в список контактов, по электронной почте или иными способами, например, организуя в удобный момент встречи с помощью программы Windows Meeting Space. Одна из самых важных концепций, заложенных в Windows Contacts, – это доверенные контакты, то есть те, кто обменялся с вами контактом Me. Один из самых распространенных способов обмена контактной информацией – электронная почта.

Список контактов можно редактировать, дважды щелкнув мышью по контакту в папке Windows Contacts. На рис. 12.11 показана страница свойств контакта Windows.

Приглашения

Приложение может воспользоваться технологией People Near Me для отправки другим людям приглашения к совместной работе. На рис. 12.12 показано, как выглядит приглашение на встречу. Получив приглашение, пользователь может просмотреть его (View), отклонить (Decline) или проигнорировать (Dismiss).

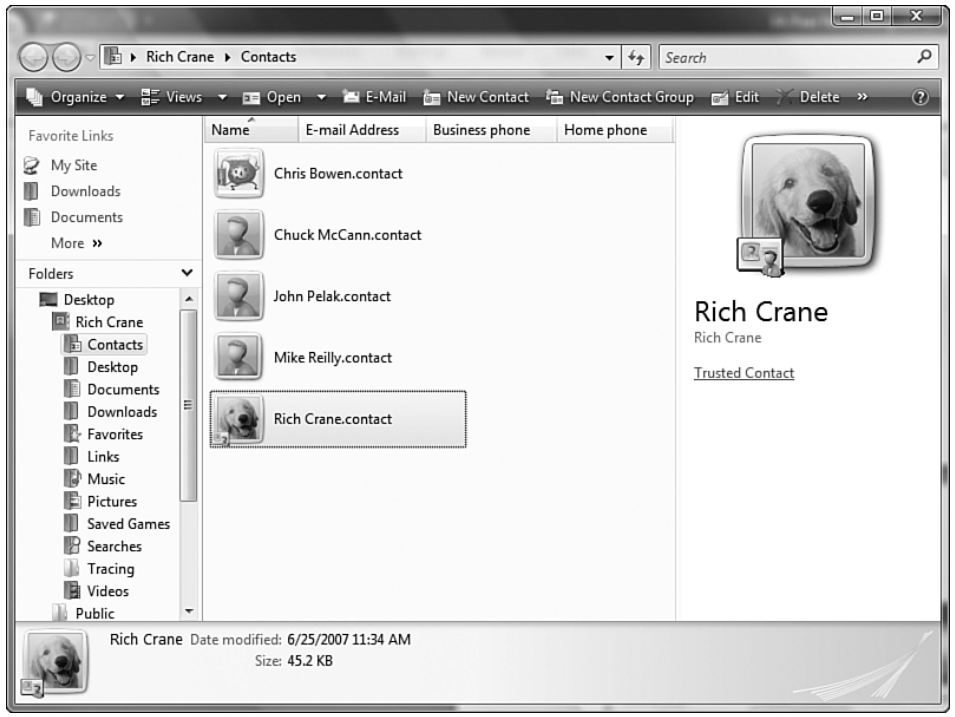

Рис. 12.10. Папка Windows Contacts

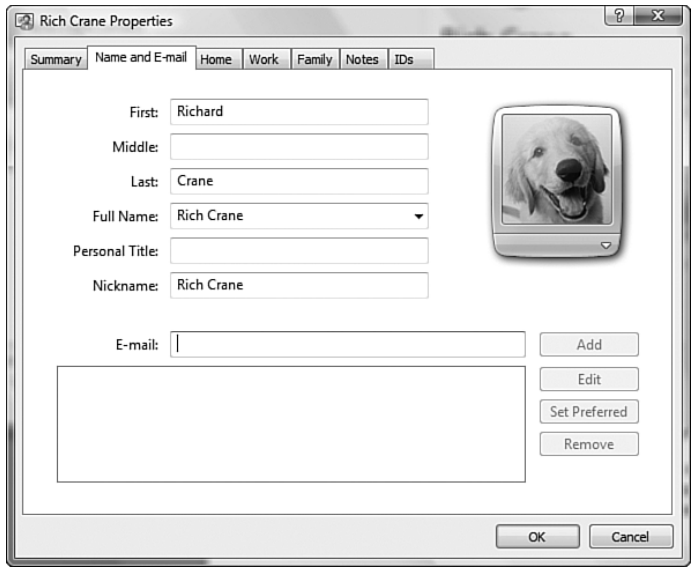

Рис. 12.11. Свойства контакта Windows

При выборе Decline приложению-отправителю в ответ посылается сообщение о том, что приглашение отклонено. Если же пользователь выберет вариант Dismiss, то приглашение будет проигнорировано, а приложение-отправитель не получит никакого ответа. По истечении таймаута приглашение перестает действовать. При выборе варианта View пользователь получает возможность просмотреть приглашение. На рис. 12.13 приведен пример приглашения, отправленного программой Windows Meeting Space. В нем есть ряд важных сведений: кто прислал приглашение, является ли приглашающий доверенным контактом и какое приложение будет запущено, если принять приглашение. В этот момент пользователь может принять приглашение, нажав кнопку Accept.

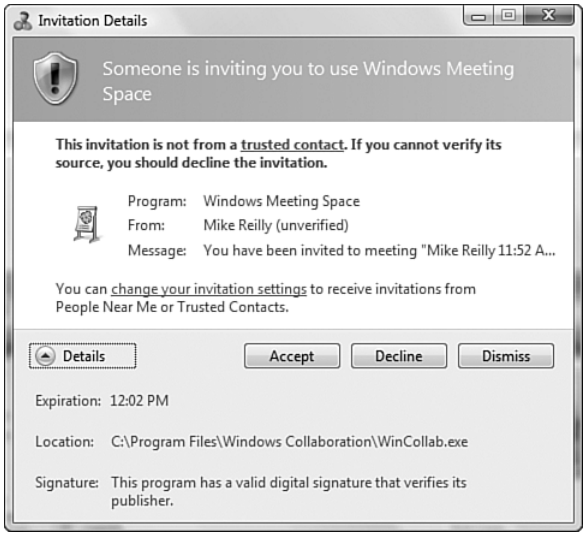

Рис. 12.13. Детали приглашения, отправленного программой Windows Meeting Space

Пространство имен System.Net.PeerToPeer.Collaboration

Технологии People Near Me и Windows Contacts and Invitations позволяют любому приложению инициировать совместную работу. До выхода .NET Framework 3.5 разработчикам, чтобы воспользоваться этими функциями, приходилось работать на уровне неуправляемого API, то есть либо писать на C++, либо создавать обертывающие сборки с помощью механизма P/Invoke. В .NET Framework 3.5 появились управляемые классы для доступа к инфраструктуре People Near Me, Windows Contacts and Invitations, встроенной в Windows Vista. Они находятся в пространстве имен `System.Net.PeerToPeer`. Чтобы ими воспользоваться, необходимо включить в проект ссылку на сборку `System.Net`. Мы покажем, как применять эти классы, на примере приложения Peer Chat (рис. 12.14).

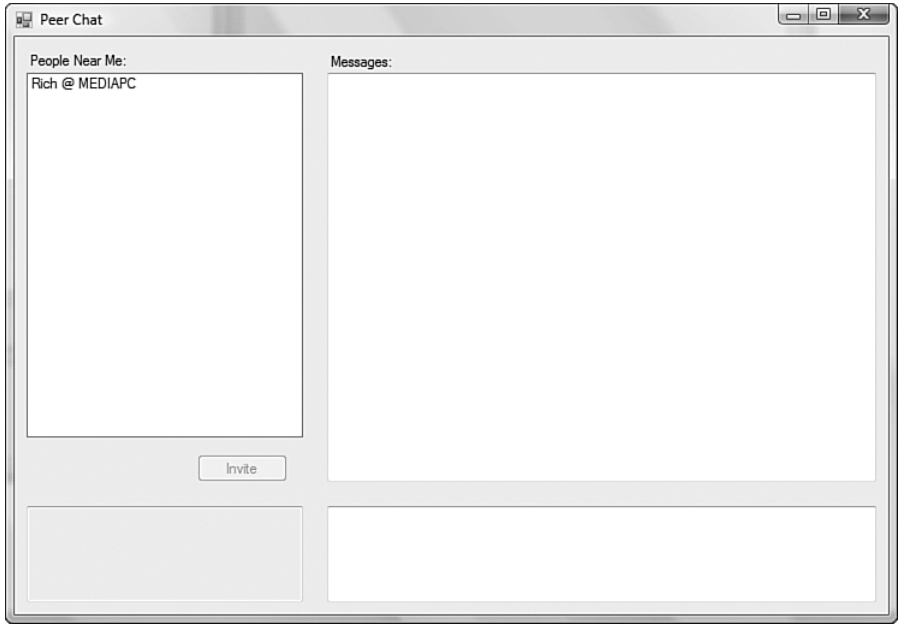

Рис. 12.14. Окно приложения Peer Chat

Прежде всего, приложение должно зарегистрироваться в инфраструктуре совместной работы Windows Vista. Иначе вы не сможете посылать приглашения, в которых предлагается запустить это приложение. В листинге 12.9 показано, как Peer Chat регистрирует себя. Для этого мы вызываем метод `Register` статического класса `PeerCollaboration`, передавая ему экземпляр класса `PeerApplication`. В этом экземпляре находится описание пирингового приложения, в том числе его идентификатор и краткая характеристика.

Листинг 12.9. Регистрация пирингового приложения

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.PeerToPeer;
using System.Net.PeerToPeer.Collaboration;
using System.Text;

namespace PeerChat
{
   public class PeerChatApplication
   {
      private static PeerApplication PeerChatPeerApplication;
      private static Guid PeerChatAppId =
            ➥new Guid("4BC6F59E-124E-4e75-8CD9-BB75BCA78CA8");
      private static string PeerChatDescription =
```

```
            "Пример пирингового приложения.";

      static PeerChatApplication()
      {
         PeerChatPeerApplication =
               new PeerApplication(PeerChatAppId,
                     PeerChatDescription,
                     null,
                  System.Windows.Forms.Application.ExecutablePath,
                     null,
                     PeerScope.All);
      }

      public static void Register()
      {
         PeerApplicationCollection peerAppsColl =
             PeerCollaboration.GetLocalRegisteredApplications(
             PeerApplicationRegistrationType.AllUsers);

         // Вы полюбите LINQ! Смотрите, как здорово!
         IEnumerable<PeerApplication> findPeerApp =
               from peerApp in
               PeerCollaboration.GetLocalRegisteredApplications(
               PeerApplicationRegistrationType.AllUsers)
               where peerApp.Id == PeerChatAppId
               select peerApp;

         if (findPeerApp.Count<PeerApplication>() != 0)
            PeerCollaboration.UnregisterApplication(
                  PeerChatPeerApplication,
                  PeerApplicationRegistrationType.AllUsers);

         PeerCollaboration.RegisterApplication(
            PeerChatPeerApplication,
            PeerApplicationRegistrationType.AllUsers);
      }

      public static void UnRegister()
      {
            PeerCollaboration.UnregisterApplication(
               PeerChatPeerApplication,
               PeerApplicationRegistrationType.AllUsers);
         }
      }
}
```

Далее приложение может вывести список людей в локальной подсети, чтобы пользователь мог пригласить их к совместной работе. Для этого перебираются все, кто зарегистрировался в инфраструктуре People Near Me. Если пользователь хочет получать список соседей, то должен зарегистрироваться сам. Чтобы помочь ему в этом деле, мы создали вспомогательный класс `PeopleNearMeHelper`, предлагающий зарегистрироваться еще до первого запроса (листинг 12.10). Этот класс вызывает метод `SignIn` статического класса `PeerCollaboration`. После того как пользователь зарегистрируется, он сможет вызывать статический ме-

тод `GetPeersNearMe`, возвращающий набор экземпляров класса `PeerNearMe`, каждый из который содержит описание человека, зарегистрированного в инфраструктуре People Near Me в пределах локальной подсети.

Листинг 12.10. Кто рядом со мной

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.PeerToPeer;
using System.Net.PeerToPeer.Collaboration;
using System.Text;

namespace PeerChat
{
   public class PeopleNearMeHelper
   {
      public PeopleNearMeHelper()
      {
         PeerCollaboration.SignIn(PeerScope.All);
      }

      public PeerNearMeCollection PeopleNearMe
      {
         get
         {
            return PeerCollaboration.GetPeersNearMe();
         }
      }
   }
}
```

Теперь нужно послать приглашение другому пользователю. В приложении Peer Chat под совместной работой понимается чат. Но перед тем как посылать приглашение, нужно отправить кое-какую дополнительную информацию, чтобы инициировать процесс коммуникации. Напомним, что собственно коммуникацией занимается WCF и инфраструктура пирингового канала. А, значит, мы должны указать имя ячейки, в которой мы будем общаться, и пароль для доступа к ней. Эта дополнительная информация посылается вместе с приглашением. В листинге 12.11 показано, как отправлять и принимать приглашения в контексте инфраструктуры совместной работы в Windows Vista. Для отправки приглашения служит метод `Invite` или `InviteAsync` класса `PeerNearMe`. Рекомендуется применять метод `InviteAsync`, иначе пользовательский интерфейс блокируется на все время, пока адресат раздумывает, принять ли ему приглашение. Оба метода позволяют посылать дополнительные данные в виде массива байтов. В данном случае мы приложим к сообщению имя ячейки и пароль. Если вы опасаетесь, что таким образом мы выставляем имя и пароль на всеобщее обозрение, то успокойтесь. Инфраструктура People Near Me передает все данные по защищенному соединению.

Листинг 12.11. Отправка и получение приглашений

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;
using System.Net.PeerToPeer;
using System.Net.PeerToPeer.Collaboration;
using System.Text;

namespace PeerChat
{
   public class InvitationHelper
   {
      private static PeerApplication PeerChatPeerApplication;
      private static Guid PeerChatAppId =
               new Guid("4BC6F59E-124E-4e75-8CD9-BB75BCA78CA8");
      private static string PeerChatDescription =
               "Пример пирингового приложения.";

      static InvitationHelper()
      {
         PeerChatPeerApplication =
               new PeerApplication(PeerChatAppId,
                     PeerChatDescription,
                     null,
                     System.Windows.Forms.Application.ExecutablePath,
                     null,
                     PeerScope.All);
      }

      public static PeerInvitationResponseType
               Invite(PeerNearMe personTo, Guid chatId, Guid meshPassword)
      {
         byte[] data;
         using (MemoryStream ms = new MemoryStream())
         {
            using (StreamWriter sw = new StreamWriter(ms))
            {
               sw.Write(chatId.ToString());
               sw.WriteLine();
               sw.Write(meshPassword.ToString());
            }
            data = ms.ToArray();
         }

         PeerInvitationResponse response =
                  personTo.Invite(PeerChatPeerApplication,
                        "Вас приглашают в чат.", data);
                  return response.PeerInvitationResponseType;
      }

      public static void InviteAsync(PeerNearMe personTo,
                  Guid chatId, Guid meshPassword)
      {
         byte[] data;
```



```
         using (MemoryStream ms = new MemoryStream())
         {
            using (StreamWriter sw = new StreamWriter(ms))
            {
               sw.Write(chatId.ToString());
               sw.WriteLine();
               sw.Write(meshPassword.ToString());
            }
            data = ms.ToArray();
         }

         object userToken = Guid.NewGuid();
         personTo.InviteAsync(PeerChatPeerApplication,
                  "Вас приглашают в чат.", data, userToken);
      }

      public static bool IsLaunched
      {
         get
         {
            return
               (PeerCollaboration.ApplicationLaunchInfo != null) &&
               (PeerCollaboration.ApplicationLaunchInfo.Data != null);
         }
      }

      public static Guid ChatId
      {
         get
         {
            Guid chatId;

            using (MemoryStream ms =
                     new MemoryStream(
                           PeerCollaboration.ApplicationLaunchInfo.Data))
            {
               using (StreamReader sr = new StreamReader(ms))
               {
                  string chatIdString = sr.ReadLine();
                  string meshPasswordString = sr.ReadToEnd();

                  chatId = new Guid(chatIdString);
               }
            }

            return chatId;
         }
      }

      public static Guid MeshPassword
      {
         get
         {
            Guid meshPassword;

            using (MemoryStream ms =
```

```
                    new MemoryStream(
                          PeerCollaboration.ApplicationLaunchInfo.Data))
            {
               using (StreamReader sr = new StreamReader(ms))
               {
                  string chatIdString = sr.ReadLine();
                  string meshPasswordString = sr.ReadToEnd();

                  meshPassword = new Guid(meshPasswordString);
               }
            }

            return meshPassword;
         }
      }
   }
}
```

И последнее, что нам осталось рассмотреть, – как определить, запущено ли приложение в ответ на приглашение, полученное от People Near Me. Для этого понадобится экземпляр класса `PeerApplicationLaunchInfo`, который является значением свойства `ApplicationLaunchInfo` статического класса `PeerCollaboration`. В листинге 12.11 показано также, как можно воспользоваться этим классом для ответа на вопрос, запущено ли приложение в ответ на приглашение, и для получения дополнительных данных, поступивших вместе с приглашением.

Организация направленного вещания с помощью заказной привязки

Типичная ошибка при работе с пиринговым транспортным каналом – предполагать, что он поддерживает направленное вещание в ячеистой сети. Говоря «направленное», мы имеем в виду, что сообщение можно послать конкретному узлу, рассчитывая, что оно дойдет до него через соседей. Пиринговый транспортный канал так не умеет. Это ограничивает наши возможности по разработке пиринговых приложений разных видов, так как отправляемые сообщения рассылаются всем узлам. Однако, зная, как приняться за дело, и приложив минимум усилий, некоторые ограничения можно снять.

Есть несколько типов направленного вещания, в частности, один-одному и многие-одному. Коммуникация типа один-одному означает возможность послать сообщение одному конкретному узлу в ячеистой сети. В случае коммуникации типа многие-одному узел может ответить инициатору запроса. Для отправки сообщения один-одному необходим механизм маршрутизации по сети до получателя. Обычно применяются маршрутные индексы, позволяющие ранжировать соседей в соответствии с вероятностью того, что они смогут доставить запрос получателю. К сожалению, это решение слишком сложно, и у нас не хватит времени рассмотреть такой подход к направленному вещанию. Поэтому обратимся к более простой форме – сценарию многие-одному.

Коммуникация типа многие-одному позволяет узлу отправлять ответное сообщение инициатору запроса. Для реализации такого сценария применяются два подхода. Первый – когда ответ посылается инициатору по ячеистой сети так же, как в случае один-одному. Второй подход подразумевает, что инициатор посылает адрес, по которому он готов принимать ответы, и этот вариант гораздо проще реализовать с использованием имеющейся инфраструктуры WCF. Для решения нам потребуется создать составной транспортный канал, который объединяет два готовых односторонних канала, так чтобы запросы посылались по одному, а ответы принимались по другому. Для отправки сообщений мы возьмем пиринговый транспортный канал, а для приема ответов – канал TCP. Кроме того, мы еще изменим канальную форму, воспользовавшись элементом привязки `CompositeDuplexBindingElement`, который обеспечивает дуплексный обмен сообщениями по составному транспортному каналу. В листинге 12.12 приведен код класса `CompositeTransportBindingElement`, служащего для создания асимметричного транспорта.

Листинг 12.12. Класс CompositeTransportBindingElement

```
using System;
using System.Text;
using System.Collections.Generic;
using System.ServiceModel.Channels;

namespace EssentialWCF.PeerApplication.Bindings
{
   public class CompositeTransportBindingElement<TChannelBinding,
TListenerBinding>
      : TransportBindingElement
      where TChannelBinding : Binding, new()
      where TListenerBinding : Binding, new()
   {
      TChannelBinding channelBinding;
      TListenerBinding listenerBinding;

      public CompositeTransportBindingElement(
TChannelBinding channelBinding, TListenerBinding listenerBinding)
      {
         this.channelBinding = channelBinding;
         this.listenerBinding = listenerBinding;
      }

      public CompositeTransportBindingElement(
               CompositeTransportBindingElement<TChannelBinding,
                                    TListenerBinding> other)
               : base(other)
      {
         this.channelBinding = (TChannelBinding)other.channelBinding;
         this.listenerBinding = (TListenerBinding)other.listenerBinding;
      }

      public TChannelBinding ChannelBinding
```

```
{
   get { return this.channelBinding; }
}

public TListenerBinding ListenerBinding
{
   get { return this.listenerBinding; }
}

public override bool CanBuildChannelFactory<TChannel>(
                  BindingContext context)
{
   ThrowIfContextIsNull(context);

   return channelBinding.CanBuildChannelFactory<TChannel>(
               context.BindingParameters);
}

public override bool CanBuildChannelListener<TChannel>(
                  BindingContext context)
{
   ThrowIfContextIsNull(context);

   return listenerBinding.CanBuildChannelListener<TChannel>(
               context.BindingParameters);
}

public override IChannelFactory<TChannel>
               BuildChannelFactory<TChannel>(BindingContext context)
{
   ThrowIfContextIsNull(context);

   return channelBinding.BuildChannelFactory<TChannel>(
            context.BindingParameters);
}

public override IChannelListener<TChannel>
               BuildChannelListener<TChannel>(BindingContext context)
{
   ThrowIfContextIsNull(context);

   return listenerBinding.BuildChannelListener<TChannel>(
                  context.ListenUriBaseAddress,
                  context.ListenUriRelativeAddress,
                  context.ListenUriMode,
                  context.BindingParameters);
}

public override BindingElement Clone()
{
   return new CompositeTransportBindingElement<TChannelBinding,
                     TListenerBinding>(this);
}

public override T GetProperty<T>(BindingContext context)
```



```
      {
         ThrowIfContextIsNull(context);

         T result = this.channelBinding.GetProperty<T>(
                        context.BindingParameters);
         if (result != default(T))
               return result;

         result = this.listenerBinding.GetProperty<T>(
                        context.BindingParameters);
         if (result != default(T))
               return result;

         return context.GetInnerProperty<T>();
      }

      public override string Scheme
      {
         get
         {
            return listenerBinding.Scheme;
         }
      }
   }
}
```

Одного элемента `CompositeTransportBindingElement` недостаточно для решения. Мы должны создать привязку, объединяющую несколько элементов, необходимых для организации составного транспортного канала. Подчеркнем, что клиент и сервер пользуются разными привязками. Раз транспорт асимметричен, то и привязка должна быть асимметричной. В листинге 12.13 демонстрируется создание разных привязок.

Листинг 12.13. Заказные привязки с элементом CompositeTransportBindingElement

```
using System;
using System.Collections.Generic;
using System.Net;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;

using EssentialWCF.PeerApplication.Bindings;
using EssentialWCF.PeerApplication.Helpers;
using EssentialWCF.PeerApplication.Encoder;

namespace EssentialWCF.PeerApplication.Helpers
{
   public static class BindingHelper
   {
      public static Binding CreateClientBinding()
      {
         Uri callbackUri = new Uri(UriHelper.GetSearchCallbackUri());
```

```
   ListenUriBindingElement listenUri = new
               ListenUriBindingElement(callbackUri, "",
               ListenUriMode.Explicit);
   BinaryMessageEncodingBindingElement binaryEncoder =
               new BinaryMessageEncodingBindingElement();
   binaryEncoder.MessageVersion =
               MessageVersion.Soap12WSAddressingAugust2004;
   GZipMessageEncodingBindingElement encoder =
               new GZipMessageEncodingBindingElement(binaryEncoder);
   OneWayBindingElement oneWay = new OneWayBindingElement();
   CompositeDuplexBindingElement duplex =
               new CompositeDuplexBindingElement();
   TcpTransportBindingElement tcpTransport =
               new TcpTransportBindingElement();
   tcpTransport.ManualAddressing = false;
   tcpTransport.PortSharingEnabled = false;
   tcpTransport.TeredoEnabled = true;
   PeerTransportBindingElement peerTransport =
               new PeerTransportBindingElement();
   peerTransport.ManualAddressing = false;
   PnrpPeerResolverBindingElement pnrpResolver =
               new PnrpPeerResolverBindingElement();

   CustomBinding tcpBinding = new CustomBinding(oneWay,
               listenUri, encoder, tcpTransport);

   encoder.MessageVersion =
               MessageVersion.Soap12WSAddressing10;
   CustomBinding peerBinding = new CustomBinding(pnrpResolver,
               encoder, peerTransport);

   CompositeTransportBindingElement<CustomBinding,
               CustomBinding> compositeTransport =
                  new CompositeTransportBindingElement<CustomBinding,
                           CustomBinding>(peerBinding, tcpBinding);

   duplex.ClientBaseAddress = callbackUri;

   return new CustomBinding(duplex, compositeTransport);
}

public static Binding CreateServerBinding()
{
   BinaryMessageEncodingBindingElement binaryEncoder =
               new BinaryMessageEncodingBindingElement();
   binaryEncoder.MessageVersion =
               MessageVersion.Soap12WSAddressingAugust2004;
   GZipMessageEncodingBindingElement encoder =
               new GZipMessageEncodingBindingElement(binaryEncoder);
   OneWayBindingElement oneWay = new OneWayBindingElement();
   CompositeDuplexBindingElement duplex =
               new CompositeDuplexBindingElement();
   TcpTransportBindingElement tcpTransport =
               new TcpTransportBindingElement();
   tcpTransport.ManualAddressing = false;
```



```
   tcpTransport.PortSharingEnabled = false;
   tcpTransport.TeredoEnabled = true;
   PeerTransportBindingElement peerTransport =
               new PeerTransportBindingElement();
   peerTransport.ManualAddressing = false;
   PnrpPeerResolverBindingElement pnrpResolver =
               new PnrpPeerResolverBindingElement();

   CustomBinding tcpBinding = new CustomBinding(oneWay,
               encoder, tcpTransport);
   encoder.MessageVersion =
               MessageVersion.Soap12WSAddressing10;
   CustomBinding peerBinding = new CustomBinding(pnrpResolver,
               encoder, peerTransport);

   CompositeTransportBindingElement<CustomBinding,
               CustomBinding> compositeTransport =
                  new CompositeTransportBindingElement<CustomBinding,
                        CustomBinding>(tcpBinding, peerBinding); ;

   return new CustomBinding(duplex, compositeTransport);
   }
}
}
```

И последнее замечание – эта привязка не допускает автоматическую адресацию внутри WCF. К решению этой проблемы есть два подхода. Лучше всего воспользоваться заказным элементом привязки или, быть может, инспектором сообщений. Этот подход позволяет правильно увязать адресацию со стеком каналов, и потому рекомендуется. Более простой способ – разобраться с адресацией вручную в коде приложения, его мы и продемонстрируем в листинге 12.14, где приведен код клиента. Для ручной адресации мы копируем в свойство `OutgoingMessageHeaders.ReplyTo` значение `InnerChannel.LocalAddress`. Локальный адрес – это тот адрес, по которому клиент ожидает ответные сообщения.

Листинг 12.14. Ручная адресация на стороне клиента

```
public void Search(SearchRequest request)
{

   SearchClient client = ClientHelper.GetSearchClient();

   using (OperationContextScope ctx =
            new OperationContextScope(client.InnerDuplexChannel))
   {
      string LocalAddress =
                  client.InnerChannel.LocalAddress.ToString();
      OperationContext.Current.OutgoingMessageHeaders.ReplyTo =
                  client.InnerChannel.LocalAddress;
      try
      {
         client.Search(request);
      }
```

```
      catch (CommunicationException ex)
      {
         Debug.WriteLine(ex.Message);
      }
   }
}
```

В листинге 12.15 такой же прием применен на стороне сервера, который копирует адрес `IncomingMessageHeaders.ReplyTo` в свойство `OutgoingMessageHeaders.To`. Это необходимо, чтобы посылать сообщения клиенту в соответствии с контрактом обратного вызова.

У решения на основе составного канала тот недостаток, что клиент должен создавать экземпляр службы, чтобы получать сообщения обратного вызова. Инфраструктура каналов в WCF создает для приема таких запросов прослушиватель. Однако мы задействовали отдельный коммуникационный канал, а это означает, что для получения сообщений обратного вызова клиент должен быть достижим напрямую. Это исключает клиентов, находящихся за такими сетевыми устройствами, как брандмауэр и NAT-маршрутизатор. Один из способов преодолеть это ограничение состоит в том, чтобы воспользоваться адресацией IPv6 и включить режим Teredo в транспортном канале TCP. Teredo – это технология прохода сквозь транслятор сетевых адресов (NAT), позволяющая туннелировать через NAT-маршрутизаторы IPv6-трафик, предназначенный узлам внутренней сети, поддерживающим протокол IPv6. В листинге 12.16 показано, как включить режим Teredo с помощью заказной привязки в конфигурационном файле.

Листинг 12.16. Включение режима Teredo с помощью заказной привязки

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <customBinding>
            <binding name="CustomBindingWithTeredo">
               <tcpTransport teredoEnabled="true" />
            </binding>
         </customBinding>
      </bindings>
   </system.serviceModel>
</configuration>
```

Большинство имеющихся в продаже маршрутизаторов для домашних сетей поддерживают технологию NAT, чтобы несколько компьютеров могли пользоваться общим выходом в Интернет. Чтобы транспортный канал TCP мог работать в режиме Teredo, его необходимо включить на уровне компьютера. На рис. 12.15 показано, как это сделать из командной строки с помощью утилиты `netsh`.

Режим Teredo устанавливается на уровне компьютера, поэтому, если он включен, то внешнему миру становятся видны все службы, поддерживающие протокол IPv6. В частности это относится к удаленному рабочему столу (Remote Desktop) и серверу Internet Information Services (IIS). Поэтому использовать режим Teredo

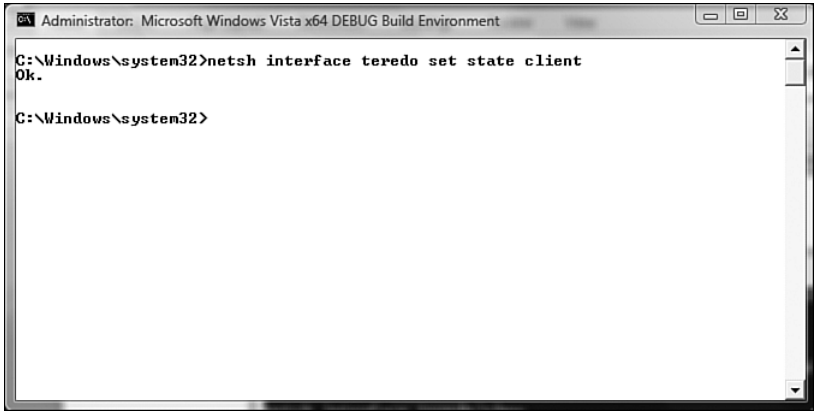

Рис. 12.15. Включение режима Teredo с помощью netsh

нежелательно из соображений безопасности, и имеет смысл поискать альтернативы. Наконец, стоит отметить, что технология Teredo зависит от доступности центрального Teredo-сервера. Он может находиться как в Интернете, так и внутри компании. Корпорация Microsoft предоставляет публичный сервер по адресу teredo.ipv6.microsoft.com. Дополнительную информацию о Teredo см. на страницах www.microsoft.com/ technet/network/ipv6/teredo.mspx и http://en.wikipedia.org/wiki/Teredo_tunneling.

Еще один подход к решению проблемы клиентов, находящихся за брандмауэром или NAT-маршрутизатором, основан на использовании службы, которая позволяет компьютерам обмениваться между собой сообщениями через центральный *сервер-ретранслятор*. Идея в том, что ретранслятор доступен по Интернету обоим участникам и служит посредником между ними. Таким способом можно раскрыть службу через Интернет, даже если сервер расположен за брандмауэром или NAT-маршрутизатором. Но для реализации такого подхода нужно приложить немало усилий, и той информации, что приведена в этой книге, недостаточно. Однако в Microsoft работают над серией продуктов и технологий, которые призваны облегчить решение этой проблемы. Продукт BizTalk Services – это новый набор Интернет-служб, обеспечивающих контроль идентичности и связность. Одной из них является публичная служба ретрансляции. На момент работы над настоящей книгой продукт BizTalk Services находился на стадии экспериментов, и для поставки на рынок потребуется еще некоторое время. В конечном итоге он станет частью проекта под кодовым названием Oslo, по которому сейчас активно ведутся работы. Oslo – это комбинация различных продуктов Microsoft, призванных улучшить технологию создания приложений, представленных в виде служб, по описанию модели. Дополнительную информацию о проектах BizTalk Services и Oslo см. на сайте http://labs.biztalk.net или на странице http://www.microsoft.com/soa/products/oslo.aspx.

Резюме

Пиринговые приложения – весьма заманчивый способ создания распределенных приложений. Они не нуждаются в выделенном сервере, то есть между клиентом и сервером нет никаких различий. При правильной реализации пиринговые приложения заметно повышают масштабируемость и надежность. Примерами могут служить интернет-пейджеры, игры и файлообменные сети.

Основное препятствие на пути разработки пиринговых приложений – сложность и непонимание принципов их построения. WCF существенно упрощает дело, предоставляя необходимую инфраструктуру. Привязка `netPeerTcpBinding` позволяет организовывать коммуникации в ячеистой сети. Операционная система Windows Vista обладает дополнительными средствами обнаружения и совместной работы, воплощенными в таких технологиях, как протокол Peer Name Resolution Protocol (PNRP), People Near Me и Windows Contacts and Invitations. Совместно WCF и Windows Vista образуют платформу, на которой можно создавать пиринговые приложения.

Глава 13. Средства программирования Web

Говоря о средствах программирования Web, мы имеем в виду набор технологий, позволяющих создавать службы, к которым можно обращаться через Web. Их немало. Мы уже упоминали выше, что WCF дает возможность разрабатывать службы, совместимые со спецификациями WS-*, в основе которых лежат SOAP, HTTP и XML. Обычно при построении таких служб применяют подход, основанный на сервисно-ориентированной архитектуре.

В основу сервисно-ориентированных архитектур положены четыре основных принципа:

- границы определены явно;
- службы автономны;
- службы разделяют схему и контракт, но не класс;
- совместимость служб определяется политикой (см. http://msdn.microsoft.com/ msdnmag/issues/04/01/Indigo/default.aspx).

Архитектура служб может быть и иной, например Representational State Transfer (REST). Этот стиль описан в диссертации Роя Филдинга (Roy Fielding) (www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm). REST базируется на следующих принципах:

- для отделения пользовательского интерфейса от хранения данных используется клиент-серверная архитектура;
- при взаимодействии клиента и сервера состояние не запоминается;
- эффективность доступа к сети повышается за счет кэширования;
- компоненты системы взаимодействуют между собой, придерживаясь единообразного интерфейса;
- всю систему можно разбить на ряд уровней.

REST часто называют архитектурным стилем, потому что описанные принципы легко обнаружить в современных архитектурах Web. Мы упомянули REST рядом с сервисно-ориентированной архитектурой, поскольку в современных системах эти два подхода превалируют. Важно осознавать, что WCF не навязывает никакого архитектурного стиля или подхода к построению служб, а предоставляет средства и механизмы, позволяющие конструировать службы самыми разными способами. В этой главе нас будут интересовать те средства, которые помогают разработчикам создавать службы именно для Web. Чтобы лучше понять, почему эти средства устроены именно так, а не иначе, надо разобраться в том, как в настоящее время разрабатываются Web-приложения.

Все, что вы хотели знать о URI

Практически все знакомы с URI, потому что именно с их помощью люди путешествуют по сети Web. Чтобы зайти на какой-то ресурс, например HTML-страницу, человек вводит URI в поле адреса в браузере. Зная URI, браузер может скачивать самые разные ресурсы, например: изображения, видео, данные, приложения и т.д. Доступ к ресурсам путем указания их URI-адреса – один из основополагающих принципов архитектурного стиля REST.

В таблице 13.1 приведено несколько примеров ресурсов в Web, доступных с помощью URI.

Таблица 13.1. Примеры URI

URI	Описание
http://finance.yahoo.com/d/quotes?s=MSFT&f=spt1d	Котировки акций Microsoft (MSFT) в формате CSV (значения, разделенные запятыми) от компании Yahoo!
http://finance.google.com/finance/info?q=MSFT	Котировки акций Microsoft (MSFT) в формате JSON от компании Google
http://en.wikipedia.org/wiki/Apple	Страница википедии о яблоках
www.weather.com/weather/local/02451	Прогноз погоды для города Уолтхэм, штат Массачусетс, с сайта Weather.com
www.msnbc.msn.com/id/20265063/	Новость на сайте MSN.com
http://pipes.yahoo.com/pipes/pipe.run?_id=jIM12Ljj2xGAdeUR1vC6Jw&_render=json&merger=eg	События из мира бизнеса (обмен акциями, слияния компаний и т.д.) в формате JSON
http://rss.slashdot.org/Slashdot/slashdot	Синдицированный Web-канал в формате RSS на сайте Slashdot
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key=20701ea0647b482bcb124b1c80db976f&text=stocks	Результат поиска фотографий на сайте Flickr в формате XML

Во всех этих примерах URI-адреса содержат параметры, определяющие, какой ресурс необходим. Параметры передаются либо в строке запроса, либо как часть пути. Следовательно, URI служит средством для идентификации, поиска и получения доступа к ресурсу. Чтобы лучше понять, что мы имеем в виду, рассмотрим URI для получения котировок акций на сайте Google. Очевидно, что в показанном ниже URI параметр S обозначает символ компании и передается службе в строке запроса.

```
http://finance.google.com/finance/info?q=MSFT
```


Остается неясным, соответствует ли такому URI запрос HTTP GET или какой-то иной вид HTTP-запроса. Пока предположим, что это GET. Вместо MSFT в этот URL можно подставить любой другой символ. Значит, с помощью следующего простого URL мы можем идентифицировать целый ряд ресурсов:

```
http://finance.google.com/finance/info?q={StockSymbol}
```

Этот пример помогает понять принципиальные основы идентификации ресурсов в сети Web.

Вездесущий GET

У всех URI в таблице 13.1 есть общая черта – для доступа к ресурсам используется протокол HTTP. Именно он считается основным протоколом Web. Первоначально HTTP предназначался для доступа к HTML-страницам, но с тех пор был обобщен на любые виды ресурсов: изображения, видео, приложения и т.д. Чтобы все это работало, необходимо указать идентификатор ресурса и действие над ним. Каждому допустимому действию соответствует специальный глагол, определенный в спецификации HTTP. В таблице 13.2 перечислены наиболее употребительные глаголы. Способов взаимодействия с ресурсами несколько, но самым распространенным является метод GET. Вслед за ним идет POST, потом PUT, DELETE и некоторые другие.

Таблица 13.2. Наиболее употребительные глаголы HTTP

Глагол	Описание
GET	Получить ресурс, указанный в URI
POST	Послать на сервер данные, адресовав их ресурсу, указанному в URI
PUT	Сохранить данные в ресурсе, указанном в URI
DELETE	Удалить ресурс, указанный в URI
HEAD	То же, что GET, но возвращаются не сами данные ресурса, а только его метаданные

Глаголы HTTP – это основа для взаимодействия с Web-ресурсами. Глагол GET наиболее популярен, потому что именно он используется для получения ресурса. Глаголы HTTP как раз и образуют тот единообразный интерфейс, с помощью которого производится взаимодействие с ресурсами в соответствии с архитектурным стилем REST.

Формат имеет значение

На примере URI в таблице 13.1 продемонстрированы различные форматы данных, применяемые в настоящее время. Это HTML, XML, JSON, RSS, CSV и ряд нестандартных форматов. Стало быть, пока разработчики не отыскали единый формат, пригодный для представления всех ресурсов в Web. В какой-то момент казалось, что все дороги ведут к XML, который должен стать универсальным средством обмена информацией. Например, сообщения, передаваемые по прото-

колу SOAP, который лежит в основе традиционных Web-служб, записываются на диалекте XML. WCF поддерживает протокол SOAP. Заголовки SOAP несут информацию, необходимую для реализации таких возможностей, как независимость от транспортного протокола, безопасность на уровне сообщений и транзакции. Но Web-разработчикам все это может быть и неинтересно, им нужен лишь способ обмена данными. В таких случаях часто применяются более простые форматы, например, Plain-Old-XML (POX – старый добрый XML) JavaScript Object Notation (JSON – объектная нотация в синтаксисе JavaScript).

Формат POX обычно выбирают тогда, когда не нужны развитые средства, описанные в спецификациях WS-*, и хочется избежать присущих SOAP накладных расходов. В таких случаях POX «достаточно хорош». С другой стороны, формат JSON эффективен для возврата данных браузеру, когда на стороне клиента применяется язык JavaScript. Он компактнее, чем SOAP, и в тех случаях, когда желательно уменьшить объем трафика, может заметно повысить производительность и масштабируемость. Таким образом, мы приходим к выводу, что формат имеет значение, и разработчикам необходима возможность работать со всеми форматами, распространенными в Web.

Программирование для Web с помощью WCF

В таблицу 3.3 сведены некоторые наиболее важные средства, доступные разработчикам, применяющим WCF и .NET Framework 3.5. В оставшейся части главы мы рассмотрим, как WCF помогает «программировать Web».

Таблица 13.3. Средства программирования для Web в каркасе .NET Framework 3.5

Средство	Описание
Классы Uri и UriTemplate	Развитая поддержка работы с URI в соответствии с архитектурным стилем REST
Привязка `webHttpBinding`	Новая привязка для поддержки форматов POX и JSON; формальная поддержка глаголов HTTP, включая GET, и диспетчеризация на основе URI
Интеграция с ASP.NET AJAX	Интеграция с ASP.NET AJAX для поддержки прокси-классов на стороне клиента
Синдицирование контента	Классы для публикации и потребления синдицированных Web-каналов в форматах RSS и ATOM

Классы Uri и UriTemplate

Microsoft включила поддержку URI еще в .NET Framework v1.0. Класс `System.Uri` позволяет определить и разобрать URI на основные компоненты. Это очень удобно для передачи URI таким Web-клиентам, как элемент управления `System.Windows.Forms.WebBrowser` или класс `System.Net.WebClient`. Дополнением к `System.Uri` служит класс `System.UriBuilder`. Он позволяет модифицировать имеющийся экземпляр класса `System.Uri`, не создавая нового. Эти два

класса лежат в основе работы с URI для протокола HTTP. Но для поддержки архитектурного стиля REST, применяемого ныне разработчиками, нужны дополнительные возможности.

Из таблицы 13.1 видно, что параметры URI представлены в виде строки запроса или части пути. Классы `System.Uri` и `System.UriBuilder` не позволяют ни создавать, ни разбирать подобные URI. Поэтому был предложен другой подход с использованием именованных макросов (named tokens). Макросами представлены логические параметры, подставляемые в URI. И они же описывают параметры, извлекаемые из URI в ходе лексического разбора. В .NET Framework 3.5 появился новый класс `System.UriTemplate`, который позволяет строить и разбирать URI по шаблону. Этот класс определяет шаблон, содержащий именованные макросы. Макросы заключены в фигурные скобки. Например, шаблон /finance/info?q={symbol} определяет биржевой символ акции, передаваемый как параметр в строке запроса. Макросы могут встречаться не только в строке запроса, но и в пути URI. Например, в шаблоне `/browse/{word}` параметризован путь. Из шаблонов можно порождать экземпляры класса `System.Uri`. Теперь самое время познакомиться с тем, как всего этого добиться с помощью класса `System.UriTemplate`.

Построение URI

В листинге 13.1 приведены два примеры построения экземпляра класса `System.Uri` с помощью `System.UriTemplate`. В первом случае мы пользуемся методом `BindByPosition`, чтобы создать экземпляр `System.Uri` для получения котировок акций с сайта Yahoo. Во втором – методом `BindByName`, чтобы передать набор пар имя/значение для создания экземпляра `System.Uri`, соответствующего котировкам акций на сайте Google.

Листинг 13.1. Связывание параметров с экземпляром класса UriTemplate

```
using System;
using System.Collections.Specialized;

namespace EssentialWCF
{
   class Program
   {
      static void Main(string[] args)
      {
         string symbol = "MSFT";

         // BindByPosition
         Uri YahooStockBaseUri = new Uri("http://finance.yahoo.com");
         UriTemplate YahooStockUriTemplate =
                  new UriTemplate("/d/quotes?s={symbol}&f=sl1t1d1");
         Uri YahooStockUri =
                  YahooStockUriTemplate.BindByPosition(
                           YahooStockBaseUri,
                           symbol);
```

```
        Console.WriteLine(YahooStockUri.ToString());

        // BindByName
        Uri GoogleStockBaseUri = new Uri("http://finance.google.com");
        UriTemplate GoogleStockUriTemplate =
                new UriTemplate("/finance/info?q={symbol}");
        NameValueCollection GoogleParams =
                new NameValueCollection();
        GoogleParams.Add("symbol", symbol);
        Uri GoogleStockUri =
                GoogleStockUriTemplate.BindByName(
                        GoogleStockBaseUri,
                        GoogleParams);
        Console.WriteLine(GoogleStockUri.ToString());

        Console.ReadLine();
      }
   }
}
```

Разбор URI

Выше мы видели, как просто создавать из шаблонов новые экземпляры класса `System.Uri`. В листинге 13.2 показано, как выделить параметры из имеющегося URI. И снова мы приводим два примера. В первом демонстрируется извлечение параметров из строки запроса, а во втором – из пути. В обоих случаях, зная шаблон, мы получаем набор пар имя/значение. В разделе «Создание операций для Web» мы покажем, как класс `UriTemplate` можно применить для диспетчеризации методов Web-службы, исходя из URI.

Листинг 13.2. Сопоставление параметров с шаблоном UriTemplate

```
using System;

namespace UriTemplate102
{
   class Program
   {
      static void Main(string[] args)
      {
         Uri YahooBaseUri = new Uri("http://finance.yahoo.com");
         UriTemplate YahooStockTemplate = new UriTemplate("/d/
quotes?s={symbol}");      Uri YahooStockUri =
                 new Uri("http://finance.yahoo.com/d/quotes?s=MSFT&f=spt1d");
         UriTemplateMatch match =
                 YahooStockTemplate.Match(YahooBaseUri, YahooStockUri);

         foreach (string key in match.BoundVariables.Keys)
               Console.WriteLine(String.Format("{0}: {1}", key,
                        match.BoundVariables[key]));

         Console.WriteLine();

         Uri ReferenceDotComBaseUri =
```



```
                 new Uri("http://dictionary.reference.com");
         UriTemplate ReferenceDotComTemplate =
                 new UriTemplate("/browse/{word}");

         Uri ReferenceDotComUri =
                 new Uri("http://dictionary.reference.com/browse/opaque");
         match = ReferenceDotComTemplate.Match(ReferenceDotComBaseUri
                                 ReferenceDotComUri);

         foreach (string key in match.BoundVariables.Keys)
               Console.WriteLine(String.Format("{0}: {1}", key,
                     match.BoundVariables[key]));

         Console.ReadLine();
      }
   }
}
```

Создание операций для Web

Говоря о создании операций для Web, мы подразумеваем, что хотим раскрывать службу на основе URI, кодировать сообщения без накладных расходов, сопряженных с протоколом SOAP, передавать параметры по протоколу HTTP и представлять возвращаемые данные в формате JSON или POX. WCF предоставляет привязку `webHttpBinding`, которая поддерживает все эти возможности. Первым элементом привязки является новый кодировщик `WebMessageEncodingBindingElement`, который позволяет кодировать сообщения в форматах JSON или POX. Вторым служит транспортный элемент привязки `HttpTransportBindingElement` или `HttpsTransportBindingElement`. Оба обеспечивают коммуникацию по протоколу HTTP, но `HttpsTransportBindingElement` еще и поддерживает безопасность на транспортном уровне.

Размещение с привязкой webHttpBinding

Чтобы поближе познакомиться с привязкой `webHttpBinding`, создадим Web-службу Echo. Этот пример будет совсем простым, но позже мы его расширим. В листинге 13.3 приведен интерфейс `IEchoService`, определяющий контракт о службе с единственным контрактом об операции `Echo`. Отметим, что операция `Echo` снабжена атрибутом `WebGet`. Этот атрибут сообщает привязке `webHttpBinding` о том, что операцию нужно раскрывать по протоколу HTTP с помощью глагола GET.

Листинг 13.3. Интерфейс IEchoService

```
using System;
using System.ServiceModel;
using System.ServiceModel.Web;

[ServiceContract]
public interface IEchoService
```

```
{
   [OperationContract]
   [WebGet]
   string Echo(string echoThis);
}
```

В листинге 13.4 приведен код класса `EchoService`, реализующего интерфейс `IEchoService`. Операция `Echo` принимает параметр `echoThis` и возвращает его же клиенту.

Листинг 13.4. Класс EchoService

```
using System;
using System.ServiceModel;

public class EchoService : IEchoService
{
   #region Члены IEchoService

   public string Echo(string echoThis)
   {
      return string.Format("Вы прислали мне “{0}”.", echoThis);
   }

   #endregion
}
```

Осталось еще разместить службу `EchoService` внутри IIS. В листинге 13.5 приведен конфигурационный файл этой службы с привязкой `webHttpBinding`. Подчеркнем, что сама привязка не определяет формат, в котором раскрывается служба. Для этого необходимо использовать поведение оконечной точки. В нашем распоряжении два таких поведения: `WebHttpBehavior` и `WebScriptEnablingBehavior`. Поведение `WebScriptEnablingBehavior` мы обсудим ниже в разделе «Программирование для Web с использованием AJAX и JSON». А пока сосредоточимся на поведении `WebHttpBehavior`, которое позволяет кодировать сообщения в формате JSON или XML, причем по умолчанию подразумевается XML.

Листинг 13.5. Конфигурационный файл службы EchoService

```
<system.serviceModel>
   <services>
      <service name="EchoService">
         <endpoint address=""
                   behaviorConfiguration="WebBehavior"
                   binding="webHttpBinding" contract="IEchoService"/>
      </service>
   </services>
   <behaviors>
      <endpointBehaviors>
         <behavior name="WebBehavior">
            <webHttp />
         </behavior>
      </endpointBehaviors>
   </behaviors>
</system.serviceModel>
```


На рис. 13.1 показано, что выводит служба `EchoService`, раскрываемая с помощью привязки `webHttpBinding`. Поскольку для операции был задан атрибут `WebGet`, то мы можем вызывать ее, указав URI в браузере. Этот URI выглядит следующим образом: http://localhost/SimpleWebService/EchoService.svc/Echo?echoThis=helloworld.

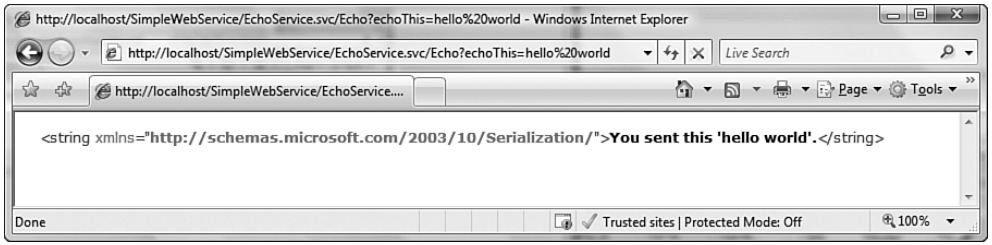

Рис. 13.1. Ответ в браузере от службы, раскрытой с помощью привязки webHttpBinding

Атрибуты WebGet и WebInvoke

Для служб с привязкой `webHttpBinding` можно указывать один из атрибутов `WebGet` или `WebInvoke`. Каждый из них определяет глагол HTTP, формат сообщения и вид тела, необходимые для раскрытия операции. Рассмотрим, в каких ситуациях следует применять тот или другой атрибут.

Атрибут WebGet

Атрибут `WebGet` раскрывает операцию с помощью глагола GET. По сравнению с другими глаголами HTTP у GET есть ряд преимуществ. Во-первых, к оконечной точке можно получить доступ, просто набрав URI в поле адреса. Параметры можно послать в составе URI как часть строки запроса или пути. Во-вторых, клиенты и другие системы на пути к клиенту, например прокси-серверы, могут без труда кэшировать ресурсы, принимая во внимание политику кэширования, объявленную службой. Из-за возможности кэширования атрибут `WebGet` следует применять только для чтения ресурса.

Атрибут WebInvoke

Атрибут `WebInvoke` раскрывает службу с помощью других глаголов HTTP, например: POST, PUT и DELETE. По умолчанию подразумевается POST, но это можно изменить, задав в атрибуте свойство `Method`. Такие операции предназначены для модификации ресурсов.

В листинге 13.6 приведен код службы, операции которой помечены атрибутами `WebGet` и `WebInvoke`. Операция с атрибутом `WebGet` возвращает информацию о заказчике, а операции с атрибутом `WebInvoke` позволяют добавлять и удалять заказчиков. Наконец, свойство `UriTemplate` атрибутов `WebGet` и `WebInvoke` идентифицирует ресурс с помощью URI.

Листинг 13.6. Служба CustomerService

```
using System;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace EssentialWCF
{
   [ServiceContract]
   public class CustomerService
   {
      [OperationContract]
      [WebGet(UriTemplate="/customer/{id}")]
      public Customer GetCustomer(int id)
      {
         Customer customer = null;

         // Извлечь данные о заказчике из базы

         return customer;
      }

      [OperationContract]
      [WebInvoke(Method = "PUT", UriTemplate = "/customer/{id}")]
      public void PutCustomer(int id, Customer customer)
      {
         // Поместить данные о заказчике в базу
      }

      [OperationContract]
      [WebInvoke(Method = "DELETE", UriTemplate = "/customer/{id}")]
      public void DeleteCustomer(int id)
      {
         // Удалить данные о заказчике из базы
      }
   }
}
```

Программирование для Web с использованием AJAX и JSON

До сих пор мы рассматривали размещение службы с помощью привязки `webHttpBinding` и поведения оконечной точки `WebHttpBehavior`. Они дают возможность раскрыть службу в формате POX. Но многим разработчикам не по душе XML, они предпочитают более простой формат JSON, идеальный для приложений, работающих внутри браузера, которым нужны эффективные средства разбора ответа от служб. Кроме того, JSON интегрирован с языком JavaScript, который чаще всего применяется для программирования на стороне Web-клиента. По существу, JSON – это подмножество синтаксиса JavaScript для записи литеральных объектов. Поэтому он вполне способен заменить XML в AJAX-приложениях.

Аббревиатура AJAX означает Asynchronous JavaScript and XML (асинхронный JavaScript и XML). Web-приложения, основанные на технологии AJAX, обладают рядом достоинств по сравнению с традиционными Web-приложениями. Они удобнее для работы и позволяют сократить сетевой трафик. Достигается это за счет того, что отпадает необходимость в полной перезагрузке страницы. Это, в свою очередь, следствие асинхронной коммуникации между браузером и сервером, программируемой на языке JavaScript с помощью класса `XMLHttpRequest`. Поскольку перегружать страницу целиком не нужно, разработчик может построить весьма развитый интерфейс, по своим возможностям приближающийся к персональным приложениям. Такие Web-приложения называются обогащенными (Rich Internet Applications, или RIA).

Интеграция с ASP.NET AJAX

Существует много каркасов для построения Web-приложений на основе AJAX. Один из наиболее популярных – каркас ASP.NET AJAX. В нем заложена продуманная модель клиентской и серверной части, обеспечивающая создание AJAX-приложений. К числу достоинств следует отнести развитую библиотеку клиентских классов, множество Web-элементов управления и автоматическую генерацию клиентских прокси-классов для коммуникации со службами. Кроме того, он построен на базе технологии ASP.NET, предлагаемой Microsoft для создания Web-приложений на платформе .NET. WCF была интегрирована с ASP.NET еще в версии .NET Framework 3.0. А в .NET Framework 3.5 появилась поддержка для приложений ASP.NET AJAX на основе поведения оконечной точки `WebScriptEnablingBehavior`. Это поведение заменяет `WebHttpBehavior`, добавляя поддержку JSON по умолчанию и генерацию клиентских прокси-классов в ASP.NET. Новыми возможностями можно воспользоваться, заменив в конфигурационном файле элемент `webHttp` на `enableWebScript`.

Чтобы продемонстрировать использование привязки `webHttpBinding` и поведения `WebScriptEnablingBehavior`, мы написали приложение для ASP.NET AJAX, которое назвали XBOX 360 Game Review. Оно позволяет пользователям выкладывать обзоры своих любимых игр для приставки XBOX 360. Приложение основано на шаблоне проекта Web-сайта ASP.NET AJAX, который включен в Visual Studio 2008. На рис. 13.2 изображен внешний вид сайта.

Сайт предоставляет ряд функций. Во-первых, в элементе управления `ListBox` отображается список игр. Пользователь может выбрать из списка игру и посмотреть относящиеся к ней комментарии. Кроме того, можно добавлять свои комментарии к любой игре. В листинге 13.7 приведен код службы, реализующей эту функциональность.

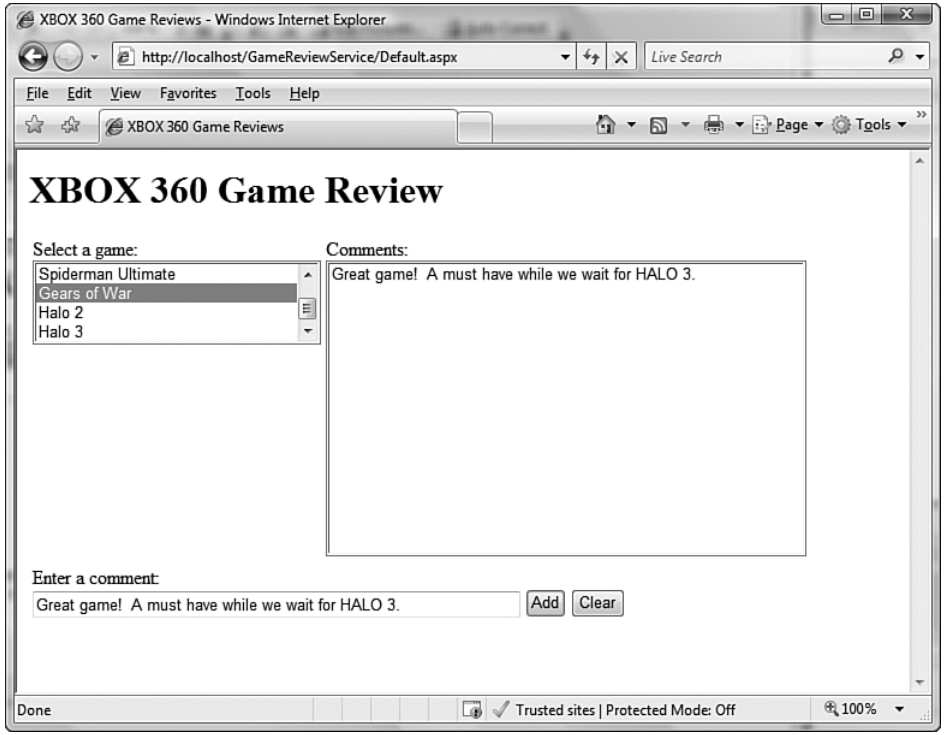

Рис. 13.2. AJAX-приложение XBOX 360 Game Review

Листинг 13.7. Файл GameReviewService.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Activation;
using System.ServiceModel.Web;

namespace EssentialWCF
{
   [ServiceContract(Namespace="EssentialWCF")]
   [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
   [AspNetCompatibilityRequirements(RequirementsMode =
            AspNetCompatibilityRequirementsMode.Allowed)]
   public class GameReviewService
   {
      private string[] gamelist = new string[] { "Viva Pinata",
               "Star Wars Lego", "Spiderman Ultimate",
               "Gears of War", "Halo 2", "Halo 3" };
      private Dictionary<string, List<string>> reviews;

      public GameReviewService()
```



```
      {
         reviews = new Dictionary<string, List<string>>();
         foreach (string game in gamelist)
               reviews.Add(game, new List<string>());
      }

      [OperationContract]
      [WebGet]
      public string[] Games()
      {
         return gamelist;
      }

      [OperationContract]
      [WebGet]
      public string[] Reviews(string game)
      {
         WebOperationContext ctx = WebOperationContext.Current;
         ctx.OutgoingResponse.Headers.Add("Cache-Control", "no-cache");

         if (!reviews.ContainsKey(game))
               return null;

         List<string> listOfReviews = reviews[game];

         if (listOfReviews.Count == 0)
               return new string[] {
                        string.Format("Для игры {0} нет обзоров.",game) };
         else
               return listOfReviews.ToArray();
      }

      [OperationContract]
      [WebInvoke]
      public void AddReview(string game, string comment)
      {
         reviews[game].Add(comment);
      }

      [OperationContract]
      [WebInvoke]
      public void ClearReviews(string game)
      {
         reviews[game].Clear();
      }
   }
}
```

Мы решили разместить эту службу внутри Internet Information Server (IIS). В листинге 13.8 показан описывающий ее файл `GameReviewService.svc`.

Листинг 13.8. Файл GameReviewService.svc

```
<%@ ServiceHost Language="C#" Debug="true"
➥Service="EssentialWCF.GameReviewService"
➥CodeBehind="~/App_Code/GameReviewService.cs" %>
```

В листинге 13.9 показан конфигурационный файл для размещения службы GameReviewService. Самое интересное в нем – привязка webHttpBinding и поведение оконечной точки enableWebScript. При этом включается поддержка JSON и генерируется необходимый ASP.NET клиентский прокси-класс.

Листинг 13.9. Файл web.config

```
<system.serviceModel>
   <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
   <services>
      <service name="EssentialWCF.GameReviewService"
               behaviorConfiguration="MetadataBehavior">
         <endpoint address=""
                   behaviorConfiguration="AjaxBehavior"
                   binding="webHttpBinding"
                   contract="EssentialWCF.GameReviewService"/>
         <endpoint address="mex"
                   binding="mexHttpBinding"
                   contract="IMetadataExchange"/>
      </service>
   </services>
   <behaviors>
      <endpointBehaviors>
         <behavior name="AjaxBehavior">
                   <enableWebScript/>
         </behavior>
      </endpointBehaviors>
      <serviceBehaviors>
         <behavior name="MetadataBehavior">
            <serviceMetadata httpGetEnabled="true" httpGetUrl=""/>
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

Для того чтобы служба GameReviewService работала вместе с ASP.NET, необходимо добавить ссылку на нее с помощью менеджера сценариев ASP.NET ScriptManager. В листинге 13.10 показано, как это сделать с помощью разметки. Невидимо для вас при этом генерируется JavaScript-сценарий, посылаемый браузеру из клиентского прокси-класса. В нашем примере этот сценарий имеет следующий URI: http://localhost/GameReviewService/GameReviewService.svc/js.

Листинг 13.10. Ссылка на службы с помощью ASP.NET ScriptManager

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
   <Services>
      <asp:ServiceReference Path="GameReviewService.svc" />
      </Services>
</asp:ScriptManager>
```

В Web-приложение XBOX 360 Game Review мы включили форму ASP.NET. Из ее кода видно, как из клиентского сценария вызывается служба, а полученные результаты используются для динамического заполнения элементов управления.

Листинг 13.11. Обращение к службе из сценария на стороне клиента

```
<%@ Page Language="C#" AutoEventWireup="true"
➥CodeFile="Default.aspx.cs" Inherits="_Default" %>

<%@ Register Assembly="AjaxControlToolkit"
➥Namespace="AjaxControlToolkit" TagPrefix="cc1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
   <title>XBOX 360 Game Reviews</title>

   <script type="text/javascript">

      function pageLoad() {
      }

   </script>

</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server">
         <Services>
            <asp:ServiceReference Path="GameReviewService.svc" />
         </Services>
      </asp:ScriptManager>

      <script type="text/javascript">
      EssentialWCF.GameReviewService.set_defaultFailedCallback(OnError);
      function ListGames()
      {
         EssentialWCF.GameReviewService.Games(OnListGamesComplete);
      }
      function ListReviews()
      {
         var gameListBox = document.getElementById("GameListBox");
         EssentialWCF.GameReviewService.Reviews(gameListBox.value,
                     OnListReviewsComplete);
      }
      function AddReview()
      {
         var gameListBox = document.getElementById("GameListBox");
         var reviewTextBox = document.getElementById("ReviewTextBox");
         EssentialWCF.GameReviewService.AddReview(gameListBox.value,
                           reviewTextBox.value, OnUpdateReviews);
      }
      function ClearReviews()
      {
         var gameListBox = document.getElementById("GameListBox");
         EssentialWCF.GameReviewService.ClearReviews(gameListBox.value,
                           OnUpdateReviews);
      }
```

```
function OnListGamesComplete(result)
{
   var gameListBox = document.getElementById("GameListBox");
   ClearAndSetListBoxItems(gameListBox, result);
}
function OnListReviewsComplete(result)
{
   var reviewListBox = document.getElementById("ReviewListBox");
   ClearAndSetListBoxItems(reviewListBox, result);
}
function OnUpdateReviews(result)
{
   ListReviews();
}
function ClearAndSetListBoxItems(listBox, games)
{
   for (var i = listBox.options.length-1; i >-1; i-)
   {
      listBox.options[i] = null;
   }

   var textValue;
   var optionItem;
   for (var j = 0; j < games.length; j++)
   {
      textValue = games[j];
      optionItem = new Option( textValue, textValue,
                  false, false);
      listBox.options[listBox.length] = optionItem;
   }
}
function OnError(result)
{
   alert("Error: " + result.get_message());
}
function OnLoad()
{
   ListGames();

   var gameListBox = document.getElementById("GameListBox");
   if (gameListBox.attachEvent) {
         gameListBox.attachEvent("onchange", ListReviews);
   }
   else {
      gameListBox.addEventListener("change", ListReviews, false);
   }
}
Sys.Application.add_load(OnLoad);
</script>

<h1>XBOX 360 Game Review</h1>
<table>
   <tr style="height:250px;vertical-align:top;"><td
         style="width:240px">Select a game:<br /><asp:ListBox
         ID="GameListBox" runat="server"
         Width="100%"></asp:ListBox></td>
```



```
         <td style="width:400px">Comments:<br /><asp:ListBox
         ID="ReviewListBox" runat="server" Width="100%"
         Height="100%"></asp:ListBox></td></tr>
      <tr style="vertical-align:top;"><td colspan="2">
         Enter a comment:<br />
         <asp:TextBox ID="ReviewTextBox" runat="server"
            width="400px"></asp:TextBox>
         <input id="AddReviewButton" type="button" value="Add"
            onclick="AddReview();" />
         <input id="ClearReviewButton" type="button" value="Clear"
            onclick="ClearReviews();" />
      </td></tr>
   </table>
</div>
</form>
</body>
</html>
```

Класс WebOperationContext

При размещении служб с помощью привязки `webHttpBinding` часто приходится читать или изменять контекст HTTP. Это позволяет сделать класс `WebOperationContext`. Для получения доступа к контексту могут быть различные причины. Например, может понадобиться прочитать нестандартные заголовки, содержащие информацию об аутентификации и авторизации, управлении кэшированием. Или установить тип содержимого.

На рис. 13.3 изображено Web-приложение, которое выводит картинки-обои. Оно целиком построено на базе WCF-службы и работает в любом Web-браузере.

В листинге 13.12 приведен код службы `WallpaperService`. Операция `Images` выводит страницу, содержащую список изображений. Эта операция устанавливает заголовок `ContentType`, так чтобы браузер интерпретировал результат как HTML. Кроме того, устанавливается заголовок `Cache-Control`, чтобы можно было добавлять новые изображения без кэширования на стороне браузера. А операция `Image` возвращает одно изображение браузеру. Она устанавливает заголовки `ContentType` и `ETag`.

Исключение SVC-файла. WCF-служба размещается в IIS с помощью SVC-файла. Но это противоречит соглашениям о структуре URI, принятым в REST. Так, для доступа к службе из листинга 13.12 используется следующий URI:
http://localhost/Wallpaper/WallpaperService.svc/images
Можно избавиться от SVC-файла, написав модуль `HttpModule` (только для IIS 7.0), который вызывает метод `HttpContext.RewritePath` для модификации URI. Тогда URI можно будет записать в таком виде:
http://localhost/Wallpaper/WallpaperService/images

Листинг 13.12. Служба WallpaperService

```
using System;
using System.Collections;
using System.Collections.Generic;
```

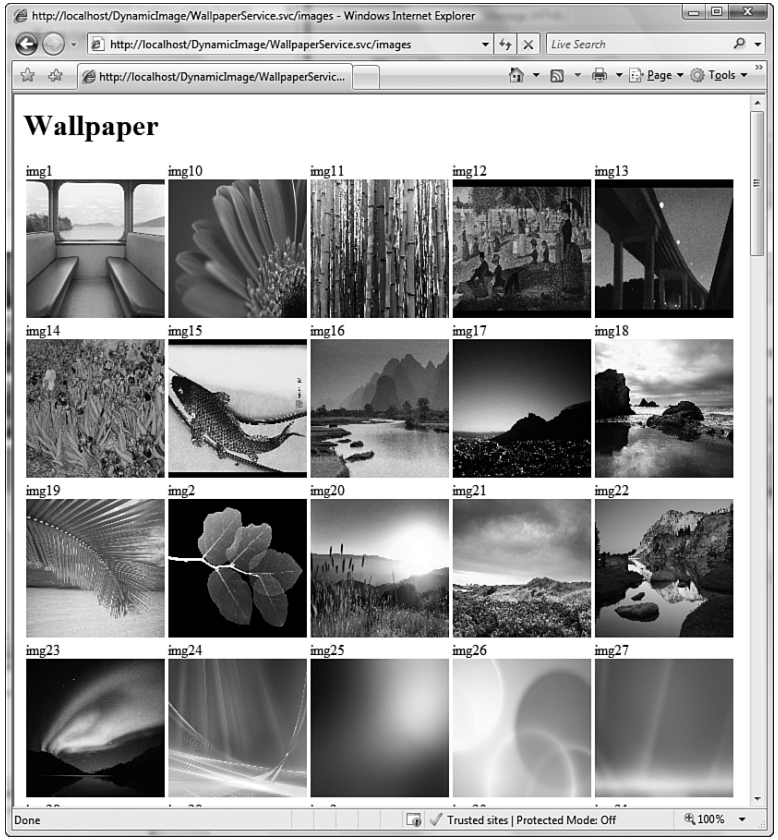

Рис. 13.3. Приложение, показывающее обои

```
using System.IO;
using System.Runtime.Serialization;
using System.Text;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.ServiceModel;
using System.ServiceModel.Activation;
using System.ServiceModel.Web;

namespace EssentialWCF
{
   [DataContract]
   public class Image
   {
      string name;
      string uri;

      public Image()
      {
```



```
      }

      public Image(string name, string uri)
      {
         this.name = name;
         this.uri = uri;
      }

      public Image(string name, Uri uri)
      {
         this.name = name;
         this.uri = uri.ToString();
      }

      [DataMember]
      public string Name
      {
         get { return this.name; }
         set { this.Name = value; }
      }

      [DataMember]
      public string Uri
      {
         get { return this.uri; }
         set { this.uri = value; }
      }
   }

   [ServiceContract]
   [AspNetCompatibilityRequirements(RequirementsMode =
           AspNetCompatibilityRequirementsMode.Required)]
   public class WallpaperService
   {
      private static UriTemplate ImageUriTemplate =
               new UriTemplate("/image/{name}");

      private string ImagePath
      {
         get
         {
            return @"C:\Windows\Web\Wallpaper";
         }
      }

      private Image GetImage(string name, Uri baseUri)
      {
         return new Image(name,
                     ImageUriTemplate.BindByPosition(baseUri,
                     new string[] { name }));
      }

      private void PopulateListOfImages(List<Image> list,
                  Uri baseUri)
      {
```

```
   System.Web.HttpContext ctx = System.Web.HttpContext.Current;
   DirectoryInfo d = new DirectoryInfo(ImagePath);
   FileInfo[] files = d.GetFiles("*.jpg");

   foreach (FileInfo f in files)
   {
      string fileName = f.Name.Split(new char[] { '.' })[0];
      string etag = fileName + "_" + f.LastWriteTime.ToString();
      list.Add(GetImage(fileName, baseUri));
   }
}

[OperationContract]
[WebGet(UriTemplate="/images")]
public void Images()
{
   WebOperationContext wctx = WebOperationContext.Current;
   wctx.OutgoingResponse.ContentType = "text/html";
   wctx.OutgoingResponse.Headers.Add("Cache-Control", "no-cache");

   Uri baseUri = wctx.IncomingRequest.UriTemplateMatch.BaseUri;
   List<Image> listOfImages = new List<Image>();
   PopulateListOfImages(listOfImages, baseUri);

   TextWriter sw = new StringWriter();
   Html32TextWriter htmlWriter = new Html32TextWriter(sw);

   htmlWriter.WriteFullBeginTag("HTML");
   htmlWriter.WriteFullBeginTag("BODY");
   htmlWriter.WriteFullBeginTag("H1");
   htmlWriter.Write("Wallpaper");
   htmlWriter.WriteEndTag("H1");
   htmlWriter.WriteFullBeginTag("TABLE");
   htmlWriter.WriteFullBeginTag("TR");

   int i = 0;

   Image image;
   while (i < listOfImages.Count)
   {
      image = listOfImages[i];

      htmlWriter.WriteFullBeginTag("TD");
      htmlWriter.Write(image.Name);
      htmlWriter.WriteBreak();
      htmlWriter.WriteBeginTag("IMG");
      htmlWriter.WriteAttribute("SRC", image.Uri);
      htmlWriter.WriteAttribute("STYLE", "width:150px;height:150px");
      htmlWriter.WriteEndTag("IMG");
      htmlWriter.WriteEndTag("TD");

      if (((i+1) % 5) == 0)
      {
         htmlWriter.WriteEndTag("TR");
         htmlWriter.WriteFullBeginTag("TR");
```



```
      }
      i++;
   }
   htmlWriter.WriteEndTag("TR");
   htmlWriter.WriteEndTag("TABLE");
   htmlWriter.WriteEndTag("BODY");
   htmlWriter.WriteEndTag("HTML");

   System.Web.HttpContext ctx = System.Web.HttpContext.Current;
   ctx.Response.Write(sw.ToString());
}

[OperationContract]
[WebGet(UriTemplate = "/image/{name}")]
public void GetImage(string name)
{
   WebOperationContext wctx = WebOperationContext.Current;
   wctx.OutgoingResponse.ContentType = "image/jpeg";

   System.Web.HttpContext ctx = System.Web.HttpContext.Current;

   string fileName = null;
   byte[] fileBytes = null;
   try
   {
      fileName = string.Format(@"{0}\{1}.jpg", ImagePath, name);
      if (File.Exists(fileName))
      {
         using (FileStream f = File.OpenRead(fileName))
         {
            fileBytes = new byte[f.Length];
            f.Read(fileBytes, 0, Convert.ToInt32(f.Length));
         }
      }
      else
         wctx.OutgoingResponse.StatusCode = System.Net.HttpStatusCode.NotFound;
   }
   catch
   {
      wctx.OutgoingResponse.StatusCode = System.Net.HttpStatusCode.NotFound;
   }

   FileInfo fi = new FileInfo(fileName);
   wctx.OutgoingResponse.ETag = fileName + "_" +
            fi.LastWriteTime.ToString();
   ctx.Response.OutputStream.Write(fileBytes, 0, fileBytes.Length);
}
  }
}
```

Файл для конфигурирования службы `WallpaperService` приведен в листинге 13.13. Служба размещается с привязкой `webHttpBinding` и поведением оконечной точки `WebHttpBehavior`.

Листинг 13.13. Конфигурация службы WallpaperService

```
<system.serviceModel>
   <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
   <services>
      <service name="EssentialWCF.WallpaperService"
               behaviorConfiguration="MetadataBehavior">
         <endpoint address="" behaviorConfiguration="WebBehavior"
                   binding="webHttpBinding"
                   contract="EssentialWCF.WallpaperService"/>
         <endpoint address="mex"
                   binding="mexHttpBinding"
                   contract="IMetadataExchange"/>
      </service>
   </services>
   <behaviors>
      <endpointBehaviors>
         <behavior name="WebBehavior">
            <webHttp />
         </behavior>
      </endpointBehaviors>
      <serviceBehaviors>
         <behavior name="MetadataBehavior">
            <serviceMetadata httpGetEnabled="true" httpGetUrl="" />
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

В листинге 13.14 приведен SVC-файл для размещения службы в IIS.

Листинг 13.14. Файл WallpaperService.svc

```
<%@ ServiceHost Language="C#" Debug="true"
Service="EssentialWCF.WallpaperService"
➥CodeBehind="~/App_Code/WallpaperService.cs" %>
```

Размещение в Web

Пожалуй, одним из самых полезных усовершенствований WCF стала возможность размещать службы в Web. До выхода .NET Framework 3.5 для размещения служб надо было составлять конфигурационный файл или писать код. Так было даже в случае, когда служба размещалась внутри IIS. При размещении служб в Web это утомительно. WCF предлагает различные возможности размещения, но лишь малая часть использовалась разработчиками, создающими службы для Web. Например, AJAX-приложение вряд ли будет поддерживать несколько привязок, обеспечивать безопасность на уровне сообщений или требовать транзакций. Чтобы упростить размещение таких служб, в WCF был добавлен механизм Configuration Free Hosting – размещение без конфигурирования. С его помощью службу можно разместить вообще без конфигурационного файла или дополнительного программирования. Впрочем, необходимая для этого инфраструктура всегда была частью модели размещения WCF. Рассмотрим два способа воспользоваться предоставленной возможностью.

Класс WebScriptServiceHost

В пространство имен `System.ServiceModel.Web` добавлен новый класс `WebScriptServiceHost`. Он позволяет авторазмещать службы с привязкой `webHttpBinding` и поведением оконечной точки `WebScriptEnablingBehavior`. Его преимущество по сравнению с классом `ServiceHost` состоит в том, что ни привязки, ни поведения явно задавать не нужно.

Класс WebScriptServiceHostFactory

Еще один класс был добавлен в пространство имен `System.ServiceModel.Activation`, а именно `WebScriptServiceHostFactory`. Он предназначен для размещения в IIS с помощью SVC-файлов служб с привязкой `webHttpBinding` и поведением оконечной точки `WebScriptEnablingBehavior` без дополнительной конфигурации. В листинге 13.15 приведен пример SVC-файла, в котором использован класс `WebScriptServiceHostFactory`. Этот файл уже встречался нам при размещении службы `WallpaperService` (см. листинг 13.12). Достоинство такого подхода заключается в том, что для размещения службы не нужно задавать конфигурационную информацию, показанную в листинге 13.13.

Размещение без конфигурации для привязки WebHttp. Еще два новых класса – `WebServiceHost` и `WebServiceHostFactory` – предназначены для размещения служб с помощью привязки `webHttpBinding` и поведения оконечной точки `WebHttpBehavior`. Они позволяют обходиться без конфигурации точно так же, как классы `WebScriptServiceHost` и `WebScriptServiceHostFactory`.

Листинг 13.15. Файл WallpaperService.svc (без конфигурации)

```
<%@ ServiceHost Factory=
➥"System.ServiceModel.Activation.WebScriptServiceHostFactory"
➥Language="C#" Debug="true" Service="EssentialWCF.WallpaperService"
CodeBehind="~/App_Code/WallpaperService.cs" %>
```

Синдицирование контента с помощью RSS и ATOM

RSS и ATOM – это форматы синдицирования информации в Web. Они пригодны для синдицирования любого контента: новостей, видео, блогов и т.д. Чаще всего они применяются именно для блогов. С тех пор как технологии RSS и ATOM обрели популярность, их можно встретить на любом крупном сайте. WCF предлагает несколько механизмов для построения синдицированных каналов в форматах RSS и ATOM. В новое пространство имен `System.ServiceModel.Syndication` вошли классы для создания, потребления и форматирования Web-каналов. Основным является класс `SyndicationFeed`. В листинге 13.16 приведен пример приложения, в котором этот класс используется для создания канала

в любом из форматов RSS и ATOM. Приложение предоставляет информацию о фонотеке музыкальных записей в виде синдицированного канала.

Листинг 13.16. Синдицирование информации о фонотеке Zune Music

```
using System;
using System.IO;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Syndication;
using System.ServiceModel.Web;

[ServiceContract]
public class ZuneFeedService
{
   private static Uri LiveSearchBaseURI = new Uri("http://
search.live.com");
   private static UriTemplate LiveSearchTemplate =
               new UriTemplate(@"/results.aspx?q={terms}");

   private string MusicPath
   {
      get
      {
         return @"C:\Users\ricrane\Music\Zune";
      }
   }

   private SyndicationFeed ZuneFeed
   {
      get
      {
         SyndicationFeed feed = new SyndicationFeed()
         {
            Title = new TextSyndicationContent("My Zune Music Library"),
            Description = new TextSyndicationContent("My Zune Music Library")
         };

         DirectoryInfo di = new DirectoryInfo(MusicPath);
         DirectoryInfo[] artists = di.GetDirectories();

         List<SyndicationItem> items = new List<SyndicationItem>();

         foreach (DirectoryInfo artist in artists)
         {
            SyndicationItem item = new SyndicationItem()
            {
               Title = new TextSyndicationContent(
                           string.Format("Artist: {0}", artist.Name)),
               Summary = new TextSyndicationContent(artist.FullName),
               PublishDate = DateTime.Now,
               LastUpdatedTime = artist.LastAccessTime,
               Copyright = new TextSyndicationContent(@"Zune Library (c)")
            };
```



```
            Uri searchUri =
➥LiveSearchTemplate.BindByPosition(LiveSearchBaseURI, artist.Name);
            item.Links.Add(new SyndicationLink(searchUri));
            items.Add(item);
         }

         feed.Items = items;

         return feed;
      }
   }


   [OperationContract]
   [WebGet]
   [ServiceKnownType(typeof(Atom10FeedFormatter))]
   [ServiceKnownType(typeof(Rss20FeedFormatter))]
   public SyndicationFeedFormatter<SyndicationFeed>
               GetMusic(string format)
   {
      SyndicationFeedFormatter<SyndicationFeed> output;

      if (format == "rss")
            output = new Rss20FeedFormatter(ZuneFeed);
      else
            output = new Atom10FeedFormatter(ZuneFeed);

      return output;
   }
}
```

В листинге 13.17 приведен код для размещения службы синдицирования. Приложение авторазмещает ее с помощью класса `WebServiceHost`, а затем само же потребляет и выводит содержимое канала на экран.

Листинг 13.17. Консольное приложение для просмотра канала Zune Music

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.ServiceModel.Syndication;
using System.ServiceModel.Web;

namespace ZuneFeed
{
   class Program
   {
      static void Main(string[] args)
      {
         ServiceHost host = new ServiceHost(typeof(ZuneFeedService),
                        new Uri("http://localhost:8000/zune"));

         ServiceEndpoint atomEndpoint =
```

```
                host.AddServiceEndpoint(typeof(ZuneFeedService),
                        new WebHttpBinding(), "feed");
            atomEndpoint.Behaviors.Add(new WebHttpBehavior());

            host.Open();

            Console.WriteLine("Владелец службы открыт");

            SyndicationFeed feed = SyndicationFeed.Load(
                    new Uri("http://localhost:8000/zune/feed/?format=rss"));

            foreach (SyndicationItem item in feed.Items)
            {
               Console.WriteLine("Исполнитель: " + item.Title.Text);
               Console.WriteLine("Описание: " + item.Summary.Text);
            }

            Console.WriteLine("Для завершения нажмите [Enter].]");
            Console.ReadLine();
        }
    }
}
```

Резюме

Появившиеся в WCF новые средства для программирования Web-приложений упрощают их создание и помогают разработчику быстро написать службу, обеспечив нужный ему механизм ее потребления. Ниже перечислены основные возможности WCF, ориентированные на создание служб для Web:

- Каркас .NET Framework 3.5 предоставляет новый класс `UriTemplate` для эффективного разбора URI, включая путь и строку запроса. Он используется в модели программирования WCF для обращения к службам из Web.
- WCF может раскрывать информацию в разных форматах сериализации, включая SOAP и POX. В версии .NET Framework 3.5 добавлена поддержка сериализации в формате JSON.
- Новая привязка `webHttpBinding` позволяет раскрывать службы, следуя модели программирования в Web.
- Привязка `webHttpBinding` применяется с поведениями оконечной точки `WebHttpBehavior` и `WebScriptEnablingBehavior`. Поведение `WebHttpBehavior` раскрывает службу в форматах POX и JSON. Поведение `WebScriptEnablingBehavior` тоже сериализует данные в формате JSON, но еще обеспечивает поддержку генерации клиентских прокси-классов для ASP.NET AJAX.
- В WCF появился новый способ размещения служб в IIS без конфигурирования. Для этого предоставляются два готовых класса: `WebServiceHostFactory` и `WebScriptServiceHostFactory`. Класс `WebServiceHostFactory` поддерживает размещение без конфигурирования с привязкой `webHttpBinding` и поведением оконечной точки `WebHttpBehavior`. Класс `WebScriptServiceHostFactory` поддерживает размещение с привязкой `webHttpBinding` и поведением оконечной точки `WebScriptEnablingBehavior`.
- В версии .NET Framework 3.5 WCF поддерживает развитую модель программирования для синдицирования контента с помощью классов из пространства имен `System.ServiceModel.Syndication`. Класс `Rss20FeedFormatter` предназначен для вывода в формате RSS, а класс `Atom10FeedFormatter` — в формате ATOM.

Приложение Дополнительные вопросы

Для разработки приложений с использованием технологии Windows Communication Foundation (WCF) нужно знать многое. Хотя мы старались включить все, что необходимо обычному разработчику, кое-что осталось за кадром. Задача настоящего приложения – восполнить некоторые пробелы.

Публикация метаданных с помощью оконечных точек

В главе 1 мы уже обсуждали метаданные и познакомились с тем, как с их помощью раскрывается конфигурация служб. Но подробно о том, как раскрываются сами метаданные, мы не говорили. Позже, в главе 4 была введена идея привязок, но обсуждение ограничилось лишь привязками, предназначенными для раскрытия служб; о привязках для раскрытия метаданных не было сказано ни слова. В WCF имеются четыре дополнительных привязки: `mexHttpBinding`, `mexHttpsBinding`, `mexTcpBinding` и `mexNamedPipeBinding`. Они раскрывают сведения о конфигурации служб по различным транспортным протоколам.

Привязки с именами, начинающимися с «mex», служат для раскрытия метаданных службы. В WCF имена всех привязок, предназначенных для раскрытия метаданных, начинаются с префикса «mex».

Привязка mexHttpBinding

Привязка `mexHttpBinding` раскрывает метаданные по протоколу HTTP. Она годится в тех случаях, когда служба раскрывается с помощью привязок `basicHttpBinding`, `wsHttpBinding`, `ws2007HttpBinding` или заказных привязок, содержащих элемент `HttpTransportBindingElement`. Как правило, именно привязкой `mexHttpBinding` вы и будете пользоваться, так как она обеспечивает доступ к метаданным для наиболее широкого круга клиентов. В частности, данные о службе непосредственно доступны таким инструментам генерации клиентов, как утилита `svcutil.exe` и операция Add Service Reference в Visual Studio 2008. А также всем остальным клиентам, работающим по протоколу HTTP, включая браузеры Internet Explorer, FireFox и Opera. Иногда это нежелательно из соображений безопасности. А если служба раскрывается локально с помощью привязки `netNamedPipeBinding`, то пользоваться привязкой `mexHttpBinding` вряд ли имеет смысл.

Привязка mexNamedPipeBinding

Привязка `mexNamedPipeBinding` предназначена для раскрытия метаданных по протоколу Named Pipes (именованные каналы) и применяется совместно со службами, которые сами пользуются привязкой `netNamedPipeBinding` или заказными, содержащими элемент `NamedPipeTransportBindingElement`. Она не позволяет публиковать метаданные в сети, разрешая доступ к ним только локальному компьютеру. WCF сознательно ограничивает применение именованных каналов лишь для локальных коммуникаций. Этот вопрос обсуждался в разделе «Коммуникация между .NET-приложениями на одной машине» в главе 4.

Привязка mexTcpBinding

Привязка `mexTcpBinding` предназначена для раскрытия метаданных по протоколу TCP и применяется совместно со службами, которые сами пользуются привязками `netTcpBinding`, `netPeerTcpBinding` или заказными, содержащими элемент `TcpTransportBindingElement`. Когда привязка `mexTcpBinding` используется в режиме обобществления портов, возникает некая проблема. Элемент `TcpTransportBindingElement`, на котором основана привязка `mexTcpBinding`, запрещает обобществление портов. Если такой режим все же необходим, то можно создать заказную привязку на базе `mexTcpBinding`, установив для транспортного протокола свойство `PortSharingEnabled`. Подробнее см. раздел «Обобществление портов несколькими службами» ниже.

Привязка mexHttpsBinding

Привязка `mexHttpsBinding` раскрывает метаданные по протоколу HTTP с SSL/TLS-шифрованием (HTTPS). Как и `mexHttpBinding`, она предназначена для служб, которые раскрываются с помощью привязок `basicHttpBinding`, `wsHttpBinding`, `ws2007HttpBinding` или заказных, содержащих элемент `HttpTransportBindingElement`. Использование привязки `mexHttpsBinding` обеспечивает шифрование на транспортном уровне, следовательно, метаданные невозможно «подсмотреть» на стадии передачи по сети.

Создание клиентов на основе метаданных

Класс `MetadataResolver` позволяет извлекать информацию из метаданных программно. Следовательно, клиенты можно создавать динамически, без всяких конфигурационных файлов. Это бывает полезно, когда вы хотите развернуть клиентов, а позже изменяете конфигурацию службы. В листинге А.1 приведен пример использования класса `MetadataResolver`, которому указана известная оконечная точка метаданных. Для построения информации о привязке вызывается метод

Resolve. Сама информация содержится в одном или нескольких объектах класса ServiceEndpoint, по одному для каждой имеющейся оконечной точки. Далее эти объекты используются для создания клиента.

Листинг А.1. Использование класса MetadataResolver

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace EssentialWCF
{
   public partial class MainWindow : System.Windows.Window
   {
      /// <summary>
      /// Логика взаимодействия для MainWindow.xaml
      /// </summary>

      public MainWindow()
      {
         InitializeComponent();
      }

      void GetPriceButton_Click(object sender, RoutedEventArgs e)
      {
         Type typeOf = typeof(ISimpleStockService);
         Uri metadataUri =
            new Uri("http://localhost/SimpleStockService/Service.svc/mex");
         EndpointAddress metadataAddress = new EndpointAddress(metadataUri);
         ServiceEndpointCollection endpoints =
               MetadataResolver.Resolve(typeOf, metadataAddress);
         string symbol = SymbolTextBox.Text;
         decimal price;

         foreach (ServiceEndpoint point in endpoints)
         {
            if (point != null)
            {
               using (ChannelFactory<ISimpleStockService> cf =
➥new ChannelFactory<EssentialWCF.ISimpleStockService>(point))
               {
                  ISimpleStockService client = cf.CreateChannel();
                  price = client.GetQuote(symbol);

                  SymbolPricesListBox.Items.Insert(0,
```



```
                     symbol + "=" + price.ToString());
               }
            }
         }
      }
   }
}
```

Создание Silverlight-клиентов на основе метаданных

Silverlight – это разработанная Microsoft технология мультимедийных Web-приложений следующего поколения. Она позволяет создавать привлекательные интерактивные пользовательские интерфейсы, включающие анимацию, видео и графику. Одно из основных достоинств технологии Silverlight заключается в том, что она работает в разных браузерах – Internet Explorer, FireFox, Safari – и в разных операционных системах – Windows, Mac OS и Linux. В настоящее время доступны две версии Silverlight – v1.0 и v1.1.

Silverlight 1.0 – это самая первая версия, в которой основное внимание уделено обогащенному мультимедийному содержимому, включающему интерактивность, анимацию, видео и графику. Она поддерживает программирование на языке JavaScript, то есть для взаимодействия с пользовательским интерфейсом пишется сценарий на языке, уже встроенном в браузер. WCF позволяет обращаться из приложений Silverlight к службам, раскрывающим оконечные точки в стиле REST/POX, с помощью сценариев на основе технологии AJAX. Этот подход подробно описан в главе 13.

Silverlight 1.1 (или SL 1.1 alpha) в настоящее время имеет статус альфа-версии и продолжает изменяться, но она весьма перспективна, поэтому следует понимать, как в ней устроено взаимодействие с WCF. В состав SL 1.1 alpha включена миниверсия исполняющей среды .NET Common Language Runtime. Это позволяет программировать Silverlight-приложения на .NET-совместимых языках. Хотя от JavaScript тоже никто не отказывался, многие разработчики на платформе .NET захотят пользоваться знакомыми языками. Заодно это означает, что WCF может генерировать клиентские прокси-классы на языках .NET и ссылаться на них из программы. Такие прокси-классы генерирует новая утилита slwsdl.exe, являющаяся модернизированной версией утилиты wsdl.exe, давно входящей в состав .NET Framework. Создаваемые ей прокси-классы позволяют обращаться к Web-службам прямо из Silverlight. Ниже показана команда для генерации клиентского прокси-класса с помощью slwsdl.exe:

```
slwsdl.exe /silverlightClient http://localhost/Service.svc
```

Сгенерированный прокси – это класс .NET, наследующий одному из базовых классов в версии .NET Framework, включенной в Silverlight. Примерно так же разработчики генерируют клиентские прокси-классы для персональных приложений с поддержкой WCF, запуская программу svcutil.exe. Конечная цель состо-

ит в том, чтобы Silverlight-приложения с поддержкой WCF можно было разрабатывать так же, как приложения для настольного ПК.

Дополнительную информацию о технологии Silverlight можно найти на сайте Microsoft Silverlight по адресу www.silverlight.net, а также на странице проекта Moonlight по адресу www.mono-project.com/Moonlight. Moonlight – это реализация Silverlight для Linux с открытыми исходными текстами.

Обобществление портов несколькими службами

В версии Internet Information Services 6.0 (IIS) и более поздних имеется способ обобществления портов несколькими процессами. Делается это на уровне нового драйвера `http.sys`, реализующего протокол HTTP в ядре. Он управляет соединениями для IIS и WCF-служб с авторазмещением. Этот подход хорошо зарекомендовал себя для служб, работающих по протоколу HTTP, и не зависит от способа размещения. В состав WCF входит Windows-служба обобществления портов Net.Tcp Port Sharing Service, которая позволяет использовать общие порты для разных TCP-соединений. По умолчанию эта служба отключена. На рис. А.1 показано, как запустить ее из командной строки.

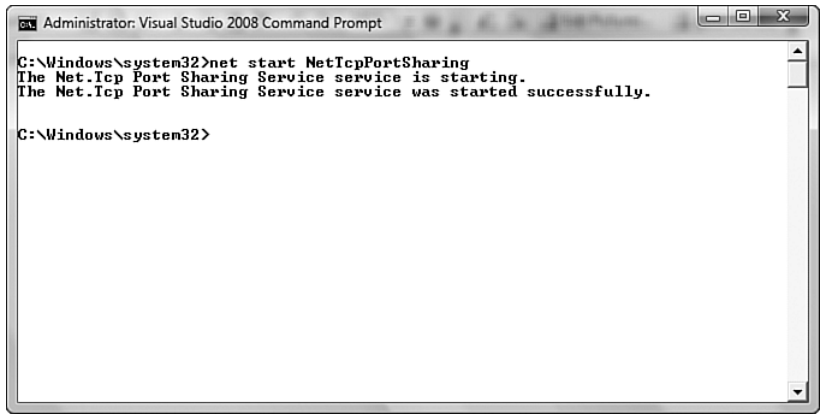

Рис. А.1. Запуск службы Net.Tcp Port Sharing Service из командной строки

После того как служба запущена, можно включать обобществление портов для привязки. В листинге А.2 показано, что для этого нужно написать в конфигурационном файле.

Листинг А.2. Включение режима обобществления портов в конфигурационном файле

```
<system.serviceModel>
   <bindings>
      <netTcpBinding>
         <binding name="NetTcpWithPortSharing"
                  portSharingEnabled="true" />
      </netTcpBinding>
   </bindings>
</system.serviceModel>
```


В листинге А.3 показано, как то же самое сделать программно.

Листинг А.3. Включение режима обобществления портов из программы

```
public void EnablePortSharing()
{
      NetTcpBinding b = new NetTcpBinding();
      b.PortSharingEnabled = true;
}
```

Конфигурирование квот для службы

Все продукты Microsoft поставляются «безопасными по умолчанию». Это относится и к WCF и означает, что параметры WCF заданы так, чтобы предотвратить различные атаки, например, имеющие целью вызвать отказ от обслуживания. Microsoft по умолчанию выставляет значения параметров в предположении, что разработка ведется на одной машине. Поэтому при передаче системы в промышленную эксплуатацию некоторые настройки придется изменить.

Одна из таких настроек – поведение `ServiceThrottlingBehavior`. Оно ограничивает объем потребляемых ресурсов путем задания квот для служб. Для этого поведения определены три параметра: `MaxConcurrentCalls`, `MaxConcurrentInstances` и `MaxConcurrentSessions`. В таблице А.1 все они описаны и приведены значения, подразумеваемые по умолчанию.

Таблица А.1. Свойства поведения ServiceThrottlingBehavior

Параметр	Описание	Значение по умолчанию
`MaxConcurrentCalls`	Ограничивает количество одновременно обрабатываемых обращений	16
`MaxConcurrentInstances`	Ограничивает количество одновременно установленных сеансов	10
`MaxConcurrentSessions`	Ограничивает количество одновременно работающих экземпляров службы	Int32.MaxValue

Принимаемые по умолчанию значения параметров `MaxConcurrentCalls` и `MaxConcurrentSessions` могут снизить пропускную способность в условиях промышленной эксплуатации. Если ваши серверы располагают достаточными ресурсами, чтобы справиться с нагрузкой, то при необходимости ограничения можно ослабить. Однако не забывайте о том, что тем самым вы открываете возможность для атак типа «отказ от обслуживания». В листинге А.4 показано, как изменить эти параметры в конфигурационном файле.

Листинг А.4. Изменение параметров поведения ServiceThrottlingBehavior в конфигурационном файле

```
<system.serviceModel>
   <behaviors>
      <serviceBehaviors>
         <behavior name="ServiceThrottlingBehavior">
            <serviceThrottling maxConcurrentCalls="1000"
                               maxConcurrentInstances="1000"
                               maxConcurrentSessions="1000" />
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

В листинге А.5 показано, как то же самое сделать программно.

Листинг А.5. Изменение параметров поведения ServiceThrottlingBehavior из программы

```
public void IncreaseThrottle(ServiceHost serviceHost)
{
   ServiceThrottlingBehavior throttleBehavior =
➥serviceHost.Description.Behaviors.Find<ServiceThrottlingBehavior>();

   if (throttleBehavior == null)
   {
      throttleBehavior = new ServiceThrottlingBehavior();
      serviceHost.Description.Behaviors.Add(throttleBehavior);
   }

   throttleBehavior.MaxConcurrentCalls = 4000;
   throttleBehavior.MaxConcurrentInstances = 4000;
   throttleBehavior.MaxConcurrentSessions = 4000;
}
```

Конфигурирование HTTP-соединений

В спецификации протокола HTTP 1.1 описан механизм поддержания HTTP-соединений – HTTP Keep-Alive. Он позволяет некоторое время сохранять TCP-соединение между клиентом и сервером, так чтобы клиент мог отправлять по нему последующие запросы. Чтобы ограничить потребление ресурсов сервера, в спецификации HTTP 1.1 оговаривается, что для каждого клиента одновременно может быть открыто не более двух соединений. По умолчанию HTTP-клиенты в .NET Framework включают механизм Keep-Alive, и WCF – не исключение.

Элемент привязки `HttpTransportBindingElement` пользуется классом `HttpWebRequest` из пространства имен `System.Net` для отправки HTTP-запросов. Он, в свою очередь, обращается к классам `ServicePointManager` и `ServicePoint` для управления HTTP-соединениями, в частности, временем их жизни. В настоящем разделе мы рассмотрим вопрос о том, как управлять HTTP-соединениями для достижения оптимальной производительности и масштабируемости WCF-служб.

Рециркуляция простаивающих соединений

Свойство `MaxIdleTime` класса `ServicePoint` задает время простаивания соединения перед закрытием. Для каждого нового экземпляра `ServicePoint` оно по умолчанию равно 100 секунд. Это значение определяет свойством `MaxServicePointIdleTime` класса `ServicePointManager`. Изменять его особенно полезно, когда нужно балансировать нагрузку на серверы в составе фермы. Чем меньше значение, тем больше вероятность, что соединение не будет простаивать зазря. Это позволит клиентам устанавливать новые соединения с другими серверами фермы. В листинге А.6 показано, как изменить значение свойства `MaxIdleTime` для всех экземпляров класса `ServicePoint`.

Листинг А.6. Задание свойства MaxIdleTime из программы

```
public void SetConnections()
{
   Uri myUri = new Uri("http://www.somewhere.com/");
   System.Net.ServicePoint sp =
ServicePointManager.FindServicePoint(myUri);
   sp.MaxIdleTime = 30000;
}
```

Изменение времени жизни соединения

Свойство `ConnectionLeaseTimeout` класса `ServicePoint` определяет, сколько времени соединение может оставаться активным до того, как его закроют. В каждом новом экземпляре `ServicePoint` оно по умолчанию равно -1, то есть соединение может оставаться открытым неопределенно долго. В фермах серверов с балансированием нагрузки это нежелательно, так как в этом случае соединение между клиентом и сервером никогда не разрывается по инициативе сервера. Если значение `ConnectionLeaseTimeout` положительно, то по истечении указанного времени соединение можно использовать для других целей. При поступлении каждого запроса проверяется величина `ConnectionLeaseTimeout`. Если время жизни соединения истекло, то активное соединение закрывается и создается новое. Чтобы соединение закрывалось после обработки каждого запроса, следует задать `ConnectionLeaseTimeout` равным 0. В листинге А.7 показано, как изменить значение `ConnectionLeaseTimeout` для всех экземпляров класса `ServicePoint`.

Листинг А.7. Задание свойства ConnectionLeaseTimeout из программы

```
public void SetConnections()
{
   Uri myUri = new Uri("http://www.somewhere.com/");
   System.Net.ServicePoint sp =
ServicePointManager.FindServicePoint(myUri);
   sp.ConnectionLeaseTimeout = new TimeSpan(0,0,30);
}
```

Отключение механизма HTTP Keep-Alive

Манипуляции со свойствами `MaxIdleTime` и `ConnectionLeaseTimeout` класса `ServicePoint` помогают управлять временем жизни соединения. Это особенно полезно для ферм серверов с балансированием нагрузки. К сожалению, не все балансировщики поддерживают механизм HTTP Keep-Alive. Иногда единственный способ добиться равномерной загрузки – отключить этот механизм вовсе. Сделать это можно разными способами.

Будет ли использоваться механизм HTTP Keep-Alive, зависит от многих настроек. Например, можно включить соответствующий режим в сервере Internet Information Services (IIS). На рис. А.2 показано окно HTTP Response Headers, появившееся в версии IIS 7.0 в системах Windows Server 2008 и Windows Vista с пакетом обновлений Service Pack 1.

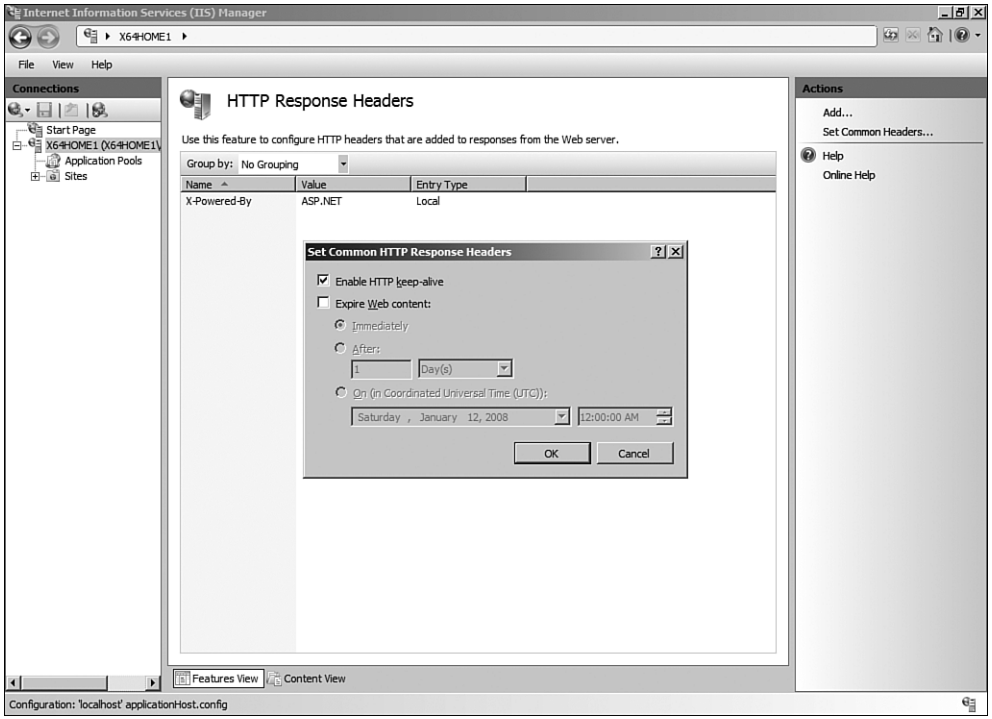

Рис. А.2. Настройка режима HTTP Keep-Alive в IIS 7.0

В IIS 7.0 настроить режим HTTP Keep-Alive можно также из командной строки. Ниже показана команда для отключения HTTP Keep-Alive в IIS 7.0. До выхода SP1 никакого другого способа сделать это в Window Vista не было.

```
appcmd set config /section:httpProtocol /allowKeepAlive:false
```


Управлять механизмом HTTP Keep-Alive можно также с помощью свойства `KeepAliveEnabled` элемента привязки `HttpTransportBindingElement`. В листинге А.8 показано, как отключить его на этом уровне.

Листинг А.8. Использование свойства KeepAliveEnabled элемента привязки HttpTransportBindingElement

```
public void SetHttpKeepAlives()
{
   HttpTransportBindingElement be = new HttpTransportBindingElement();
   be.KeepAliveEnabled = false;
}
```

Отключить HTTP Keep-Alive можно также в конфигурационном файле, если воспользоваться заказной привязкой на основе элемента `HttpTransportBindingElement`. Пример приведен в листинге А.9.

Листинг А.9. Задание атрибута keepAliveEnabled в элементе httpTransport

```
<system.serviceModel>
   <bindings>
      <customBinding>
         <binding name="CustomBindingWithoutKeepAlives">
            <httpTransport keepAliveEnabled="false" />
         </binding>
      </customBinding>
   </bindings>
</system.serviceModel>
```

И последний способ, о котором мы уже говорили, – задать свойство `ConnectionLeaseTimeout` класса `ServicePoint` равным 0. В этом случае соединение будет закрываться после обработки каждого запроса. Поэтому при каждом запросе клиент будет вынужден устанавливать новое соединение. Эффект тот же самый, что при отключении режима HTTP Keep-Alive.

Увеличение количества соединений

Свойство `ConnectionLimit` класса `ServicePoint` задает максимальное количество открытых соединений с данным экземпляром `ServicePoint`. Значение по умолчанию зависит от владельца службы. Если экземпляр `ServicePoint` создан на стороне клиента, то разрешено не более 2 соединений; если на стороне сервера в среде ASP.NET, то не более 10. Определяется это значением свойства `DefaultConnectionLimit` класса `ServicePointManager`. За счет увеличения `ConnectionLimit` можно повысить пропускную способность в случае межсерверных коммуникаций или многопоточных клиентов. В листинге А.10 показано, как увеличить максимальное количество соединений с помощью конфигурационного файла.

Листинг А.10. Задание атрибута maxConnection в конфигурационном файле

```
<system.net>
   <connectionManagement>
```

```
    <add address="http://www.somewhere.com/" maxconnection ="1000" />
  </connectionManagement>
</system.net>
```

Конфигурирование TCP-соединений

В отличие от HTTP-соединений, для управления временем жизни TCP-соединений WCF не пользуется имеющимися в .NET Framework классами, а организует пул соединений. Для управления пулом служит свойство `ConnectionPoolSettings` элемента привязки `TcpTransportBindingElement`. Оно возвращает экземпляр класса `TcpConnectionPoolSettings`, в котором есть три свойства: `IdleTimeout`, `LeaseTimeout` и `MaxOutboundConnectionsPerEndpoint`.

Рециркуляция простаивающих соединений

Свойство `IdleTimeout` определяет, сколько времени соединение может простаивать, находясь в пуле, пока не будет закрыто и удалено из него. По умолчанию это две минуты. Задание меньшего значения может положительно сказаться на балансировании нагрузки, так как увеличивается вероятность, что соединение не будет простаивать долго.

Изменение времени жизни соединения

Свойство `LeaseTimeout` определяет, сколько времени соединение может оставаться активным до того, как его закроют. Если время жизни соединения истекло, оно может быть закрыто и удалено из пула. По умолчанию время жизни – пять минут. В случае, когда применяется балансирование нагрузки, имеет смысл уменьшить значение.

Увеличение количества соединений

Свойство `MaxOutboundConnectionsPerEndpoint` определяет максимальное количество соединений в пуле. По умолчанию оно равно 10. За счет увеличения можно повысить производительность и масштабируемость в случае межсерверных коммуникаций или многопоточных клиентов.

LINQ и WCF

Новая технология интегрированного языка запросов (Language Integrated Query – LINQ), включенная в .NET Framework 3.5, позволяет делать запросы к данным из языка C# или Visual Basic .NET. Традиционно запросы формулировались в виде строк на таких языках, как SQL или XPath, поэтому не могли быть проверены на этапе компиляции. Средства IntelliSense им тоже были недоступны. LINQ позволяет сделать запросы полноценными языковыми конструкциями. LINQ поддерживает несколько источников данных, включая базы данных SQL Server, XML-документы, наборы данных ADO.NET DataSet и объекты .NET. При

этом предоставляется единый механизм формулирования и выполнения запросов к разнородным источникам. LINQ перебрасывает мост между миром данных и миром объектов.

Представление реляционных данных с помощью LINQ-to-SQL

Термин LINQ-to-SQL относится к средствам LINQ, позволяющим представить реляционные данные, хранящиеся в базе SQL Server, в виде объектов. Это очень удобно, когда нужно отобразить сущности, хранящиеся в таблицах базы данных, на объекты приложения. Часто такой подход называют *объектно-реляционным отображением* (object-relational mapping – ORM). Чтобы упростить процедуру отображения, в Visual Studio 2008 имеется визуальный объектно-реляционный конструктор – Object Relational Designer, позволяющий отображать сущности на объекты. Его внешний вид показан на рис. А.3.

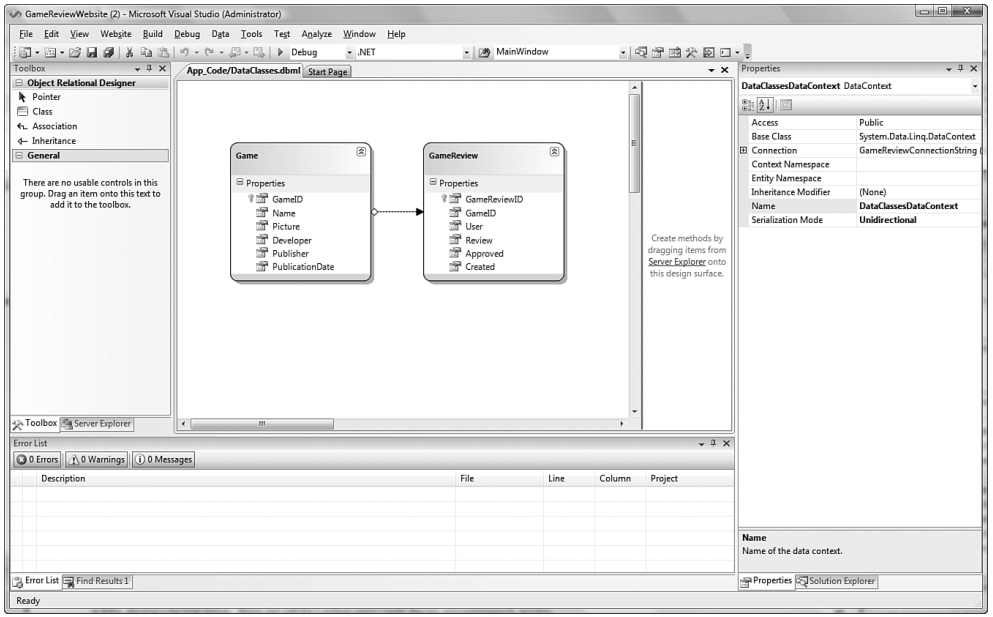

Рис. А.3. Объектно-реляционный конструктор LINQ-to-SQL

Следует знать, что по умолчанию конструктор не раскрывает сущности LINQ-to-SQL с помощью сериализатора `DataContractSerializer`. Следовательно, их нельзя раскрыть через WCF-службы. Однако возможность раскрытия хранящихся в базе данных с помощью службы важна при разработке сервисно-ориентированных приложений. К счастью, у поверхности конструирования

LINQ-to-SQL имеется свойство `Serialization Mode`. Если задать для него значение `UniDirectional`, то сущности LINQ-to-SQL можно будет снабдить атрибутами `[DataContract]` и `[DataMember]`, а, стало быть, они станут частью контракта о данных, сериализуемого средствами WCF. То же самое можно сделать с помощью командной утилиты `SqlMetal.exe`, которая позволяет генерировать код сущностей LINQ-to-SQL по описанию базы данных SQL Server. Флаг `/serialization:Unidirectional` говорит `SqlMetal.exe`, что она должна генерировать типы, сериализуемые средствами WCF.

Алфавитный указатель

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ш

Э

Я

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@alians-kniga.ru**.

Стив Резник, Ричард Крейн, Крис Боуэн

Основы Windows Communication Foundation для .NET Framework 3.5

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод *Слинкин А. А.*
Корректор Синяева Г. И.
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 13.05.2008. Формат 70×100 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 45. Тираж 1000 экз.
№
Издательство ДМК Пресс
Web-сайт издательства: www.dmk-press.ru
Internet-магазин: www.alians-kniga.ru