Л. Е. Карпов

Архитектура распределенных систем программного обеспечения

Настоящее учебное пособие издано при поддержке образовательной программы "Формирование системы инновационного образования в МГУ".

ISBN 978-5-84907-304-0
ISBN 978-5-317-02113-9

Шифр в библиотеке МГУ: 5ВГ66, К-265

Доступ через интернет к электронному каталогу библиотеки:
http://search.nbmgu.ru/resurs.jsp?f=1016

Москва, МАКС Пресс, 2007

Москва, МГУ им. М. В. Ломоносова, 2007

Оглавление

1. Введение в распределенные системы программного обеспечения [1]

1.1. Основные свойства распределенных систем

Большая часть проблем, которые решаются использованием распределенных систем программного обеспечения, так же, как и ограничений, с которыми это использование постоянно сталкивается, могут быть поняты при рассмотрении процесса эволюционного развития методов распределенной обработки информации. Необходимо осознавать, что, несмотря на то, что сама технология работы программистов и пользователей их программ изменилась, проблемы остаются теми же самыми, что были и в прошлом. Изучение различных методов программной поддержки работы распределенных систем будет вестись в некотором хронологическом порядке от первых систем, основанных непосредственно на методе удаленного вызова процедур, до наиболее современных форм системной поддержки – сетевых служб.

Распределенная система – это набор независимых компьютеров, представляющихся их пользователям единой объединенной системой (определение вольное, но пригодное).

От пользователей скрыты различия между компьютерами и способы связи между ними (от пользователя скрыто даже то, что компьютер, вообще, может быть всего один, во всяком случае компьютеры распределенной системы автономны). Пользователи и приложения единообразно работают в распределенных системах, независимо от того, где и когда происходит их взаимодействие. Вычислительная система, состоящая из множества различных вычислительных машин, на которых установлено самое разное программное обеспечение, может называться распределенной системой только в том случае, если для своих пользователей она выглядит и ведет себя как классическая однопроцессорная система с разделением времени. Чтобы поддерживать представление различных компьютеров и вычислительных сетей в виде единой системы, организация распределенных систем часто включает в себя дополнительный уровень программного обеспечения. Этот уровень называется уровнем системной поддержки (*middleware*).

Основная задача распределенных систем программного обеспечения – облегчить их пользователям доступ к удаленным ресурсам, а также контролировать совместное использование этих ресурсов. Ресурсы могут быть виртуальными, но могут быть и традиционными – компьютерами, принтерами, устройствами хранения файлов, файлами и данными.

---

[1] *Настоящее учебное пособие издано при поддержке образовательной программы "Формирование системы инновационного образования в МГУ".*

1.2. Основные требования к распределенным системам

В качестве основного требования к распределенным системам предъявляется достижение их прозрачности, открытости и масштабируемости.

1.2.1. Прозрачность

Распределенная система должна скрывать разницу в способах представления данных и в способах доступа к ресурсам. Такое свойство распределенных систем называется прозрачностью доступа к данным.

Распределенная система должна обеспечивать прозрачность местоположения ресурса, то есть скрывать его физическое расположение. Важно, чтобы ресурсы имели только логические имена. Примером такого имени может служить универсальный указатель ресурса URL, в котором нет никакой информации о том, где находится файл, который ищется в Интернете.

Ресурс может время от времени менять свое расположение, и при следующем вызове может быть обнаружен в другом месте (но по тому же логическому адресу). Распределенная система, позволяющая ресурсам менять свое расположение от вызова к вызову, обладает свойством прозрачности смены местоположения ресурса (пример – система ICQ).

Иногда ресурсу было позволено менять свое положение непосредственно в процессе его использования (пример такого ресурса – мобильные пользователи с беспроводной связью, не отключающиеся от сети при переходе в другую зону обслуживания). Это более сильное свойство называется прозрачностью динамической смены местоположения ресурса.

Для балансировки использования ресурсов они могут быть реплицированы, то есть, размножены по нескольким физическим адресам. Прозрачность репликации скрывает это. Из наличия этого свойства сразу следует и прозрачность местоположения.

Часто совместное использование ресурсов достигается за счет действительно совместной работы и тесного взаимодействия пользователей системы. Однако пользователь распределенной системы не должен знать, что он является не единственным ее пользователем. Например, при работе с системой управления базой данных (СУБД) пославший запрос пользователь не должен знать, что одновременно СУБД получает и обрабатывает запросы многих других пользователей. Такое прозрачное параллельное использование должно быть непротиворечивым, для чего составляются специальные правила блокировок, когда

пользователи поочередно получают исключительные права на ресурс. Этой же цели достигают с помощью отправки транзакций.

В распределенной системе должны быть приняты меры, чтобы часть аппаратуры брала на себя выполнение работы в случае выхода из строя другой части системы. Основная трудность достижения прозрачности поломки в том, чтобы отличить по-настоящему сломанные части от медленно работающих фрагментов системы.

Данные могут размещаться на различных физических носителях, в том числе таких, которые могут сохранять их значения в период выключения системы. Действия системы, обладающей свойством прозрачности сохранности данных, при обработке таких объектов должны быть скрыты от пользователя. Сохранность важна для любых информационных систем, не только распределенных.

Существуют ситуации, когда полностью скрыть распределенность (то есть достичь абсолютной прозрачности) не удается. При сильной удаленности узлов системы друга от друга возникают заметные задержки передачи информации. Существует проблема часовых поясов. Существует связь между прозрачностью и производительностью распределенной системы. Необходимо соблюдать баланс этих системных свойств.

1.2.2. Открытость

Открытость – это использование синтаксических и семантических правил, основанных на стандартах. Для распределенной системы – это, прежде всего, использование формализованных протоколов. Службы, входящие в распределенную систему, определяются через интерфейсы, которые часто описываются при помощи языков описания интерфейсов IDL. Языки IDL касаются почти исключительно синтаксиса (имена доступных функций, типы параметров, возвращаемых значений, исключительные ситуации). Семантика чаще задается неформально, на естественном языке.

Если интерфейс описан правильно, возникает возможность правильной совместной работы одного произвольного процесса, нуждающегося в интерфейсе, с другим произвольным процессом, представляющим интерфейс. Один и тот же интерфейс может быть также реализован в разных распределенных системах (независимо друг от друга), но работать обе системы будут одинаково. Для обеспечения переносимости и способности к взаимодействию в интерфейсе должно быть все, что нужно для его реализации, но он не должен определять внешний вид реализации. Переносимость характеризует, насколько приложение, сделанное для одной распределенной системы, может

работать в составе другой системы, а способность к взаимодействию показывает, насколько две реализации систем или компонентов, выполненных разными людьми, в состоянии работать совместно.

Открытые системы обладают очень важной характеристикой – гибкостью. Гибкость есть легкость конфигурирования системы, состоящей из различных компонентов. Достижения необходимого уровня гибкости приводит к тому, что открытая распределенная система становится расширяемой.

1.2.3. Масштабируемость

Распределенные системы программного обеспечения обладают свойством масштабируемости, которая может проявляться по отношению к размеру, к географическому положению, к административному устройству систем. Достижение масштабируемости связано с решением проблем, возникающих из-за наличия узких мест по обслуживанию (один сервер для множества клиентов), данным (множественный доступ к одному файлу данных), и алгоритмам (перегрузка коммуникаций из-за использования централизованных алгоритмов).

Требование масштабируемости часто является препятствием распространения систем, реализованных для локальных сетей, на уровень сетей глобальных (корпоративных или Интернета). В глобальных сетях время получения ответа может значительно превышать локальные задержки, поэтому там чаще используется асинхронная связь. Кроме того, в локальных сетях службы часто распределены по компьютерам фиксированно, а в глобальных сетях местоположение необходимой службы заранее неизвестно.

Еще одно следствие масштабируемости: аппаратные решения для распределенных систем могут быть гетерогенными. В отличие от гомогенных систем, построенных на единой технологии, гетерогенные системы могут состоять из частей, построенных на разных физических принципах, и обладающих разными свойствами и характеристиками.

Разнородность в сочетании с независимостью приводят к важнейшему свойству распределенных систем: сами системы могут существовать продолжительное время, но отдельные их части могут время от времени отключаться. При этом пользователи и приложения не должны уведомляться о том, что эти части вновь подключены, что добавлены новые части для поддержки дополнительных пользователей или приложений.

1.3. Логические программные слои распределенных систем

Концептуально распределенные системы строятся послойно: *презентационный* слой, слой *прикладной логики* и слой *управления ресурсами*. Эти слои могут быть только абстракциями, существуя лишь на этапе проектирования, но могут быть четко видимы в программном обеспечении, когда их реализуют в виде отдельных подсистем, иногда с применением различного инструментария.

Презентационный слой. Все распределенные системы должны общаться с внешним миром, с пользователями-программистами или другими программными системами. Значительная часть этого общения связана с преобразованием информации и представлением ее для внешних пользователей, подготовкой запросов и получением ответов. Компоненты распределенной системы, обеспечивающие эту деятельность, формируют презентационный слой.

Часто этот слой называют *клиентом* распределенной системы, что неверно. Все распределенные системы имеют клиентов, то есть сущности, которые пользуются сервисами, предоставляемыми этими системами. Клиенты могут быть полностью внешними по отношению к системам и независимыми от них. В этом случае они не являются презентационным слоем самих систем. Лучшими примерами программ, спроектированных таким образом, являются сетевые навигаторы, обрабатывающие документы, написанные на языке HTML. Презентационным слоем распределенной системы в данном случае будет сетевой сервер, а также модули, занятые созданием HTML-документов.

Случается, что клиент и презентационный слой слиты воедино. Это типично для систем типа *клиент/сервер*, в которых имеется программа, одновременно исполняющая роль клиента и презентационного слоя.

Слой прикладной логики. Любые системы программного обеспечения не только демонстрируют информацию, но и осуществляют обработку данных. Эта обработка производится программой, реализующей фактические операции, запрошенные клиентом через посредство презентационного слоя. Такая программа называется программой слоя прикладной логики. Иногда эти программы называются *службами*, предлагаемыми распределенными системами. В зависимости от сложности выполняемой логики этот слой может называться *бизнес процессом*, *бизнес логикой*, *бизнес правилами* или просто *сервером*.

Слой управления ресурсами. Для работы любая система программного обеспечения нуждается в данных. Данные могут размещаться в базах данных, файловых системах, в других репозиториях.

Программы слоя управления ресурсами объединяют все такие элементы. Иногда, чтобы показать, что этот слой реализуется с использованием системы управления базой данных, этот слой называют *слоем данных*. Этот подход, однако, имеет ограничения, поскольку он концентрируется только на аспекте управления данными. Однако в управлении нуждаются все внешние системы, поставляющие информацию. Сюда входят не только базы данных, но и другие распределенные системы со своими слоями презентационным, прикладным и управления ресурсами. При этом появляется возможность рекурсивно строить распределенные системы, состоящие из других систем, как из компонентов.

Описанные три слоя – это концептуальные конструкции, которые логически разделяют функциональность большинства распределенных систем. В практических реализациях они могут комбинироваться различными способами. В этих случаях говорят не о концептуальных слоях, а о ярусах (звеньях). Известны 4 типа основных типа распределенных систем, отличающихся количеством входящих в них ярусов: одно-, двух-, трех- и многоярусные системы.

1.4. Виды распределенных систем программного обеспечения

Одноярусные архитектуры возникли под влиянием архитектуры первых вычислительных систем, которая включала в себя большой вычислительный модуль и некоторое количество периферийных устройств для ввода и вывода информации, подготовленной при вычислениях.

Системы неизбежно получались монолитными, то есть все три слоя (презентационный, прикладной и ресурсный) являлись частями одной программы (располагались на одном ярусе), а в качестве клиента выступал простой терминал, имевший только клавиатуру для ввода команд оператора и экран для отображения ответов от вычислителя.

Спроектированы одноярусные системы так, что у них нет никаких точек входа, кроме канала связи с терминалом. В случае необходимости связать их с другими системами нужно применять имитаторы терминалов, осуществляя обмен экранами.

Но есть у одноярусных систем и достоинства. Во-первых, объединение слоев может проводиться для повышения эффективности. Обычно в таких системах минимизированы накладные расходы на переключение контекста и переходы между компонентами, а также на сложные преобразования данных. Не требуется разрабатывать специальную клиентскую часть, что снижает затраты на внедрение и поддержку.

Сопровождать одноярусные системы трудно и дорого. В настоящее время есть все возможности разрабатывать одноярусные системы, но общее развитие науки идет в противоположном направлении.

Двухъярусные архитектуры появились после возникновения персональных ЭВМ, на которых стало возможно не просто отображать информацию, но и обрабатывать ее. Роль сервера по-прежнему играла большая ЭВМ, но презентационный слой больше не требовалось объединять с другими слоями системы. У такого подхода есть два важных преимущества: использование вычислительных мощностей персональных ЭВМ (снятие нагрузки с других слоев), и использование одного презентационного слоя для разных целей, без усложнения системы.

Двухъярусные архитектуры стали весьма популярны в виде архитектур *клиент/сервер*. Часто под термином *клиент* понимают презентационный слой и собственно клиентское программное обеспечение, называя сервером объединение прикладной логики и управления ресурсами. Клиент может принимать различные формы, он может даже выполнять некоторую функциональность, которая в противном случае была бы на сервере. В зависимости от сложности клиентской программы ее называют *тонким клиентом*, если она выполняет минимум функциональности, либо *толстым клиентом*, если эта функциональность достаточно развита.

Разработка архитектуры клиент/сервер привела к появлению множества полезных механизмов, применяющихся в настоящее время в распределенных системах. Ярким примером является *удаленный вызов процедуры*. Еще одним полезным новшеством явились *прикладные программные интерфейсы*, существенно повлиявшие на методологию разработки распределенных систем. Прикладной программный интерфейс определяет, как надо обращаться к службе, какие можно ожидать ответы или изменения внутреннего состояния сервера, происходящие в результате обращения к нему. Если сервер имеет хорошо определенный, стабильный прикладной программный интерфейс, для него можно разработать разные виды клиентов и производить модернизацию сервера без какого-либо изменения в клиентах.

Двухъярусные системы имеют много преимуществ перед одноярусными. Объединение прикладной логики и управления ресурсами позволяет выполнять важнейшие вычисления очень быстро, поскольку переключения контекста не происходит. Двухъярусные системы гораздо более мобильны: сервер в них отделен от презентационного слоя.

Одной из основных проблем двухъярусных систем является ограниченность возможностей сервера по связям со многими клиентами одновременно. Двухъярусные архитектуры не справились с требованиями локальных информационных сетей. С их помощью клиенты могли общаться только со своими серверами, не будучи в силах взаимодействовать с другими.

Трехъярусные архитектуры сложнее и разнообразнее архитектур клиент/сервер. Все слои в них четко разделены (*Рис. 1.1*). Презентационный слой размещается в клиенте, как в двухъярусной архитектуре. Прикладная логика размещается в среднем ярусе и называется слоем системной поддержки или промежуточным слоем программного обеспечения (*middleware*). Слой управления ресурсами располагается на третьем ярусе и состоит из всех серверов, которые интегрируются в архитектурном решении. С точки зрения подсистемы управления ресурсами программы, работающие в слое прикладной логики, это просто клиенты (*Рис. 1.2*).

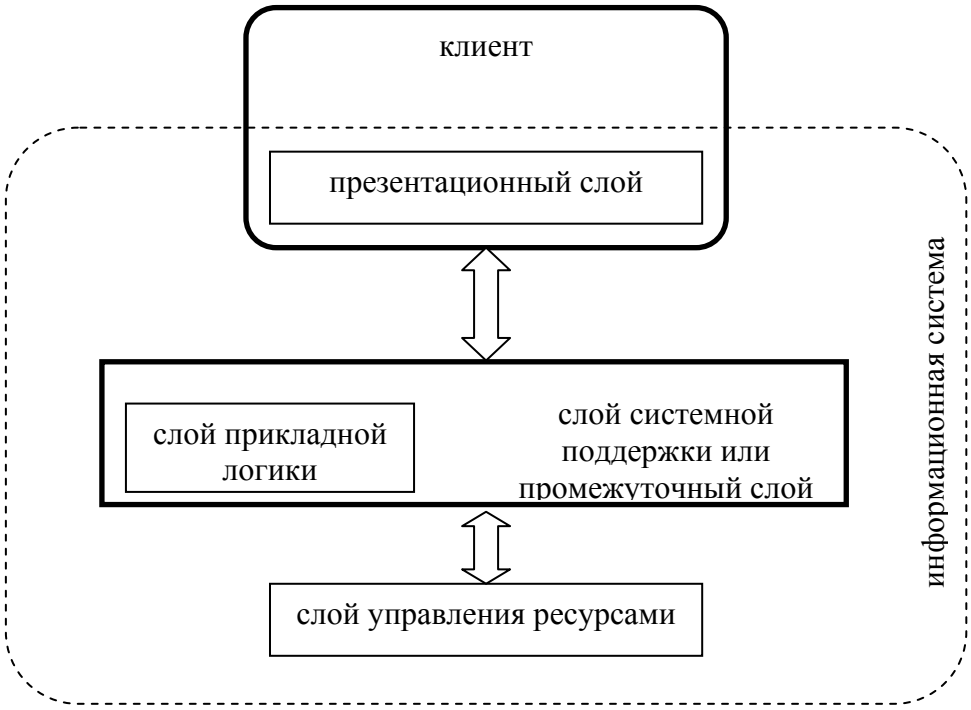


Рис. 1.1. В трехъярусной архитектуре между презентационным слоем и слоем управления ресурсов представлен промежуточный слой системной поддержки (middleware).

Хотя трехъярусные архитектуры разрабатывались преимущественно для интеграции, их можно использовать точно так же, как и двухъярусные. Их преимуществом при этом будут возросшие возможности по масштабированию. Каждый слой может работать на отдельной ЭВМ. В частности, прикладной слой может быть распределен по разным

компьютерам. Прикладная логика при этом может быть сделана существенно более независимой от управления ресурсами, ее переносимость и переиспользуемость существенно возрастает.

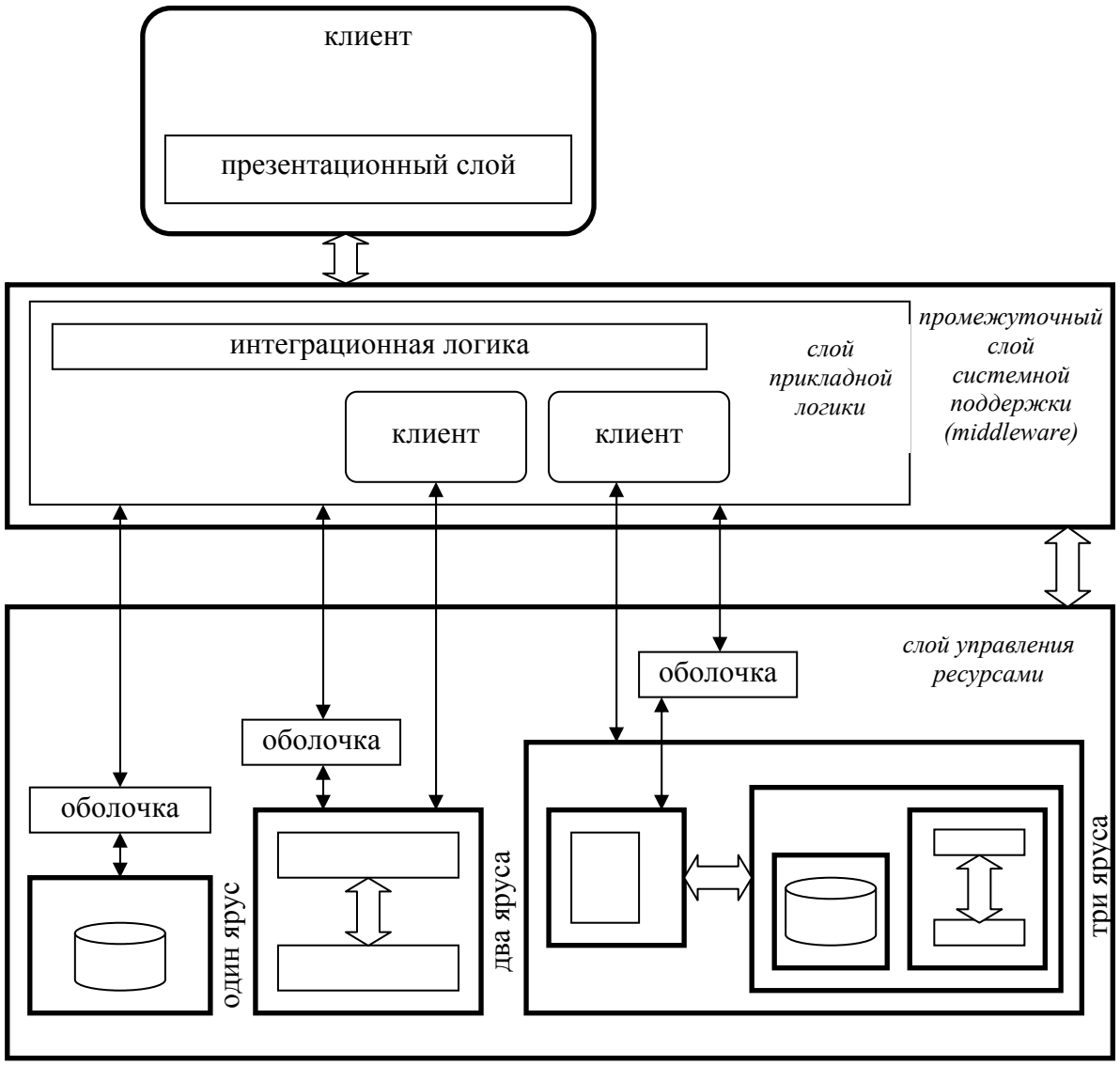


Рис. 1.2. Интеграция систем различной архитектуры с использованием трехъярусного подхода.

Трехъярусные системы дополнили и развили концептуально важные понятия, выдвинутые двухъярусными системами. Стало еще очевиднее, что управление ресурсами должно подчиняться четким интерфейсам, которыми должны пользоваться программы прикладной логики, находящиеся в промежуточном слое. Если двухъярусные архитектуры потребовали определения интерфейсов прикладного слоя, то трехъярусные привели к стандартизации интерфейсов слоя управления ресурсами.

Особенно четко преимущества трехъярусной архитектуры проявляются при интеграции разнородных ресурсов. Современное программное обеспечение промежуточных слоев включает в себя

функциональность, необходимую для введения в эти слои дополнительных свойств: транзакционных гарантий для различных видов ресурсов, балансировки загрузки оборудования, возможностей по регистрации событий, репликации, сохранности данных и многого другого. Используя системную поддержку, разработчики прикладной логики могут при создании сложнейших моделей взаимодействия пользоваться поддержкой, обеспечиваемой промежуточным слоем, а не программировать все самостоятельно. Потери в производительности компенсируются распространением модели промежуточного слоя на разные сетевые узлы, что существенно влияет на масштабируемость и надежность систем.

Ограниченность модели трехъярусных систем проявилась при попытках интегрировать несколько трехъярусных систем, а также при выходе распределенных систем на уровень Интернета, что связано недостаточной стандартизацией этих систем.

Многоярусные архитектуры не сильно отличаются от трехъярусных: это обобщение трехъярусной модели с учетом важности доступа к данным через Интернет. Многоярусные архитектуры разрабатываются для двух основных применений: объединение разнородных систем и подключение к Интернету. Отдельные слои многоярусных систем сами представляют собой двух- или трехъярусные системы.

Большинство современных систем построено по принципу многоярусности. Их создание потребовало больших усилий по интеграции других систем. В этом проявляется основной недостаток многоярусных систем – в них слишком много промежуточных слоев, часто с избыточной функциональностью, сложных, дорогих в разработке, регулировке и поддержке. Хотя с введением в систему новых ярусов достигается повышение ее гибкости, растет функциональность, но одновременно возрастает стоимость взаимодействия между ярусами, возникают проблемы с производительностью системы.

1.5. Способы взаимодействия в распределенных системах

Основной характеристикой способа взаимодействия подсистем распределенной системы является его *синхронность* или *асинхронность*. По отношению к исследуемым системам точнее говорить о *блокирующем* или *неблокирующем* взаимодействии. Блокирующим называется взаимодействие, если вовлеченные стороны прежде, чем переходить к следующим работам, должны дожидаться окончания взаимодействия. Следует понимать, что параллельная работа фрагментов систем не имеет никакого отношения к асинхронности и неблокирующему

взаимодействию. Сервер, получив запрос от одного из своих клиентов, может, обрабатывая запрос, работать параллельно другим своим клиентам, синхронность же относится к работе запрашивающего обслуживания клиента и того процесса в сервере, который этот запрос обрабатывает. Если работа клиента на время обработки запроса сервером приостанавливается – это *синхронное взаимодействие*. Если вместо блокировки клиент после выдачи запроса выполняет другие действия – это *взаимодействие асинхронное*.

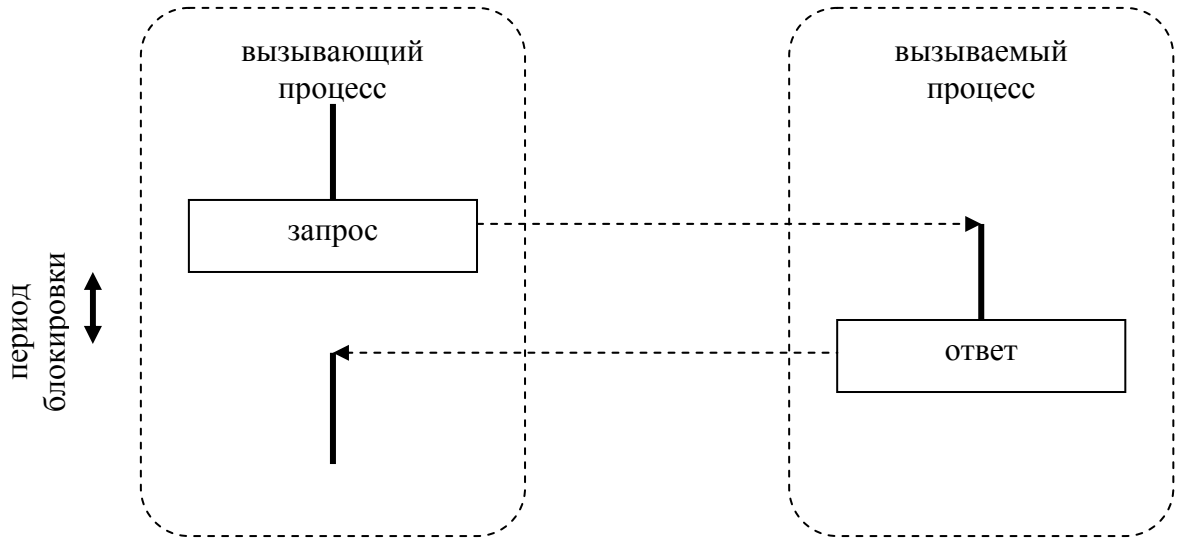


Рис. 1.3. Синхронный вызов требует заблокировать вызывающий процесс до получения ответа.

Синхронное взаимодействие (*Рис. 1.3*) обладает преимуществом простоты. Обрабатывая запрос, сервер может быть уверен, что состояние клиента во время этой обработки не изменится. В результате синхронное взаимодействие применяется в подавляющем большинстве систем промежуточного слоя. Эти преимущества являются одновременно и недостатками. Синхронное взаимодействие приводит к существенным потерям времени, а значит и производительности. Ожидающий процесс может, вообще, быть удален из оперативной памяти, что приведет к еще большим потерям времени.

Однако не всегда требуется работать синхронно. Один из примеров асинхронной связи – электронная почта. Асинхронные распределенные системы строятся аналогичным образом. Вместо вызова и ожидания ответа выполняется отправка сообщения (*Рис. 1.4*), а через некоторое время программа может проверить, прибыл ли ответ. Это позволяет программе выполнять другие работы и делает ненужной какую-либо координацию между сторонами взаимодействия.

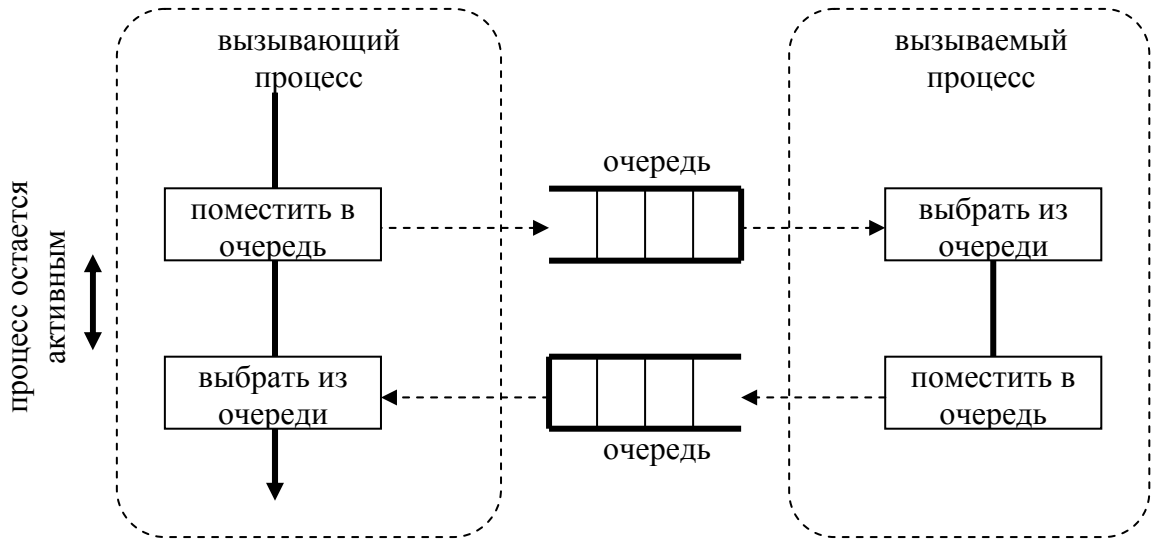


Рис. 1.4. Асинхронный вызов позволяет вызывающему процессу продолжить работу, пока его запрос обрабатывается.

Для асинхронного взаимодействия сообщения должны запоминаться в некотором промежуточном месте, откуда они впоследствии могут извлекаться сервером. Эта промежуточная память открывает возможности для новой функциональности, которую больше не требуется делать частью отдельных компонентов. Такие системы очередей теперь превратились в брокеры сообщений, которые фильтруют их и управляют потоком сообщений. Тем самым появляется возможность менять способы фильтрации, преобразования и доставки сообщений без изменения самих компонентов, генерирующих и получающих эти сообщения.

Уже рассмотренная система электронной почты представляет собой пример *сохранной почты*. Сообщения не пропадают, если компьютер пользователя выключен, а хранятся в коммуникационной системе до тех пор, пока его не удастся передать получателю. Кроме сохранной связи существует *связь без сохранения сообщений*. В таких системах сообщения сохраняются только в период работы приложений, которые их отправляют и получают.

На практике применяются различные комбинации этих типов взаимодействия. В случае *сохранной асинхронной связи* (*Рис. 1.5-а*) сообщение сохраняется в буфере сервера. Именно этот вид связи используется в системах электронной почты. В случае *сохранной синхронной связи* (*Рис. 1.5-б*) сообщения хранятся только на принимающем прикладном комплексе. Отправитель блокируется на все время, пока сообщение не попадет в этот буфер. Усеченный вариант сохранной синхронной связи заключается в том, что блокировка отправителя осуществляется до доставки сообщения серверу получателя.

Рис. 1.5. Сохранная асинхронная и сохранная синхронная связь между подсистемами А и Б.

При *асинхронной несохранной связи* (*Рис. 1.6-а*) приложение отправляет сообщение, временно сохраняемое в локальном буфере передающего комплекса, а отправитель продолжает свою работу. Параллельно коммуникационная система направляет сообщение в точку, из которой оно может достигнуть места назначения, возможно с сохранением в локальном буфере. Если получатель в момент прихода сообщения на принимающий комплекс этого получателя не активен, передача обрывается. Другой пример этого вида связи – асинхронный удаленный вызов процедуры.

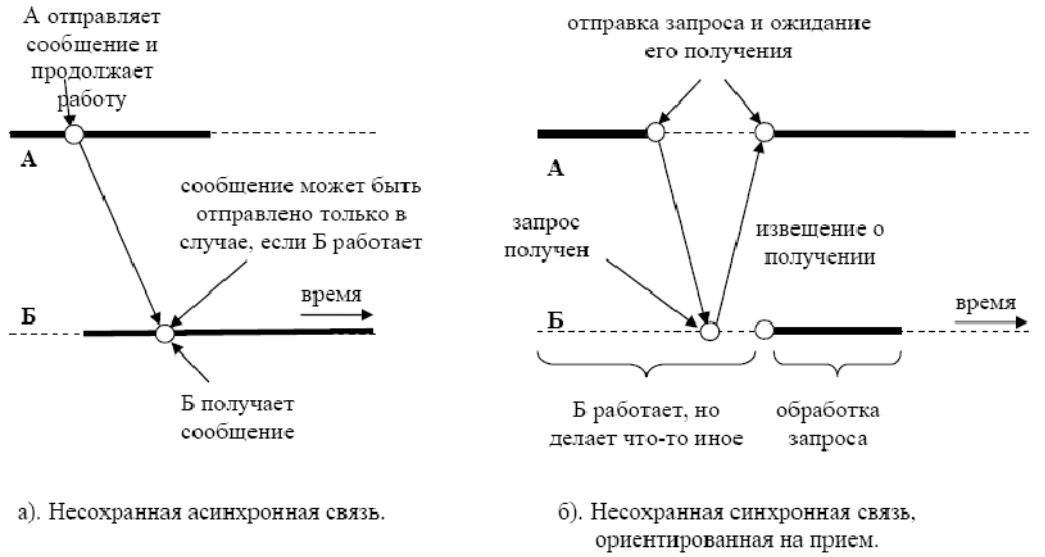
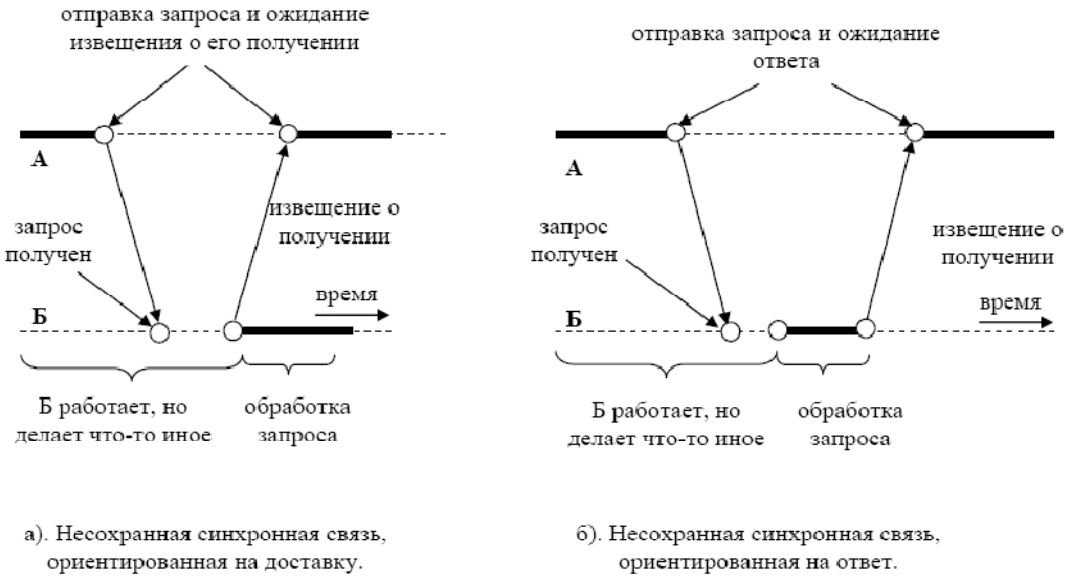


Рис. 1.6. Несохранная асинхронная связь и несохранная синхронная связь между подсистемами А и Б распределенной системы, ориентированная на прием.

Синхронная несохранная связь существует в нескольких вариантах. В наиболее слабой форме, основанной *на подтверждениях приема*

сообщений (*Рис. 1.6-б*), отправитель блокируется до тех пор, пока сообщение не окажется в локальном буфере принимающего комплекса. После получения подтверждения отправитель продолжает свою работу. Другая форма этой связи, ориентированная *на доставку* (*Рис. 1.7-а*), предполагает, что отправитель должен быть заблокирован до момента доставки сообщения самому получателю, который продолжит свою часть работы по его обработке. Наиболее жесткий вариант синхронной несохранной связи, ориентированный *на ответ* (*Рис. 1.7-б*), предполагает блокировку отправителя до получения ответного сообщения с другой стороны, как при работе систем *клиент/сервер в режиме запрос/ответ*. Эта же схема взаимодействия характерна и для систем, построенных на базе моделей удаленного вызова процедуры и удаленного обращения к методу. Каждый вид связи находит свое применение в распределенных системах на том или ином уровне составляющих их компонентов.



а). Несохранная синхронная связь, ориентированная на доставку.

б). Несохранная синхронная связь, ориентированная на ответ.

Рис. 1.7. Виды несохранной синхронной связи между подсистемами А и Б распределенной системы.

2. Основные механизмы в распределенных системах
2.1. Формы реализации системной поддержки

Примером того, как абстракции системной поддержки могут оказаться полезными при разработке программ взаимодействия, служит понятие *удаленного вызова процедуры* (*Remote Procedure Call, RPC*). Пользуясь абстракцией RPC можно полностью отвлечься от необходимости беспокоиться о каналах связи, об ошибках, возникающих при передаче, о согласовании действий двух прикладных систем, работающих на разных ЭВМ, о разнице в форматах представления данных на разных ЭВМ. Все, что требуется при пользовании системной поддержкой RPC, это сформулировать запрос в виде обращения к процедуре с параметрами, которая скроет нижние уровни сетевого взаимодействия (*Рис. 2.1*).

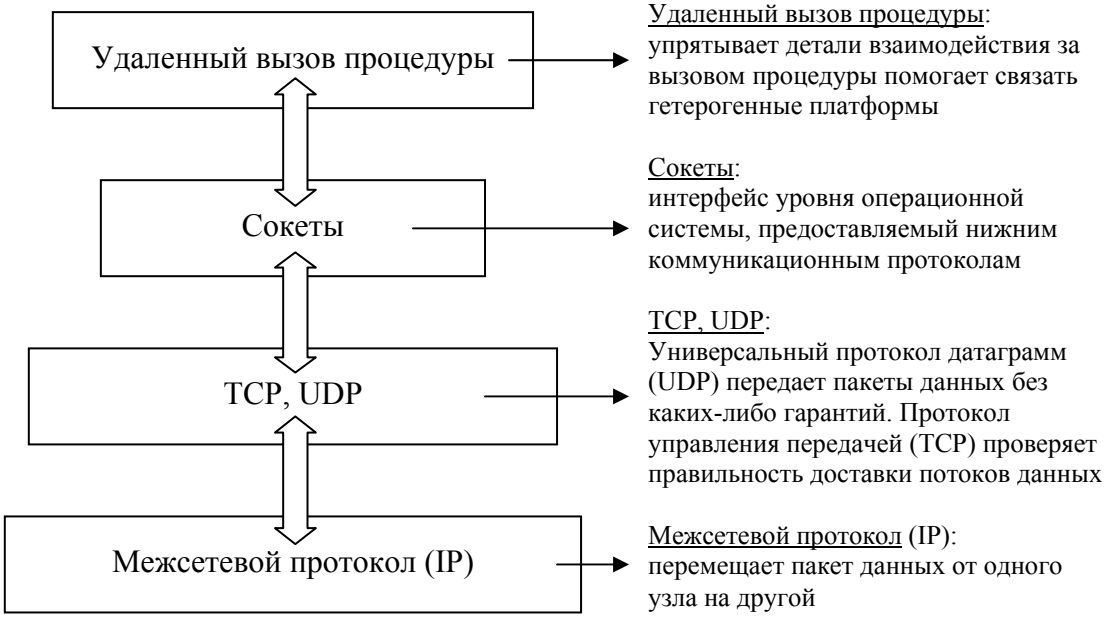


Рис. 2.1. Удаленный вызов процедуры как программная абстракция, построенная на базе других коммуникационных слоев.

Если необходимо обеспечить поддержку управления ошибками и сбоями, можно использовать транзакционный вариант удаленного вызова процедур. Делая транзакционные вызовы, можно не беспокоиться о состоянии вызова или взаимодействия, если в процессе их выполнения произошла ошибка. Если приложение или какая-нибудь его часть не срабатывает, транзакционные гарантии означают, что не возникнет никаких нежелательных побочных эффектов. Транзакционные гарантии в данном случае даются именно на прикладном уровне, то есть по отношению последовательности из нескольких удаленных вызовов процедур. Это предполагает значительно более сложный контроль, чем

гарантии, обычно дающиеся на транспортном уровне (например, в транспортном протоколе TCP, и даже в его транзакционном варианте T/TCP, гарантирующем надежную доставку отдельных пакетов данных, составляющих отдельное сообщение).

Чтобы такое упрощение способа взаимодействия стало возможным, необходимо иметь развитое программное обеспечение: язык описания интерфейсов, транслятор с этого языка и большое число библиотек, реализующих функциональность, необходимую для удаленного вызова процедуры. Часть этих программ используется на стадии разработки программ, другие необходимы в момент выполнения вызова. Системная поддержка может принимать следующий вид:

- ***Системы на базе удаленного вызова процедуры***. Наиболее общая форма взаимодействия. В настоящее время системы RPC лежат в основе почти всех других форм системного программного обеспечения, включая и сетевые службы.
- ***Транзакционные мониторы*** – наиболее надежная и наиболее стабильная технология, применяемая при интеграции прикладных систем. Транзакционные мониторы выполняют удаленные процедуры с транзакционными расширениями. В зависимости от реализации в двух- или трехъярусной архитектуре, транзакционные мониторы делятся на легкие и тяжелые. Легкие мониторы обеспечивают RPC-интерфейс к базам данных, а тяжелые представляют собой наиболее содержательные системы с огромным числом программных инструментов, которые часто превосходят инструментарий операционных систем.
- ***Брокеры объектов***. Системы RPC появились в те времена, когда доминировали процедурные языки программирования. Объектно-ориентированный подход привел к появлению брокеров объектов. Эти платформы по своим спецификациям более развиты, чем большинство RPC систем, но в терминах реализаций от них мало отличаются. На практике почти все они используют механизм RPC для реализации своих функций.
- ***Мониторы объектов***. Развитие функциональности брокеров объектов сделало очевидным, что ее большая часть была уже реализована в транзакционных мониторах. В то же время возникла необходимость перевести транзакционные мониторы, основанные на процедурных языках, на объектно-ориентированные языки программирования. Возникшие системы стали называться мониторами объектов.
- ***Системы на базе обмена сообщениями***. Еще в первых RPC-системах выяснилось, что синхронное взаимодействие не всегда обязательно.

Асинхронное взаимодействие было обеспечено транзакционными мониторами с системами сохранных очередей. Однако такие системы очередей имеют собственную ценность. Их стали называть системами на основе обмена сообщениями (*Message Oriented Middleware, MOM*). Они обеспечили транзакционный доступ к очередям, сохранные очереди и большое количество примитивов для чтения и записи в локальные и удаленные очереди.

- ***Брокеры сообщений*** – это отдельная форма систем на основе обмена сообщениями, дополнительно выполняющая преобразования и фильтрацию сообщений при прохождении через систему очередей. На основе содержания сообщений они могут динамически выбирать их получателей. Единственным их отличием от систем очередей является то, что здесь к очередям добавлена дополнительная функциональность, а это допускает более сложное асинхронное взаимодействие.

2.2. Принципы реализации удаленного вызова процедур

Модель RPC – это одна из основных моделей, применяемых в программном обеспечении системной поддержки. Впервые она была применена в двухъярусных системах типа "клиент/сервер": клиент вызывает процедуру, работающую на сервере. Существенно, что для клиента этот вызов не отличим от вызова локальной процедуры. Модель RPC ввела сами понятия клиента (вызывающая программа) и сервера (программа, реализующая удаленно вызываемую процедуру). В модели также были представлены другие концепции, широко применяемые до сих пор: языки описания интерфейсов, службы ведения имен и каталогов, динамическое связывание, интерфейс службы и т. д.

При применении модели RPC синхронизация, установление связи, передача параметров и результата – все делается скрытно от клиента. Это первая модель, позволившая добиться прозрачности и межъязыковой интероперабельности, отказаться от явного обмена сообщениями с помощью протоколов нижних уровней.

Клиентская и серверная части распределенной системы оказываются при взаимодействии с помощью удаленного вызова процедур совершенно независимыми от реализаций друг друга, в частности от используемых в этих реализациях языков программирования. В традиционных системах при обращении из программы, написанной на одном языке, к процедуре, написанной на другом языке, необходимо знать многие технические детали такого обращения: подробности представления типов данных на разных языках (точнее при использовании разных компиляторов), способы выравнивания элементов и заполнения пустот в сложных структурах,

детали стекового механизма. Сложности многократно возросли бы, если бы также организовывалось взаимодействие программ, не только написанных на разных языках, но и работающих на разных вычислительных машинах. От всего этого позволяет избавиться модель удаленного вызова процедуры. Программисты получили возможность строить распределенные приложения, не меняя языков и программных парадигм.

Модель RPC является ядром большинства распределенных систем. На этой модели основаны многие появившиеся позднее модели, в частности, модели удаленного вызова метода (*Remote Method Invocation, RMI*) и хранимых процедур (*stored procedures*) баз данных. Часто модель RPC используется как низкоуровневый примитив для реализации более сложных форм взаимодействия.

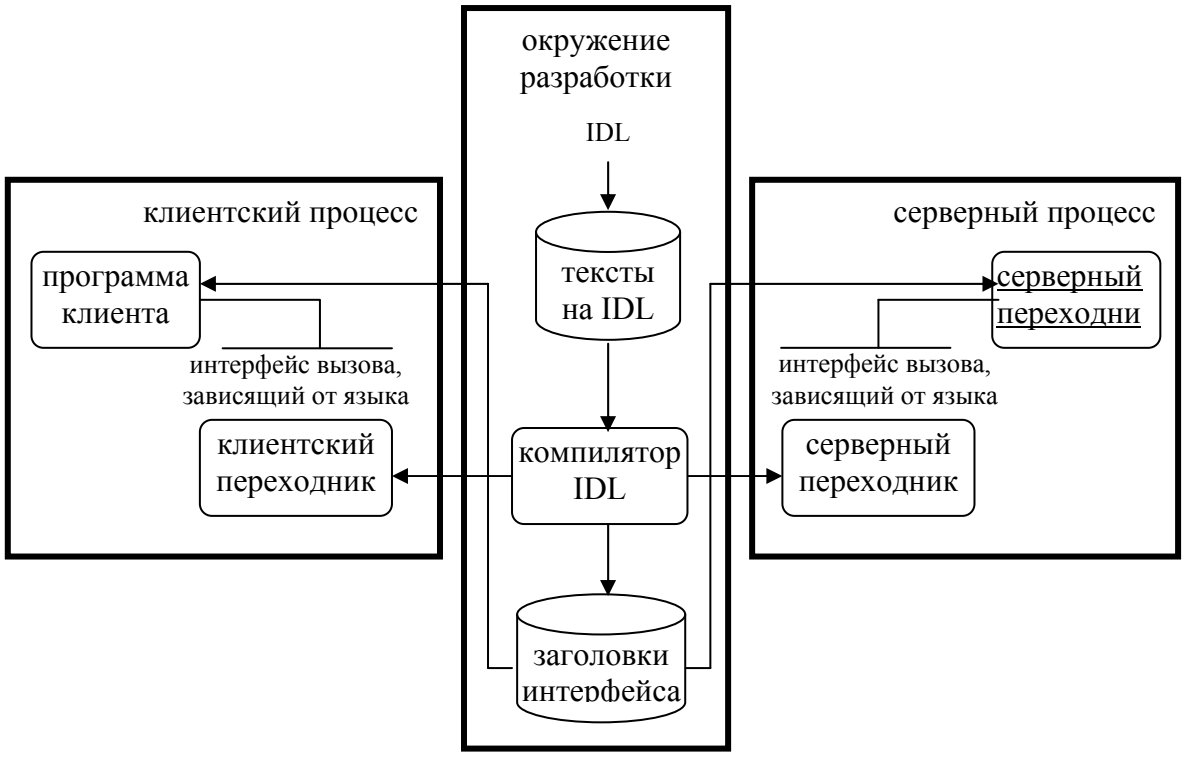


Рис. 2.2. Разработка распределенных приложений с помощью модели RPC.

Процесс удаленного вызова выглядит следующим образом:

- Процесс на машине клиента вызывает процедуру на машине сервера.
- Процесс клиента приостанавливается.
- На машине сервера запускается процесс выполнения вызванной процедуры.
- Результат передается на машину клиента.
- Процесс клиента возобновляется.

В описанном механизме много сложностей: разные адресные пространства клиента и сервера, необходимость передачи параметров и результатов, необходимость обрабатывать сбои и отказы оборудования. Дополнительную сложность вызывает то, что часть параметров в программах на процедурных языках программирования может передаваться по значению, а часть ссылками на значения, передаваемые в качестве параметров.

При реализации модели RPC необходимо преодолевать все подобные трудности, поэтому прежде, чем выполнять удаленную процедуру, необходимо провести предварительную подготовку.

Первым шагом должно быть определение интерфейса процедуры, что делается с помощью языка описания интерфейсов (IDL). Это определение задает краткое описание параметров процедуры, передаваемых ей до выполнения и возвращаемых после работы. Определение процедуры на языке IDL можно рассматривать как спецификацию сервиса, предоставляемого сервером (*Рис. 2.2, 2.3*).

```
[
  object, uuid (E7CDODOO-1827-11CF-9946-444553540000)
]
  interface ISpellChecker : Unknown
{ import "unknwn.idi"
  HRESULT LookUpWord              (  [in] OLECHAR word [31], [out] boolean * found);
  HRESULT AddToDictionary         (  [in] OLECHAR word [31]):
  HRESULT RemoveFromDictionary    (  [in] OLECHAR word [31]);
}
```

Рис. 2.3. Пример записи на языке описания интерфейсов IDL DCE для разработки распределенных приложений с помощью модели RPC.

Вторым шагом подготовки является трансляция созданного описания. Всякая реализация механизма RPC, любая интеграционная платформа, использующая RPC или сходную концепцию, содержит специальный интерфейсный транслятор. В результате трансляции создаются:

- ***Клиентский переходник***. Каждое описание заголовка процедуры в файле IDL приводит к созданию отдельного клиентского переходника (*stub*) для данной процедуры (*Рис. 2.4*). Переходник – это программа, которая после трансляции присоединяется к программе клиента. В состав клиентского переходника входят программы поиска сервера, форматирования данных, взаимодействия с сервером, получения ответа и передачи ответа в виде возвращаемых параметров вызванной клиентом процедуры. Форматирование данных клиентом состоит из двух процессов: маршалинга и сериализации. Маршалинг заключается в

перекодировке и упаковке данных в принятый в конкретной системе формат сообщения. Сериализация состоит из преобразования сообщения в последовательность байтов.

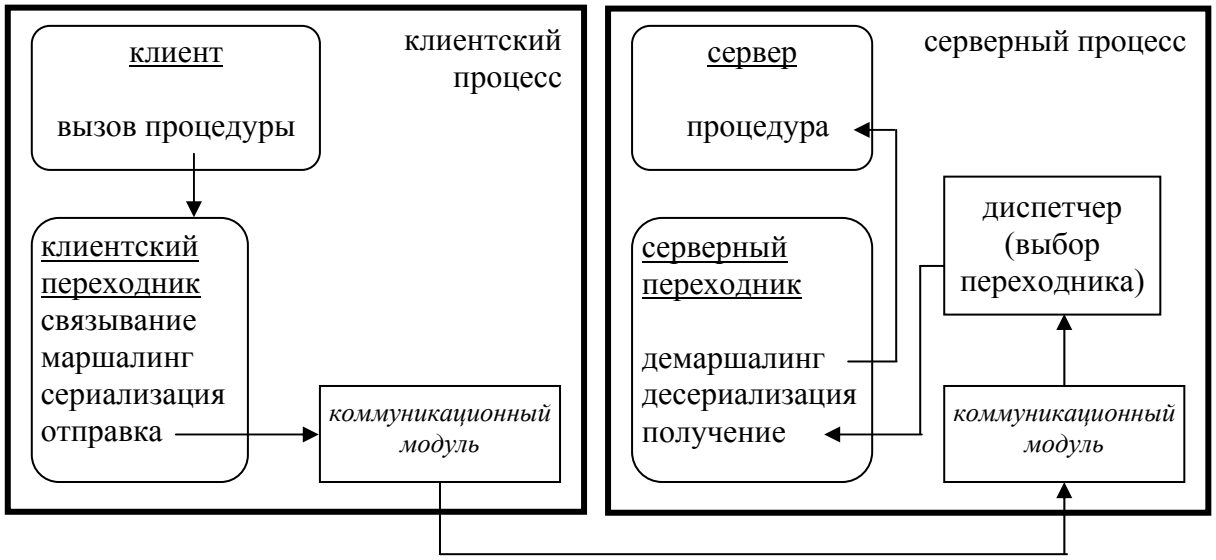


Рис. 2.4. Принцип работы модели RPC.

- ***Серверный переходник*** похож на клиентский, но реализует серверную часть вызова. Он состоит из программ, выполняющих получение запроса от клиента, форматирования данных (зеркального преобразования, то есть десериализации и демаршалинга), вызова реальной процедуры, реализованной на сервере, а также возврата результатов клиенту. Подобно клиентскому переходнику после трансляции серверный переходник присоединяется к программе сервера. От всей серверной программы, включая реализацию процедуры, вызываемой удаленным клиентом, не требуется быть написанной на том же языке, который был использован для реализации клиента, чем удается добиться существенного повышения межъязыковой интероперабельности, то есть возможности осуществлять взаимодействия программ, написанных на разных языках программирования.
- ***Программные шаблоны и ссылки***. Язык IDL и транслятор с него помогают вести разработку процедур, создавая вспомогательные файлы. Например, первые версии систем RPC создавались на языке Си, поэтому в дополнение к клиентскому и серверному переходнику транслятор IDL создавал файлы-заголовки (*.h). Многие современные трансляторы генерируют шаблоны программ для сервера, например, программы, содержащие заголовки процедур, затем разработчик должен сам создать реализацию этих процедур.

Когда клиент вызывает удаленную процедуру, на самом деле выполняется локальный вызов процедуры, являющейся частью переходника. Параметры вызова клиентский переходник получает стандартным образом, но действия, соответствующие реальной процедуре, не выполняются. Вместо этого переходник отыскивает сервер и форматирует необходимые данные. Перед отправкой сообщения в коммуникационный канал выполняются маршалинг и сериализация, что делает сообщение понятным получателю. Затем устанавливается соединение с сервером, клиент блокируется и ждет ответа (*Рис. 2.5*).

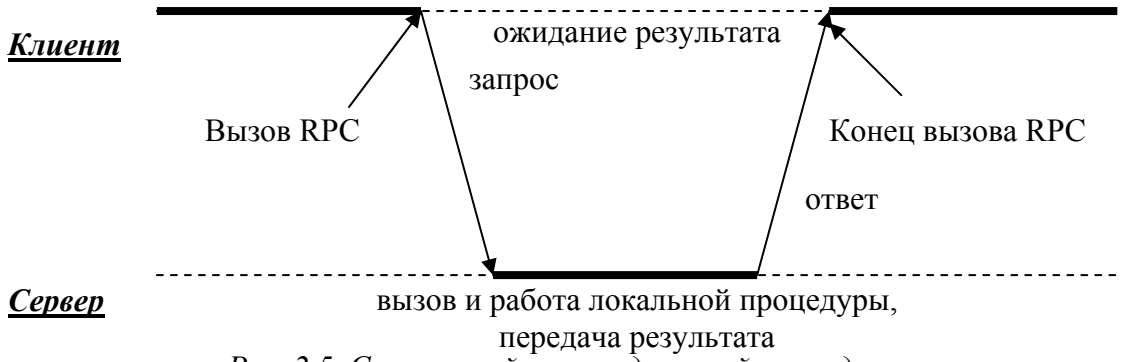


Рис. 2.5. Синхронный вызов удаленной процедуры.

На сервере запускается находящийся в состоянии ожидания серверный переходник, который преобразует сообщение в параметры локальной процедуры. Сервер воспринимает вызов как прямое обращение к его локальной процедуре с параметрами. Результаты работы процедуры упаковываются серверным переходником в сообщение для клиента, которое ему отсылается. На клиентской стороне выясняется, что сообщение адресовано клиенту в качестве ответа на вызов (операционная система не различает клиента и его переходник), сообщение попадает в буфер, а клиент выводится из состояния ожидания. Его переходник распаковывает результаты, записывает их в память клиента и передает управление основной программе.

Основное преимущество модели RPC состоит в том, что и клиент, и сервер не знают об удаленности вызова.

Установление связи с сервером, на котором локализована удаленная процедура, есть процесс, посредством которого клиент создает локальное соединение с данным сервером с целью обратиться к удаленной процедуре. Связывание может быть статическим или динамическим. При статическом связывании информация о сервере, на котором размещена процедура, закодирована прямо в программах клиента. Это может IP-адрес и номер порта, адрес Ethernet, адрес X.500 и т. д. Преимущества статического связывания – в простоте и эффективности. Недостаток

статического связывания заключается в слишком тесной связи клиента и сервера. Проявляется недостаток в том, что если сервер не работает, клиент тоже не в состоянии работать. Если сервер сменил адрес, клиент должен быть перекомпилирован с новым переходником, в котором указан новый правильный адрес. Наконец, для повышения производительности невозможно использование дополнительных серверов. Балансировку нагрузки на сеть надо проводить на стадии разработки клиентской части.

При динамическом связывании создается специализированная служба, ответственная за локализацию серверов. Динамическое связывание создает новый программный слой для снятия косвенности и повышения гибкости системы за счет ее производительности. Этот новый слой называется сервером имен и каталогов, он предназначен для поиска адресов серверов по именам вызываемых процедур. В этом варианте перед вызовом клиентский переходник запрашивает сервер каталогов и только потом обращается к указанному ему серверу (*Рис. 2.6*).

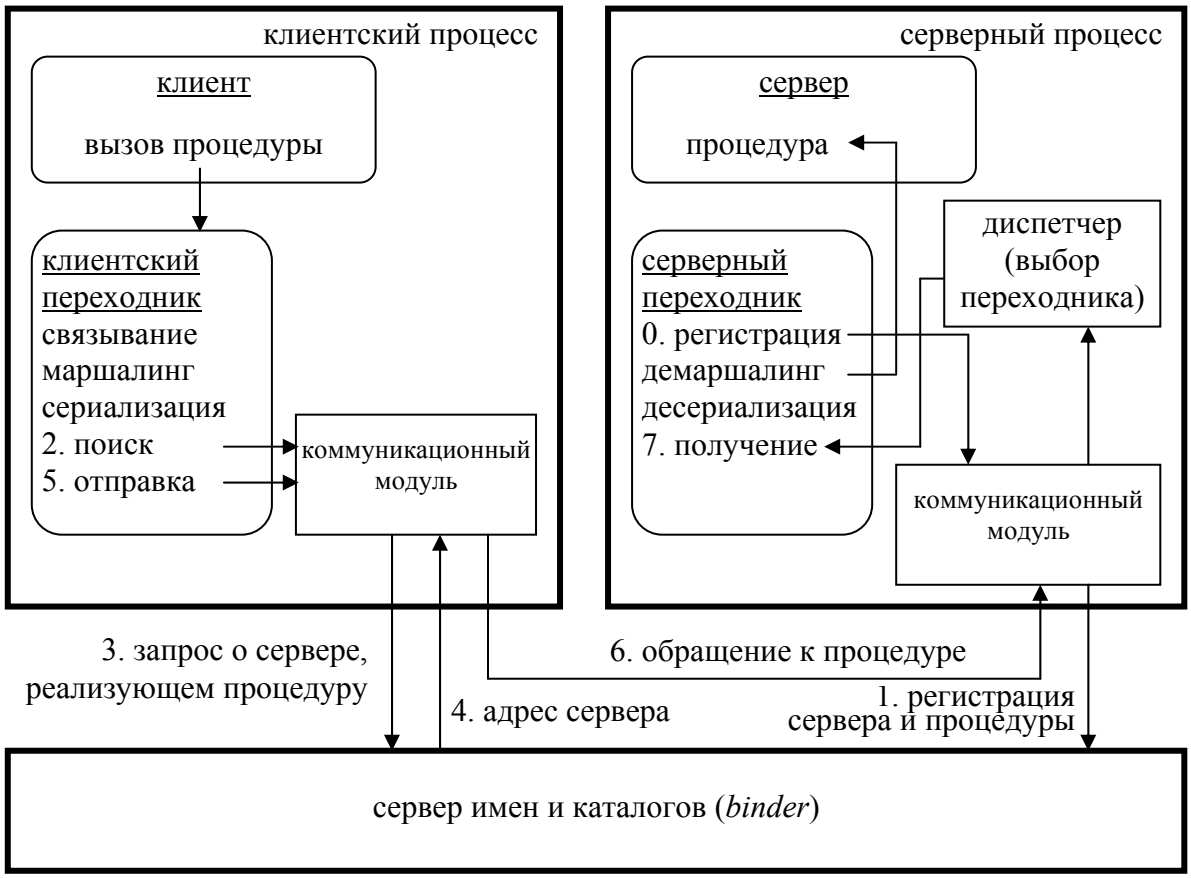


Рис. 2.6. Динамическое связывание: регистрация процедуры сервером (шаги 0 и 1), запрос адреса сервера, реализующего нужную процедуру, клиентом (шаги 2 и 3), получение адреса от сервера имен и каталогов (шаг 4), вызов процедуры (шаги 5, 6 и 7).

Динамическое связывание добавило в модель RPC много новых возможностей. Отслеживая обращения к серверам, удается, например,

балансировать нагрузку на них. Если сервер меняет свой адрес, все обращения к нему могут менять свои маршруты, а это развязывает клиенты и серверы, добавляя гибкость при внедрении и эксплуатации системы. Платой за эту гибкость является добавочная инфраструктура в виде сервера каталогов, протокола взаимодействия с ним и примитивов регистрации процедур на серверах. Клиент не должен знать, проводится связывание статически или динамически.

Если машины клиента и сервера идентичны, то после их связывания друг с другом в сообщения, отправляемые между ними, можно упаковывать параметры и имя (код) вызываемой процедуры, а затем отправлять эти сообщения получателям. Если же эти машины в чем-то различаются, приходится предпринимать множество дополнительных действий.

Разные системы могут иметь разные представления чисел и символов. При передаче информации между машинами с разными архитектурными особенностями эту информацию приходится предварительно преобразовывать к некоторому стандартному виду, не зависящему от архитектурных особенностей ЭВМ, а затем снова преобразовывать в иное конкретное представление. Еще более сложна передача параметров не по значению, а по ссылке. Передавать адрес памяти с одной машины на другую – бессмысленно, поскольку на другой машине по этому адресу размещены другие данные. Иногда для решения проблемы принимают решение передавать массивы по значению (в сообщение от клиента к серверу упаковывается весь массив). Однако очевидно, что при выполнении удаленного вызова процедуры обе стороны должны следовать согласованным протоколам, а интерпретация форматов должна быть однозначной.

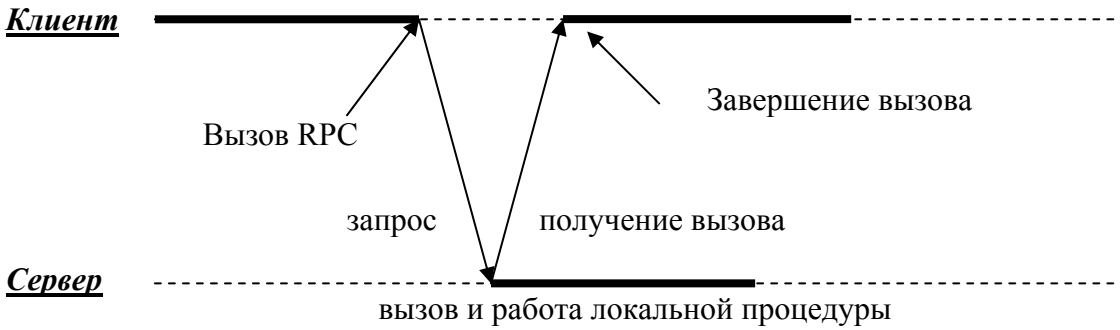


Рис. 2.7. Асинхронный вызов удаленной процедуры.

Стандартный вызов удаленной процедуры подразумевает полную блокировку клиента до получения ответа от сервера. Иногда ответ от сервера не нужен, в подобных случаях используют асинхронные вызовы.

Клиент продолжает работу после запуска RPC (*Рис. 2.7*). Иногда делают два раздельных (встречных) вызова (*Рис. 2.8*), что называется *отложенной синхронизацией*. Вызов, при котором от сервера не требуется подтверждения получения запроса, называется *односторонним* удаленным вызовом.

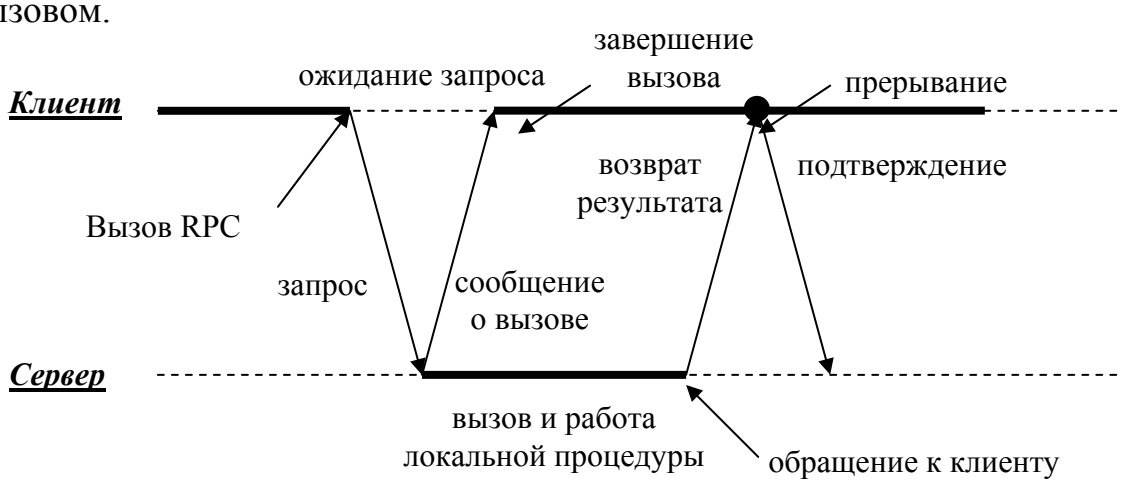


Рис. 2.8. Отложенная синхронизация вызова удаленной процедуры.

2.3. Транзакционное взаимодействие

2.3.1. Свойства транзакционного взаимодействия

Транзакцией называется последовательность операций, выполняемая системой, как единое целое. Транзакции превращают процессы доступа и модификации множества элементов данных в одну атомарную операцию. Если в процессе выполнения транзакции будет определено, что дальнейшее ее выполнение невозможно (по любой причине), все данные восстанавливаются с теми значениями и в том состоянии, в котором они были до начала транзакции. Это свойство называется "***все или ничего***".

Транзакции могут состоять и из одной операции, но особенно важно применять подход "*все или ничего*" в тех случаях, когда необходимо последовательно выполнить несколько технически независимых, но нераздельных с точки зрения прикладной программы операций (например, банковский перевод со снятием и вложением средств на другой счет). Именно такие операции необходимо объединять в транзакции, чтобы либо операции выполнялись совместно, либо не выполнялась ни одна.

Ключевой является возможность отката ситуации к исходному состоянию при невозможности совершить транзакцию. Для программирования транзакций создаются специальные приемы и разрабатываются системные примитивы, которые могут поддерживаться как базовой операционной системой, так и дополнительными программными системами (транзакционными мониторами). Список примитивов зависит от используемых в транзакциях объектов, и в разных системах эти примитивы разные. Однако практически во всех системах

должны быть транзакционные скобки (начать и завершить транзакцию), операция прерывания транзакции (на случай, например, фиксации ошибки в данных), операции транзакционного чтения и транзакционной записи данных, входящих в некоторый файл, таблицу базы данных и так далее.

Чтобы транзакции действительно выполняли свою роль, они должны:

- быть атомарными (***A***tomic). Атомарность гарантирует, что транзакция либо полностью выполняется, либо полностью не выполняется, то есть с точки зрения окружающих систем транзакция выполняется как одна неделимая операция. Пока транзакция находится в процессе выполнения, другие системы не могут наблюдать никаких ее промежуточных состояний;
- быть непротиворечивыми (***C***onsistent). Непротиворечивость есть соблюдение инвариантов системы. Для каждой системы такие инварианты свои, например, в банковских системах инвариантом служит общая сумма вложенных средств. Никакая внутренняя операция (не затрагивающая кассу) не меняет общую сумму средств в банке;
- быть изолированными (***I***solated). Изолированность или сериализуемость – это отсутствие влияния на параллельно выполняемые транзакции. Если какие-либо транзакции выполняются параллельно, итог будет таким же, как если бы все транзакции выполнялись последовательно в некотором (задаваемом системой) порядке;
- быть долговечными (***D***urable). Никакие сбои после завершения операции не могут привести к отмене результатов транзакции.

В совокупности все эти свойства объединяют термином ***ACID***. К наиболее важным видам транзакций можно отнести транзакции *плоские*, *составные* и *распределенные*.

Плоские транзакции в полной мере обладают свойствами ACID. Это наиболее простой и наиболее часто используемый тип транзакции. Однако плоские транзакции имеют ограничения, в частности, они не могут иметь частичного результата в случае завершения или прерывания.

С помощью ***составных и вложенных транзакций*** удается разделить сложные транзакции верхнего уровня на серию иерархически вложенных параллельно работающих транзакций. Параллельность эта может быть вполне реальной: вложенные транзакции могут выполняться на других машинах, но может быть и виртуальной, то есть выполняемой для ускорения или упрощения программирования. Каждая из вложенных транзакций может делиться на транзакции аналогичным образом.

Составные транзакции не в полной мере обладают свойствами ACID. Например, свойство долговечности применимо только к транзакциям самого верхнего уровня (результаты вложенных транзакций могут быть отменены, если не сможет быть выполнена какая-либо из других вложенных транзакций). Вложенные транзакции требуют серьезного подхода к администрированию. Семантика такого администрирования такова: в начале любой транзакции создается копия данных всей системы. Если транзакция прерывается, копия просто уничтожается, если она завершается успешно, ее внутренняя копия заменяет внешнюю и так далее.

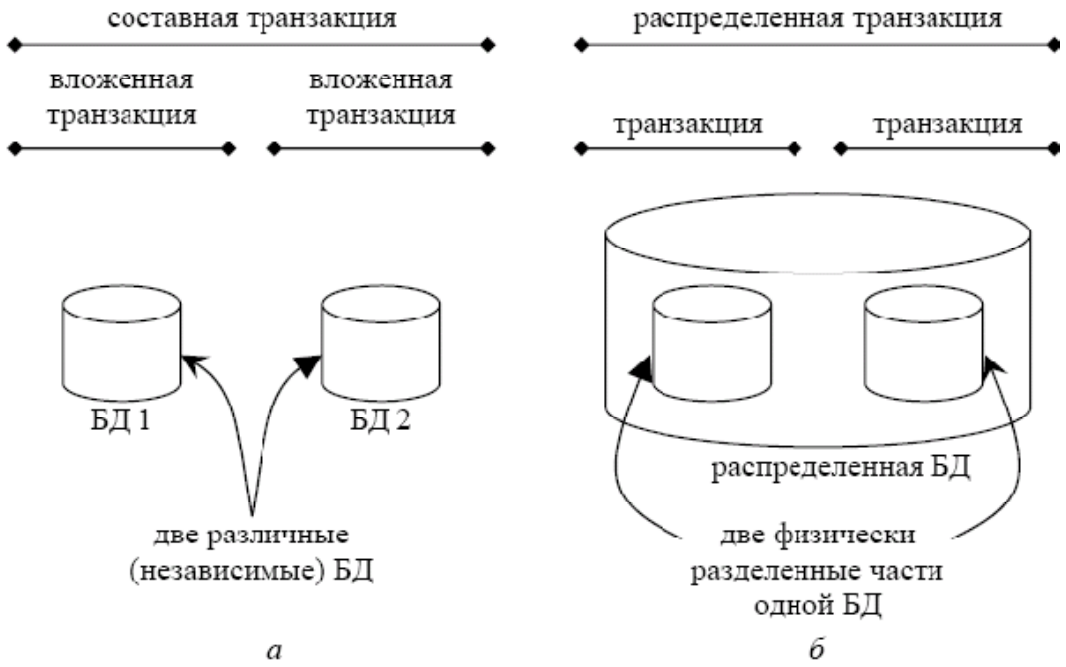


Рис. 2.9. Составная и вложенные транзакции (а) и распределенная транзакция (б).

Вложенные транзакции делят исходную транзакцию *логически*, а логическое разделение транзакций не обязательно означает *распределенности*. ***Распределенные транзакции*** представляют собой плоские неделимые транзакции, работающие с распределенными данными (*Рис. 2.9*).

2.3.2. Протоколы подтверждения транзакции

Реализация распределенных транзакций требует реализации надежных алгоритмов подтверждения возможности их выполнения, которые должны получить все участники процесса (либо ни один из них). Распределенное подтверждение часто реализуется при помощи координаторов. В простейшей схеме координатор сообщает всем остальным процессам, участвующим в транзакции, в состоянии ли они осуществить запрашиваемую операцию. Эта схема известна под названием ***однофазного подтверждения*** (*1PC*). Протокол 1PC обладает серьезным

недостатком: если один из участников на самом деле не может осуществить операцию, он не в состоянии сообщить об этом координатору.

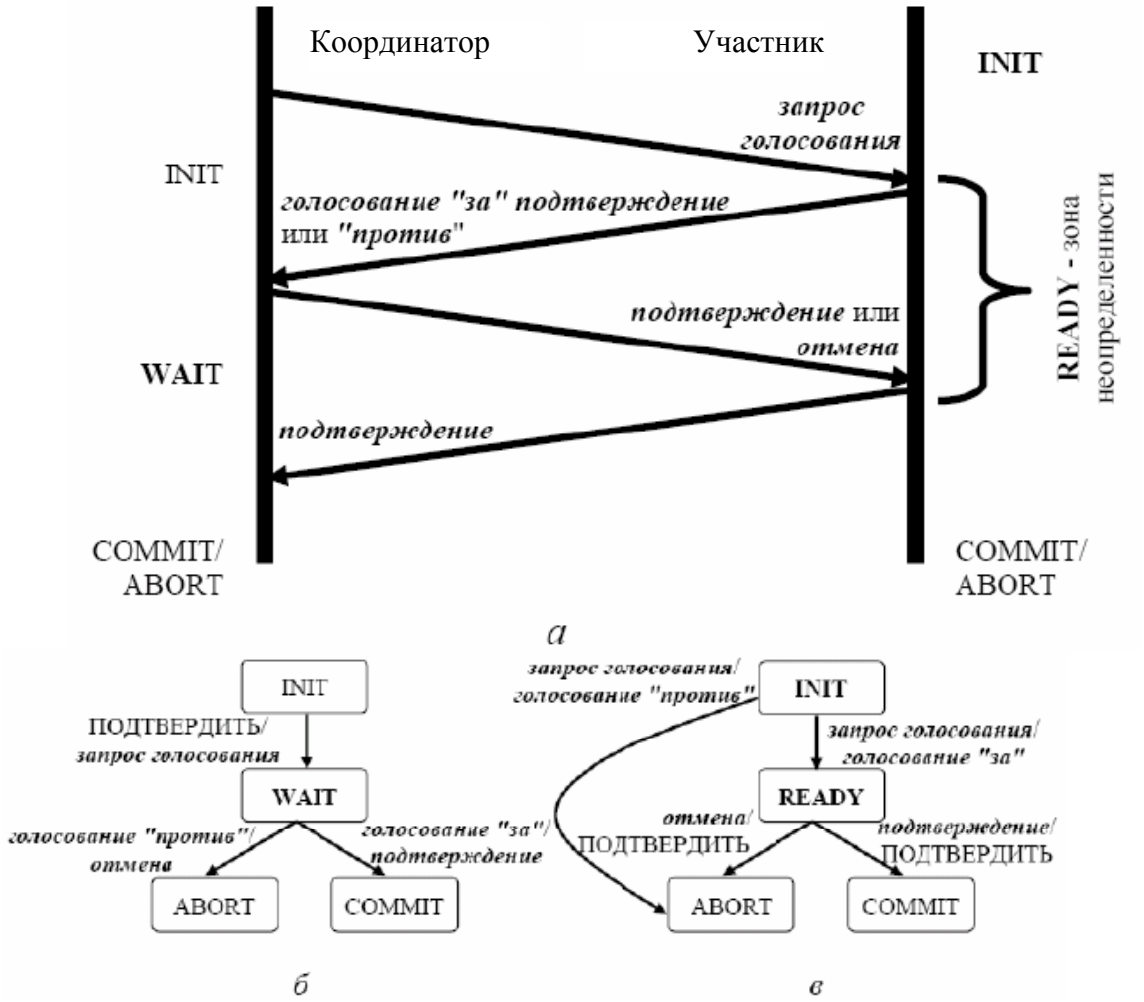


Рис. 2.10. Протокол двухфазного подтверждения: (а) обмен сообщениями, (б) конечный автомат координатора, (в) конечный автомат участника (выделены блокирующие состояния).

Протокол ***двухфазного подтверждения*** (*2PC*) строится из двух фаз, каждая из которых включает в себя два шага. Первая фаза называется фазой голосования и состоит из шагов 1 и 2, вторая фаза (фаза решения) состоит из шагов 3 и 4 (*Рис. 2.10*):

1. Координатор рассылает участникам запрос голосования.
2. Участник после получения запроса подает свой голос координатору о том, что он готов подтвердить свою часть транзакции (проголосовать "за"), либо отказаться от такого подтверждения (проголосовать "против").
3. Координатор собирает ответы всех участников. Если все участники подтвердили транзакцию, координатор начинает выполнять

соответствующие действия и посылает всем участникам свое решение – провести подтверждение транзакции. Однако если хотя бы один участник отказался подтвердить свою часть транзакции, координатор рассылает всем указание отмены транзакции.

4. Если участник, проголосовавший за подтверждение, получает подтверждение от координатора, он подтверждает транзакцию (выполняет свои внутренние действия по ее завершению). В случае же получения указания отмены выполнение транзакции прекращается.

Проблемы возникают, если при использовании двухфазного подтверждения в системе возникают ошибки. Координатор и другие участники имеют такие состояния, в которых они блокируются и ждут поступления сообщений извне. Если в работе координатора возникнет ошибка, работа протокола нарушается, поскольку другие процессы будут бесконечно ожидать сообщений. Для исправления ситуации вводится механизм тайм-аута. Если в течение некоторого времени сообщения о сводных результатах голосования от координатора не поступают прочим участникам, они приходят к выводу о необходимости прервать свою работу и послать координатору сообщение об отказе от подтверждения (проголосовав "против").

Из-за сбоев в работе самого координатора для протокола 2PC характерно наличие ситуации, когда участнику приходится блокироваться до восстановления работоспособности этого координатора. Такая ситуация возникает, когда все участники успевают получить запрос голосования, но после этого координатор выходит из строя. В этом случае участники даже совместно не состоянии решить, каким образом продолжить работу далее. По этой причине протокол 2PC называется *протоколом блокирующего подтверждения*. В таких случаях решение о разблокировке передается системному администратору, который подтверждает или отвергает транзакцию и согласовывает состояние системы после восстановления координатора.

2.3.3. Транзакционные мониторы

Для реализации транзакций применяются специальные программные системы – транзакционные мониторы, лежащие в основе множества многоярусных систем. Транзакционные мониторы появились раньше систем типа "клиент/сервер" и трехъярусных архитектур. Наиболее известным монитором транзакций является система CICS, разработанная фирмой IBM в конце 60-х годов и используемая до сих пор.

Первоначально мониторы транзакций разрабатывались для того, чтобы большие вычислительные машины могли обеспечивать мультиплексный доступ к как можно большему количеству внешних устройств (ресурсов) для как можно большего количества параллельно работающих пользователей. Частью этой задачи было обеспечение работы с параллельными процессами и надежными данными, отсюда и появилась дополнительная функциональность – транзакция.

2.3.3.1. Транзакционный удаленный вызов процедуры

Основная цель транзакционного монитора – поддержка выполнения распределенных транзакций. Традиционный удаленный вызов процедуры изначально был разработан для того, чтобы одна клиентская программа могла обратиться к одному серверу. Когда процесс взаимодействия усложняется (например, клиент вызывает процедуры с двух серверов, или клиент общается с сервером, который взаимодействует с базой данных), традиционный удаленный вызов неверно трактует вложенные вызовы как независимые друг от друга. Лучшим решением является сделать вызов процедуры *транзакционным*.

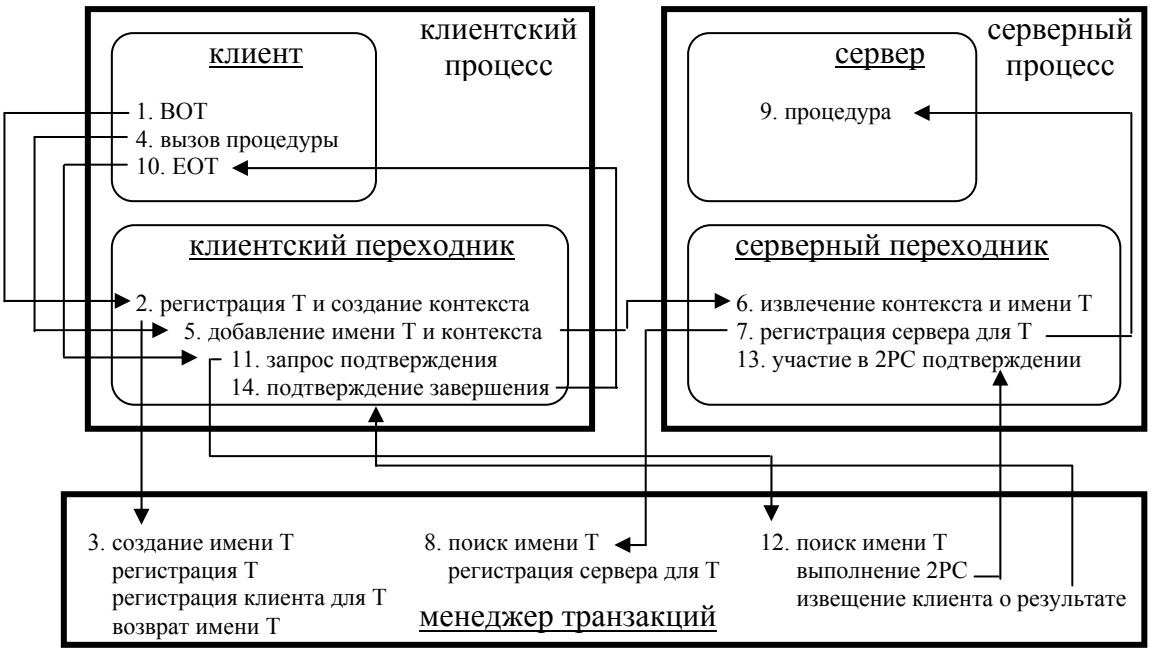


Рис. 2.11. Выполнение удаленного вызова внутри транзакционных скобок (T – транзакция, BOT, EOT – открывающая и закрывающая транзакционные скобки).

Семантика транзакционного удаленного вызова процедуры (TRPC) такова, что если группа вызовов процедур внутри транзакции успешно завершается, программист имеет гарантии, что ***все*** они завершились. Если вместо этого возникло прерывание выполнения группы вызовов (из-за

каких-либо ошибок), ни один из вызовов не выполняется (совокупный эффект будет таким, *как если бы* ни один из вызовов не выполнялся).

Таким образом, процедурные вызовы, заключенные в транзакционные скобки, рассматриваются как единое целое, и системное программное обеспечение гарантирует их атомарность (неделимость). Это достигается использованием модуля *транзакционного управления* или *менеджера транзакций*, координирующего взаимодействие между клиентами и серверами (*Рис. 2.11*). Менеджер транзакций транзакционного монитора обладает способностью создавать транзакционный контекст, который используется на протяжении всей последовательности вставленных в скобки вызовов, сколь бы длинной она ни была.

2.3.3.2. Функциональность транзакционных мониторов

Современные транзакционные мониторы – очень сложные системы, обычно их функциональность включает в себя:

- функциональность и механизмы, необходимые для поддержки удаленных вызовов (язык IDL, серверы имен и каталогов, безопасность и аутентификация, компиляторы переходников).
- программные абстракции для работы с TRPC: RPC, транзакционные скобки, механизмы обратных вызовов (когда в результате возникновения некоторого события на сервере, вызывается процедура на клиентской части).
- ведение журнальных записей, управление восстановлением, блокировкой и т. д.
- управление процессами, приписку приоритетов, балансировку нагрузки, репликацию, управление стартом и завершением.
- управление сценариями для асинхронного взаимодействия.
- инструментарий для установки, управления и мониторинга производительности компонентов системы.

В простейших случаях доступные программные абстракции просто преобразуют обычный удаленный вызов в транзакционный, в самых сложных вариантах они позволяют контролировать практически все аспекты взаимодействия.

2.3.3.3. Архитектура транзакционных мониторов

Клиентский интерфейсный компонент транзакционного монитора содержит прикладной *интерфейс*, а также поддержку прямого доступа через терминал и средства аутентификации пользователей. Компонент *программного потока* хранит, загружает и исполняет процедуры,

написанные на языке, специфичного для данного транзакционного монитора. Эти процедуры обычно содержат операции над логическими ресурсами, идентифицированными по именам.

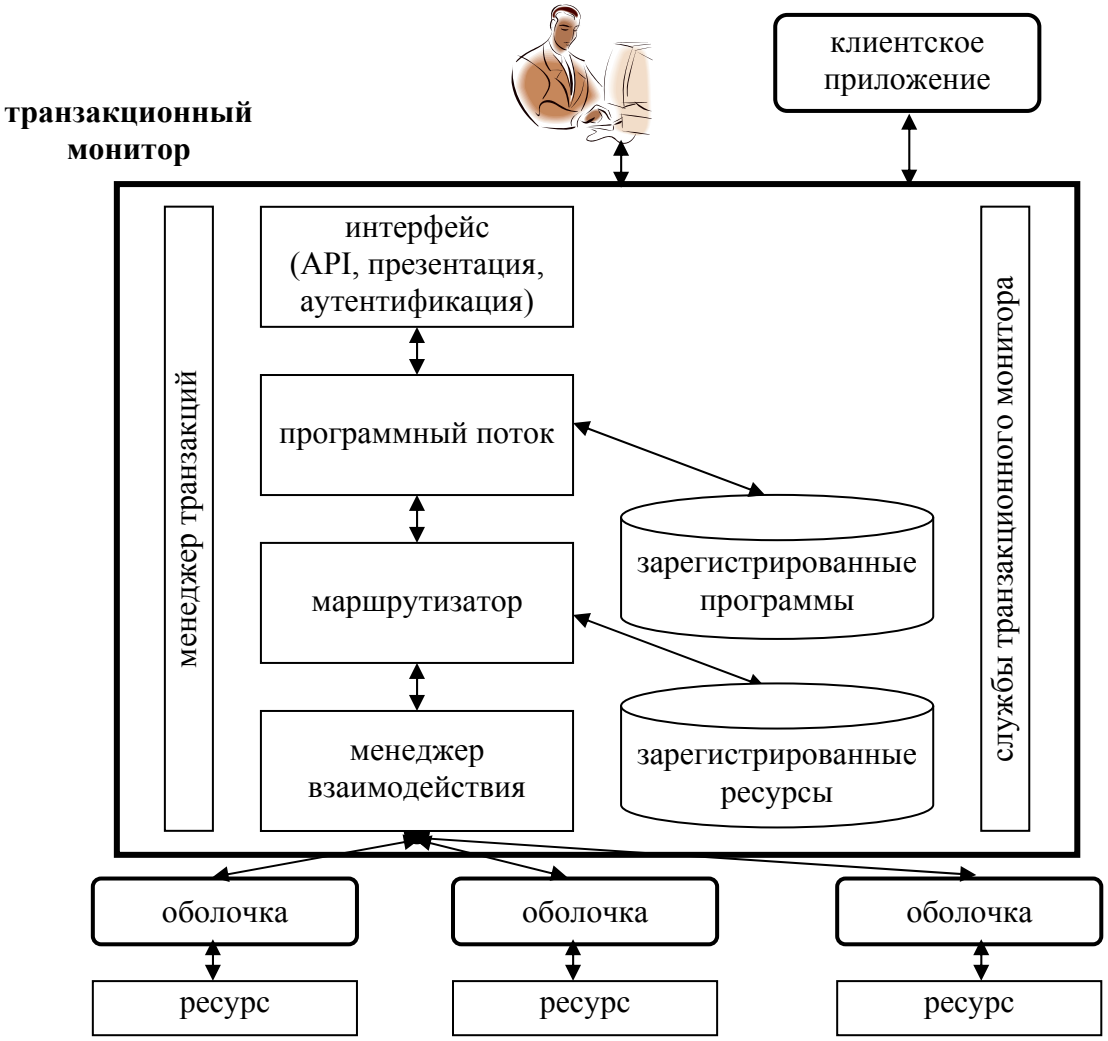


Рис. 2.12. Базовые компоненты транзакционного монитора.

Маршрутизатор ставит в соответствие операции и вызовы. Вызовы могут относиться к нижележащим ресурсам (например, базам данных) или локальным службам, предоставляемым самим транзакционным монитором. В состав маршрутизатора входит специализированная база данных, содержащая определения соответствий между именами логических ресурсов и физическими устройствами. В случае изменения конфигурации системы системный администратор должен всего лишь подправить это соответствие: клиентское приложение модифицировать не требуется, поскольку клиент знает только логические имена.

Взаимодействие с ресурсами (базами данных) осуществляется через *менеджер взаимодействия*, в то время, как *оболочки* скрывают гетерогенность различных ресурсов, связанных с транзакционным монитором. Это упрощает разработку модуля взаимодействия, поскольку

он перестает зависеть от характеристик индивидуальных ресурсов. Выполнение распределенной транзакции проходит через *менеджер транзакций*, исполняющий протокол 2PC и гарантирующий все транзакционные свойства процедур, исполняемых монитором.

В состав монитора включает значительное количество *служб транзакционного монитора*. В совокупности они обеспечивают производительность, высокую доступность, отказоустойчивость, репликацию и т. д. (*Рис. 2.12*).

2.4. Объектно-ориентированный подход к распределенной обработке информации

2.4.1. Распределенные объекты

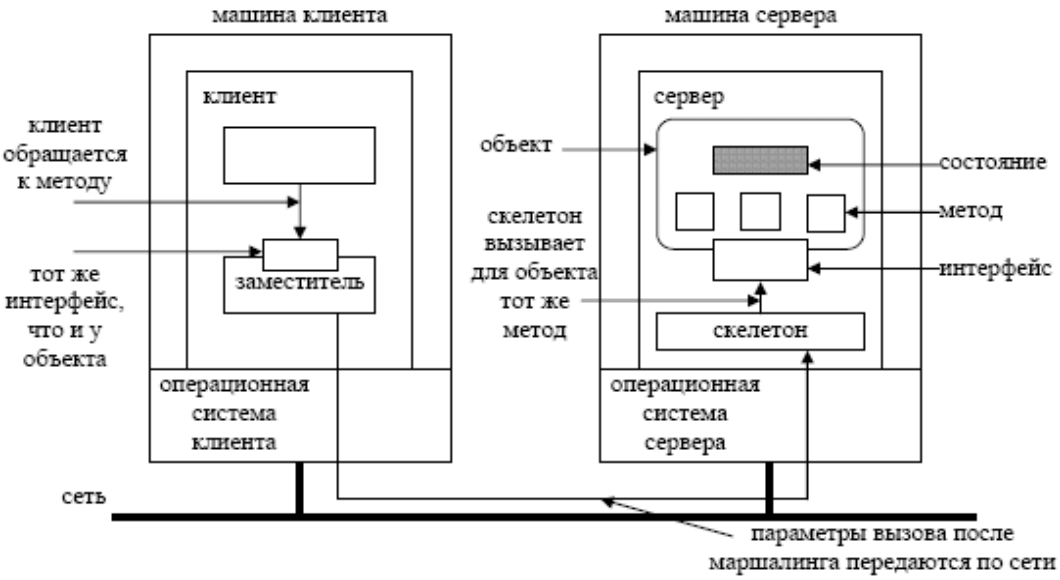


Рис. 2.13. Обобщенная организация удаленных объектов с использованием заместителя объектов.

Для распределенных систем разделение на объекты, характеризующиеся своим состоянием (данными), и интерфейсы, с помощью которых обеспечивается доступ к этим состояниям, особенно важно, поскольку позволяет помещать интерфейс на одну машину, имея сами объекты на другой. Обеспечивается такое распределенное представление информации тем, что при выполнении клиентской программой привязки к распределенному объекту (*Рис. 2.13*), в ее адресное пространство загружается реализация интерфейса объекта, называемая заместителем (*proxy*).

Заместитель клиента аналогичен переходнику при удаленном вызове процедуры. Он выполняет маршалинг параметров, упаковывая их в сообщениях при обращении к методам, и демаршалинг данных из ответных сообщений, содержащих результаты обращения к методам,

передавая их клиенту. Сами объекты находятся на сервере. Входящий запрос на обращение к методу сначала попадает в серверный переходник, называемый скелетоном (*skeleton*), который преобразует его в правильное обращение к методу через интерфейс объекта. Серверный переходник также отвечает за маршалинг параметров в ответных сообщениях и их пересылку заместителю клиента.

2.4.1.1. Объекты, создаваемые при компиляции и при выполнении

При программировании на объектно-ориентированном языке программист сам описывает *классы* (описания абстрактных типов в виде модулей, содержащих элементы данных и операции над этими данными) и вводит объекты – экземпляры классов. Работа с такими объектами в распределенной системе не представляет особых сложностей. Например, в языке Java объект может быть полностью описан в рамках своего класса и интерфейсов, которые этот класс реализует. Интерфейсы можно скомпилировать в переходники (клиентские и серверные), позволяющие обращаться к объектам Java, размещенным на удаленных машинах. Разработчик при этом работает только с текстом на языке Java.

Объекты, создаваемые таким образом, явно зависят от языка, на котором пишется исходная программа. Однако создаваться объекты могут и во время исполнения программы. Такой подход применяется во многих распределенных системах, поскольку распределенные приложения, созданные в соответствии с ним, не зависят от конкретного языка программирования. В частности, приложение может работать с объектами, описанными на разных языках программирования.

При работе с объектами времени исполнения способ реализации объекта остается открытым. Задача в том, чтобы превратить реализацию в объект, методы которого будут доступны с удаленной машины. Часто для этого используются адаптеры объектов, которые служат оболочками реализации с задачей придать реализации видимость объекта. Обычно, чтобы упростить процесс создания оболочки, объекты определяют исключительно в понятиях интерфейсов, которые они реализуют.

2.4.1.2. Сохранные объекты

Одно из важнейших свойств объекта – это его *сохранность*. Сохранный объект – это объект, продолжающий существовать, не находясь в адресном пространстве своего текущего сервера, то есть независящий от сервера. Практически это означает, что сервер, работающий с объектом, сохраняет его во вспомогательном запоминающем устройстве. Сервер может прекратить свою работу, но,

возобновив ее, может прочитать состояние сохранного объекта и вновь приступить к обработке запросов на обращение к нему. Объекты, не обладающие этим свойством, существуют, только пока сервер ими управляет.

2.4.1.3. Привязка клиента к объекту

Существенная разница между традиционными системами RPC и системами, работающими с распределенными объектами, состоит в том, что в новых системах могут создаваться уникальные в пределах системы ссылки на объекты. Такие ссылки могут свободно передаваться между процессами, запущенными на разных машинах, например, как параметры обращения к методу. Иногда механизм уникальных ссылок выбирается в качестве единственного средства обращения к объектам, улучшая прозрачность систем по сравнению с системами RPC.

Если процесс хранит ссылку на объект, то перед обращением к любому из методов объекта он должен сначала выполнить привязку. Результатом привязки будет заместитель объекта, размещаемый в адресном пространстве данного процесса и реализующий интерфейс с методами, к которым обращается процесс. Часто такая привязка выполняется автоматически. В таких случаях при получении ссылки на объект система должна найти сервер, управляющий этим объектом, и поместить заместитель объекта в адресное пространство клиента.

При *неявной привязке* клиент может напрямую запрашивать методы, используя только ссылку на объект. В случае *явной привязки* клиент должен до обращения к методам вызвать специальную функцию привязки к объекту. При явной привязке обычно возвращается указатель на локально доступный заместитель объекта.

2.4.1.4. Статическое и динамическое обращение к методам

После того, как клиент свяжется с объектом, он может через заместителя обратиться к методам объекта. Основное различие между моделями RMI и RPC состоит в том, что RMI в основном поддерживает внутрисистемные ссылки на объекты. Стандартный для RMI способ поддержки – описание интерфейсов объектов на языке определения интерфейсов (как в RPC). Такой подход называется *статическим обращением*. Статическое обращение требует, чтобы интерфейсы объекта при разработке клиентского приложения были известны. Одновременно предполагается, что при изменении интерфейса клиентское приложение перед использованием будет заново откомпилировано.

Иногда удобнее собирать параметры перед обращением к методу во время исполнения программы. Этот процесс называется *динамическим обращением*. Отличие от статического способа в том, что приложение выбирает, какой метод удаленного объекта вызвать во время выполнения, передавая процедуре динамического вызова идентификатор этого метода.

2.4.1.5. Передача параметров в модели RMI

Модель RMI имеет больше возможностей по организации передачи параметров, чем модель RPC, благодаря поддержке системных ссылок на объекты. Если все объекты, к которым предполагается обращение, являются распределенными, то есть доступными с удаленных машин, то при обращении к их методам в качестве параметров постоянно используются ссылки на объекты. Ссылки передаются по значению и копируются с одной машины на другую. Получив в качестве результата обращения к методу ссылку на объект, процесс, как только ему это понадобится, легко может выполнить привязку к объекту.

Однако использование только распределенной схемы обращения может приводить к потере эффективности, в особенности, если объекты просты (целые числа, логические значения). Если клиент не находится на том же сервере, что и сам объект, то каждое его обращение порождает запрос между различными адресными пространствами, а может быть и между разными машинами. Поэтому работа со ссылками на реально удаленные и локальные объекты происходит по-разному.

Копирование ссылки при передаче объекта в качестве параметра происходит только тогда, когда она относится к удаленному объекту. Если же ссылка относится к локальному объекту, то есть к объекту в адресном пространстве клиента, то клиенту передается сам объект.

2.4.2. Брокеры объектов

На основе модели RMI было создано множество реализаций, значительно облегчивших создание объектно-ориентированных распределенных приложений. В своей основе брокеры объектов представляют системное программное обеспечение, поддерживающее *способность объектов к взаимодействию*. Описание их базовых принципов будет проведена на примере наиболее известного брокера объектов.

Стандарт ***Common Object Request Broker Architecture*** (*CORBA*) – это архитектура и спецификация для создания и управления объектно-ориентированными приложениями, распределенными в вычислительной сети. В настоящее время выработано несколько версий стандарта CORBA,

которые вводят стандартизованную спецификацию брокера объектов, но не содержат никакой реализации.

2.4.2.1. Архитектура CORBA

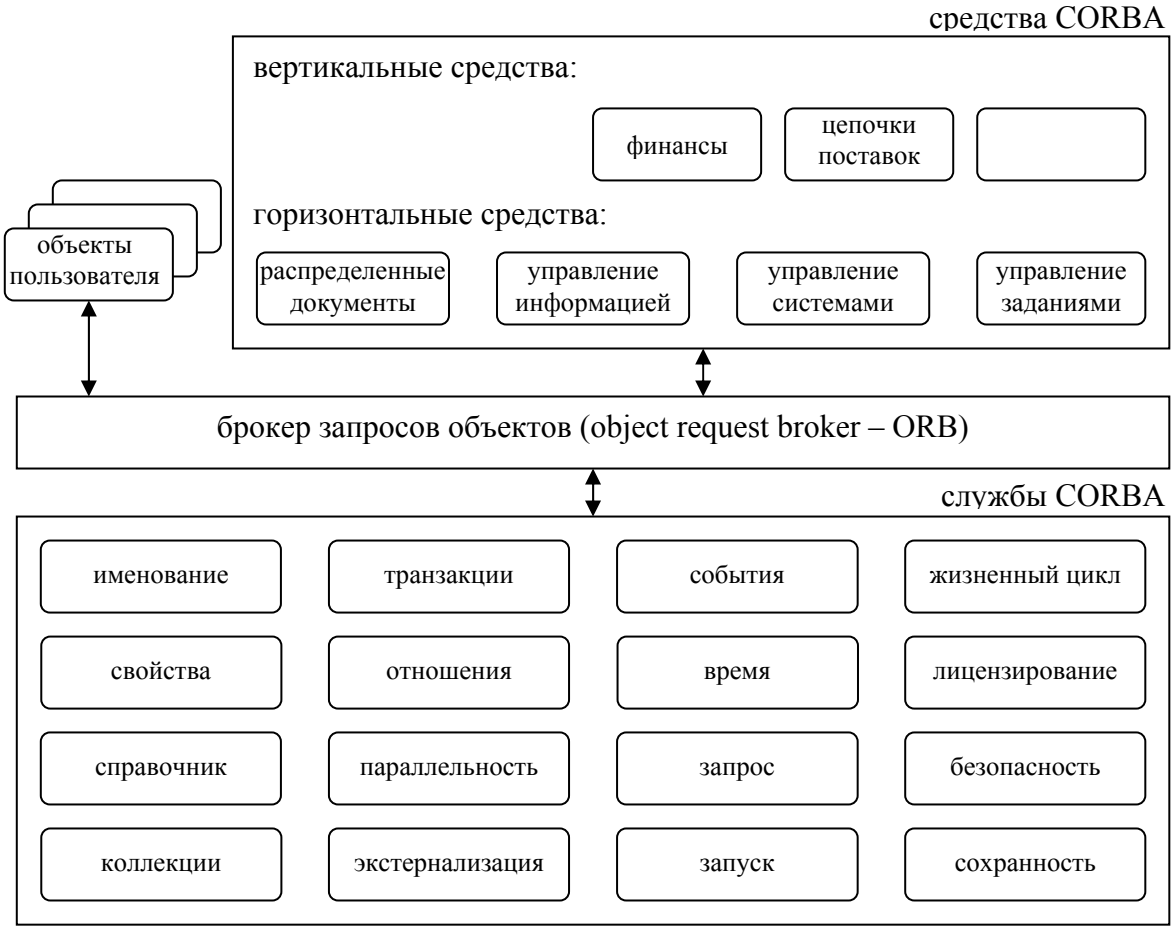


Рис. 2.14. Архитектура CORBA.

Система, подчиняющаяся спецификации CORBA, состоит из трех основных частей (*Рис. 2.14*):

- *брокер запросов объектов*, содержащий базовые функции взаимодействия объектов.
- *службы CORBA*, доступные с помощью стандартизованного прикладного программного интерфейса. В совокупности службы предоставляют функциональность, обычно необходимую большинству объектов, например, сохранность и управление жизненным циклом.
- *средства CORBA* – это набор средств и инструментов верхнего уровня, необходимых не индивидуальным объектам, а приложениям. Средства CORBA могут также включать службы, специфичные для отраслевых рынков, например, образования, здравоохранения или транспорта.

2.4.2.2. Работа CORBA

Чтобы к объекту можно было обратиться через брокер объектов, этот объект должен сначала объявить свой интерфейс, из чего клиенты узнают о методах, которые он предоставляет. Интерфейсы описываются на IDL.

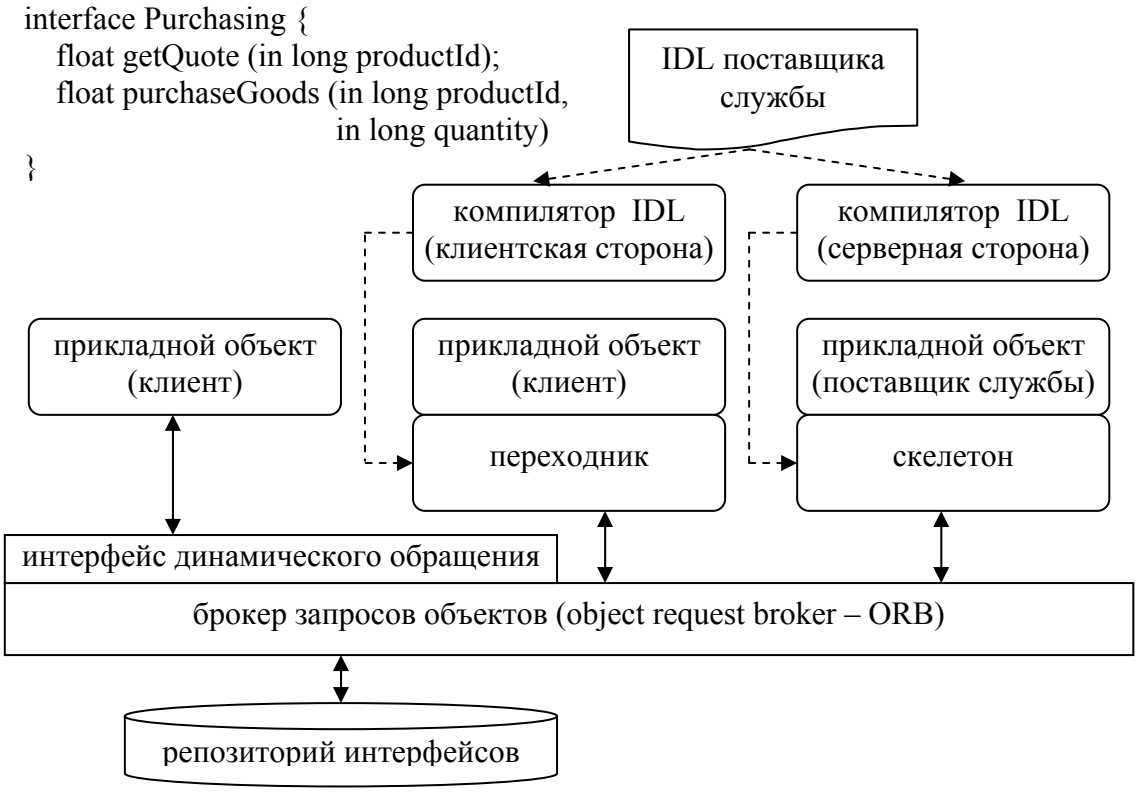


Рис. 2.15. Спецификации IDL транслируются в скелетоны на стороне сервера и в переходники на стороне клиента.

На *Рис. 2.15* показан пример спецификации IDL-интерфейса для объекта, обеспечивающего функции поддержки цепочек поставок товаров. В дополнение к описанию методов, в отличие от систем на базе RPC, язык IDL спецификации CORBA поддерживает множество объектно-ориентированных концепций, например, наследование и полиморфизм. Как и в случае RPC, спецификации, написанные на IDL, могут быть переданы компилятору с этого языка, который формирует заместителя объекта и скелетон. Заместитель объекта – это переходник, ответственный за сокрытие распределенности, его задача – представить вызовы не удаленными, а локальными. Программа заместителя содержит в себе описание методов, предоставляемых реализацией объекта. Для получения готового клиентского приложения, она должна быть загружена вместе с программой клиента. С другой стороны скелетон защищает от проблем распределенности сервер, поэтому сервер может разрабатываться так, как если бы вызовы к нему поступали из локального окружения. Как заместитель, так и скелетон могут быть написаны на любом из тех языков,

которые поддерживаются компилятором с языка IDL (например, спецификация CORBA 3 поддерживает трансляцию с IDL на Си, Си++, Java, Smalltalk, Аду, Кобол, Лисп, PL/1, Python и IDLScript).

Современные версии спецификации CORBA допускают и обратные отображения: например, в стандарте CORBA 3 предусмотрено проведение отображения записи интерфейсов на языке Java в записи интерфейсов на IDL. Обратное отображение позволяет программистам на языке Java создавать объекты, доступные из других приложений, написанных (возможно) на других языках программирования. Обработка программы на языке Java обратным компилятором позволяет получить эквивалентный интерфейс, написанный на IDL, имея который, можно построить (на языке Java или другом языке программирования) программу клиента CORBA, имеющую доступ к нужному объекту.

2.4.2.3. Динамический выбор и динамическое обращение к службе

Описанный выше механизм спецификации CORBA, призванный обеспечивать способность к взаимодействию, требует, чтобы клиент был статически привязан к интерфейсу. Однако модель RMI допускает динамическое обнаружение новых объектов и построение обращений к этим объектам в процессе работы, даже если для данного клиента не был создан никакой переходник. Эта возможность базируется на двух компонентах: *репозитории интерфейсов* и *интерфейсе динамического обращения*. Репозиторий интерфейсов хранит определения всех объектов, известных брокеру. Приложения могут использовать репозиторий для поиска, редактирования или удаления IDL-интерфейсов. Один брокер может иметь несколько репозиториев, и несколько брокеров могут иметь доступ к одному репозиторию. Единственное требование, поставленное в спецификации CORBA, заключается в том, что каждый брокер должен иметь хотя бы один репозиторий. Интерфейс динамического обращения дает доступ к операциям, которые могут использоваться клиентами для просмотра репозитория и динамического построения обращений к методам, базируясь на вновь обнаруженных интерфейсах.

Возможность динамического построения обращений к методам на основе динамически обнаруженных интерфейсов решает только часть проблемы динамического обращения к службе. Он предполагает, что клиенты уже идентифицировали нужную им службу. Для того чтобы это стало возможным, в спецификации CORBA ссылки на сервисные объекты выдаются только *службой именования* и *справочной службой*. Служба именования позволяет извлекать ссылки на объекты, отталкиваясь от их имени, а справочная служба дает возможность клиентам искать службы,

основываясь на их свойствах. Службы в свою очередь вносят сведения о своих свойствах в справочник. Используя справочную службу, клиенты могут искать не только объекты, реализующие тот или иной интерфейс, но также объекты, свойства которых имеют конкретные заданные значения.

2.4.3. Мониторы объектов

Свое название *мониторы объектов* получили от слияния брокеров объектов и транзакционных мониторов.

Часть проблем, с которыми столкнулись брокеры объектов, и в частности, CORBA, была связана с тем, что единственным настоящим нововведением, которое они внесли, была ориентация на объекты, как способ стандартизации интерфейсов различных систем и языков программирования. Спецификацию CORBA предполагалось проводить поверх традиционных систем. Многие службы CORBA были лишь специфицированы, и потребовалось много времени прежде, чем они были реализованы в коммерческих продуктах. Поскольку первые доступные коммерческие продукты обычно были реализованы вручную, то при сравнении с уже существовавшими системными платформами брокеры объектов оказывались чрезвычайно неэффективными, в них недоставало важной функциональности, например, возможностей работы с транзакциями.

Выходом оказалось использование транзакционных мониторов и других систем с промежуточными слоями, делавших их объектно-ориентированными (иногда совместимыми с CORBA). При использовании в подобном процессе транзакционных мониторов стали появляться мониторы объектов.

2.5. Распределенная обработка информации на основе обмена сообщениями

2.5.1. Системная поддержка на основе обмена сообщениями

Термин "*на основе обмена сообщениями*", относится к виду взаимодействия, при котором клиенты и поставщики служб взаимодействуют, обмениваясь структурированными наборами данных – сообщениями. Тип сообщения обычно определяется конкретной используемой системой, в настоящее время большинство систем используют типы языка XML.

Базирование системного программного обеспечения на основе обмена сообщениями – это прямое развитие идей систем очередей, заложенных в транзакционные мониторы. В последнее время системы на основе обмена сообщениями стали рассматриваться как наилучший выбор при реализации сетевых служб. Системы очередей сообщений создают

расширенную поддержку асинхронной сохранной связи. Смысл этих систем заключается в том, что они предоставляют возможность промежуточного хранения сообщений, не требуя активности во время передачи ни от отправителя, ни от получателя. Их существенное отличие от поддержки транспортного уровня состоит в том, что системы очередей сообщений обычно предназначены для поддержки обмена сообщениями, занимающего минуты, а не секунды или миллисекунды.

2.5.2. Модель очередей сообщений

При описании модели часто используются термины "клиент" и "сервер", но при обмене сообщениями это различие исчезает, по крайней мере, с точки зрения системной платформы. Разница между "клиентом" и "сервером" ("поставщиком службы") имеет чисто прикладной характер, и может ощущаться только теми, кто знаком с семантикой сообщений и процесса обмена ими. Это контрастирует с формами взаимодействия, описанными ранее, где объекты, действовавшие как клиенты, обращались к методам, предоставляемым другими объектами, действовавшими как серверы.

Модель очередей сообщений формирует основу, на которой разрабатываются многие полезные концепции и свойства, существенно упрощая разработку приложений, способных взаимодействовать друг с другом, и предоставляя поддержку для управления ошибочными ситуациями и системными сбоями. Среди таких концепций одна из наиболее важных абстракций является абстракция "*очередь сообщений*".

Основная идея, лежащая в основе систем очередей сообщений, состоит в том, что приложения общаются друг с другом путем помещения сообщений в специальные очереди. Очереди обычно идентифицируются по именам, обычно они связаны с определенным получателем. Эти сообщения передаются по цепочке коммуникационных серверов и, в конце концов, достигают места назначения, даже в том случае, если получатель в момент отправки был неактивен. Как только получатель освободится для обработки нового сообщения, он обратится к нужной функции системы МОМ и извлечет из очереди первое сообщение (*Рис. 2.16*). Очередь может быть прочитана только связанным с ней приложением, при этом несколько приложений могут совместно использовать одну очередь.

Важный момент в системах очередей сообщений состоит в том, что отправитель обычно в состоянии гарантировать только попадание сообщения в очередь получателя, не гарантируя времени попадания. Какие-либо гарантии, что сообщение будет прочитано и обработано получателем также не даются, это зависит только от самого получателя.

После того, как сообщение попало в очередь, оно будет оставаться в ней до удаления, независимо от того, активен его отправитель или его получатель.

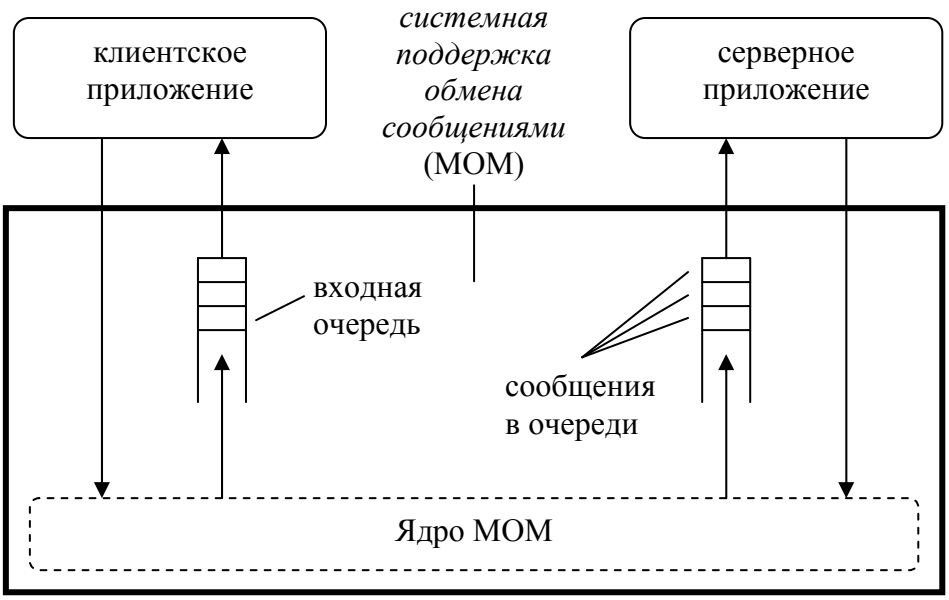


Рис. 2.16. Модель системы очередей сообщений.

Сообщения должны быть правильно адресованы и могут содержать любые данные. Обычно адресация осуществляется путем указания уникального системного имени очереди, в которую направляется сообщение. В некоторых случаях размер сообщения может быть ограничен, но базовая система в состоянии разбивать большие сообщения на части и собирать их обратно в единое целое абсолютно прозрачно для приложений.

Большинство систем очередей сообщений поддерживают процесс вставки дескриптора *функции обратного вызова*, которая автоматически вызывается при попадании сообщения в очередь. Обратные вызовы (сервер вызывает клиента) используются для автоматического запуска процесса, который будет забирать сообщения из очереди, если ни один такой процесс не был запущен заранее.

Слабая связанность отправителя и получателя сообщений имеет много преимуществ. Получатели имеют полную свободу выбора момента обработки сообщений. Извлечение сообщений из очереди может производиться только тогда, когда получатель может или должен их обработать. Важным следствием из этого является то, что системы очередей устойчивы к системным сбоям, поскольку от них не требуется поддерживать работоспособность в момент отправки сообщения. Если приложение выключено или не имеет возможности получать сообщения, они будут просто накапливаться в очереди сообщений, а после включения приложения в работу будут доставлены по назначению.

Помещенные в очередь сообщения могут иметь связанные с ними даты или временные интервалы действия. Если сообщение не извлекается из очереди до указанного времени, оно уничтожается. Очереди могут быть разделяемыми среди нескольких приложений. Этот подход часто используется, когда нужно иметь несколько приложений, обращающихся за одной и той же услугой, что позволяет распределить между ними нагрузку и повысить производительность. Система контролирует доступ к очереди и дает гарантию, что сообщение попадет только одному приложению.

2.5.3. Взаимодействие с системой очередей сообщений

Подход, применяемый в системах очередей, принципиально асинхронный. Обычно системы очередей предоставляют пользователям прикладной программный интерфейс, который удобен для использования в некотором конкретном программном окружении. Например, программисты, использующие язык Java, могут использовать стандартный интерфейс *Java Message Service* (*JMS*): отправители (получатели) сначала привязываются к очереди, то есть идентифицируют очередь, в которую они хотят послать (из которой хотят получить) сообщение, указывая имя очереди, а затем могут приступать к отправке (получению) сообщений.

JMS – это прикладной интерфейс, его реализация возлагается на пользователей или поставщиков готовых систем. Не все МОМ-системы совместимы со службой JMS, которая может быть реализована как отдельная система или модуль внутри сервера приложений.

2.5.4. Транзакционные очереди

В рамках абстракции *транзакционных очередей* системы очередей дают гарантию, что отправленное сообщение обязательно будет доставлено приложению получателя один и только один раз, даже если сама система обмена сообщениями выйдет из строя в период между объявлением о сообщении и его доставкой. Сообщения записываются в сохранную память и, следовательно, становятся снова доступными, когда система снова становится работоспособной.

В дополнение к гарантированной доставке транзакционные очереди обеспечивают защиту от сбоев. Получатель может объединять набор чтений и уведомлений в одном *атомарном* действии. Атомарное действие объединяет в себе набор операций, обладающий свойством "*все-или-ничего*": либо все операции набора выполняются успешно, либо не выполняется ни одна операция. В случае возникновения сбоя в момент, когда еще не все операции из атомарного набора выполнились, происходит

откат всех завершенных действий, что соответствует обратному помещению сообщения в очередь, так что оно может быть снова прочитано тем же приложением, либо каким-нибудь другим. С точки зрения отправителя транзакционность очереди означает, что сообщения, отправляемые внутри атомарного набора, записываются системой в сохранную память и становятся видимыми для доставки только тогда, когда выполнение атомарного набора операций заканчивается. Следовательно, откат уведомления о сообщении просто приводит к удалению сообщения из сохранной памяти.

2.6. Брокеры сообщений

При комплексной интеграции прикладных систем особенно важно автоматизировать взаимодействие цепочек поставок, то есть передач информации от одной прикладной системы другой. Практическая жизнь сложилась так, что системы автоматизации, базы данных, форматы данных у всех предприятий разные.

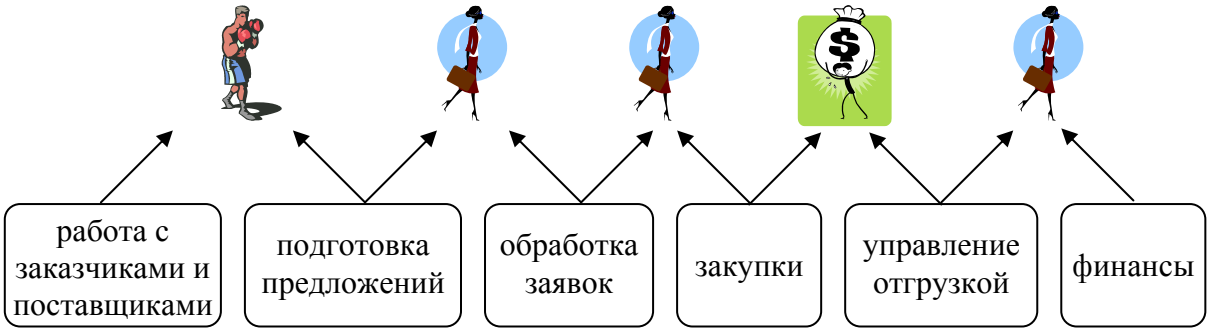


Рис. 2.17. Ручная реализация цепочки поставок, в которой люди действуют как участники эстафеты, на различных этапах, извлекая данные из одних систем, переформатируя их и помещая в другие системы.

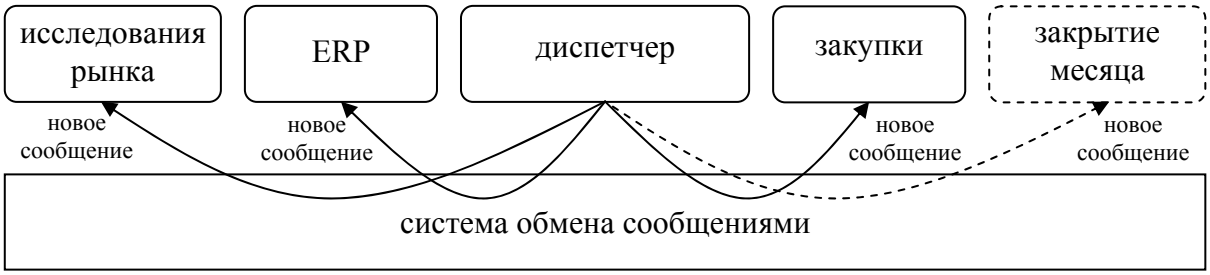


Рис. 2.18. Приложения, использующие для взаимодействия модель "точка/точка", должны быть изменены, если им потребуется начать взаимодействовать с новой системой (пунктир).

Автоматизация цепочки поставок – это сведение воедино всех таких разобщенных систем. Что еще хуже, при комплексной интеграции приходится сталкиваться со многими нетехническими проблемами: каждая включаемая в цепочку система обычно принадлежит и эксплуатируется разными подразделениями компании. Каждое подразделение управляется

автономно и выполняет много специфических функций, которые не всегда согласуются с тем, что должно получиться при интеграции. Однако если не автоматизировать цепочки поставок, многие операции на стыках систем будут выполняться вручную (*Рис. 2.17*).

Традиционные системы на базе RPC и системы очередей создают между приложениями соединения типа "*точка-точка*" (*Рис. 2.18*). Часто возникающая при их использовании проблема заключается в том, что при таком взаимодействии ответственность за определение получателя сообщения ложится на отправителя. В определенных случаях эта схема адресации становится трудно управляемой, поскольку число отправителей и получателей постоянно растет, а окружение, в котором работает система, становится более динамичным. Если отправителю потребуется начать взаимодействовать с новой системой, его приложение должно быть изменено. Снять подобные ограничения удается с помощью брокеров сообщений, которые действуют как посредники между системными сущностями. Брокеры сообщений обеспечивают гибкую маршрутизацию и другие необходимые для интеграции приложений качества. В сочетании с асинхронным обменом информацией – это наиболее пригодный подход к решению проблемы комплексной интеграции приложений отдельных предприятий. Брокеры сообщений забирают задачу определения пути доставки сообщения у отправителя и передают ее ядру системы (*Рис. 2.19*).

Имея брокер сообщений, пользователи могут выбрать такую прикладную логику, которая для каждого сообщения идентифицирует очередь, в которую сообщение должно быть доставлено. Это позволяет отправителям не указывать, кто будет получателем сообщения. Брокер сообщений, выполняя правила, заданные пользователем, сам определит получателя, что значительно упрощает схему взаимодействия прикладных компонентов системы и их внутреннюю структуру. Правила определения единственного из потенциальных получателей сообщения конкретного вида и содержания от конкретного отправителя сосредотачиваются в единственном месте системы, что облегчает процесс их модификации, если она потребуется.

Правила маршрутизации обычно кодируются на специальном языке, каждое правило содержит логическое условие, зависящее от значений данных, находящихся в доставляемом сообщении. Логика может определяться в брокере сообщений, либо на уровне очереди. Если она определяется в брокере, то относится ко всем сообщениям, которые затем соответствующим образом маршрутизируются. Если логика связана с

очередью, она определяет, в получении каких сообщений эта очередь заинтересована.

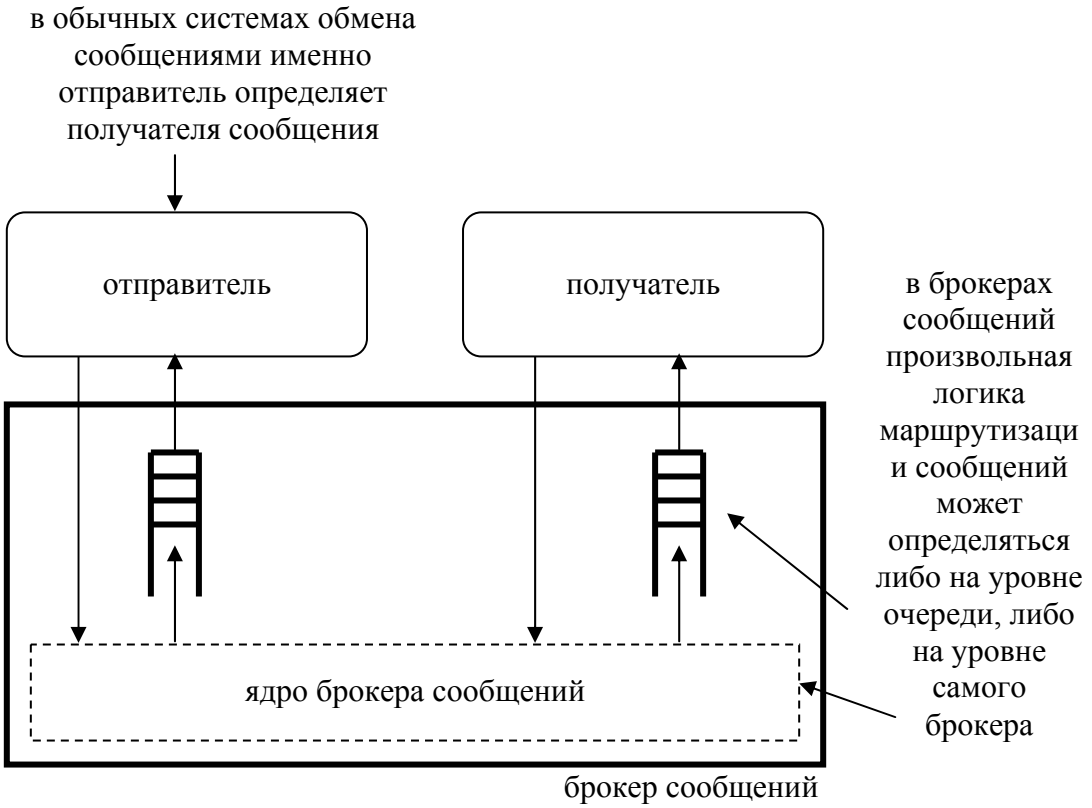


Рис. 2.19. Брокер сообщений дает возможность определять произвольную логику маршрутизации сообщений.

Таким образом, брокеры сообщений полностью развязывают отправителей и получателей сообщений. Отправители не указывают и не беспокоятся о том, какое приложение получит отправленное ими сообщение, получатели могут, а могут и не беспокоиться о том, какое приложение имеет возможность присылать им сообщения.

Обычные системы, построенные на обмене сообщений и работающие с общими (разделяемыми) очередями, тоже обеспечивают ограниченную форму разделения отправителей и получателей, поскольку в них приложения направляют сообщения в очереди, а не получателям. Однако в обычных очередях каждое сообщение доставляется не более чем одному приложению. Приложения, забирающие сообщения из общих очередей обычно однотипны (или являются разными ветвями одного процесса). Целью введения общих очередей является балансировка нагрузки. Общие, разделяемые очереди можно комбинировать с брокерами, получая и развязку и балансировку: сообщения будут доставляться в несколько очередей, в зависимости от логики маршрутизации (развязка), разные ветви или приложения могут затем

совместно извлекать сообщения из очередей и обрабатывать их (балансировка нагрузки).

Поскольку в системах обмена сообщениями (и в брокерах сообщений) взаимодействие приложений проходит через промежуточный слой, открывается возможность реализовать в этом слое еще больше функциональности для приложений, а не просто правила маршрутизации. Например, еще одной причиной привязки логики к очереди может быть потребность определить "правила преобразования содержания". Различные приложения, интегрируемые в единую систему, могут иметь разные форматы данных (могут, например, использоваться разные единицы измерения физических величин). Определяя правила преобразования содержания и ассоциируя их с очередью, можно проводить подобные преобразования в брокере, освобождая каждое приложение от выполнения такой работы. Приписывая разные преобразования разным очередям, их удается приспособить к потребностям разных приложений, не модифицируя.

2.6.1. Модель взаимодействия "публикация/подписка"

Благодаря управлению маршрутизацией, брокеры сообщений могут поддерживать различные модели взаимодействия, основанные на обмене сообщениями. Наиболее известной из них является парадигма "*публикация/подписка*": приложения взаимодействуют, обмениваясь сообщениями, характеризуемыми типом и набором параметров, но отправляющие сообщения приложения не указывают получателей. Вместо этого они просто *публикуют* сообщение в промежуточном слое, управляющем взаимодействием. По этой причине приложения, посылающие сообщения, называются "издателями". Если приложение заинтересовано в получении сообщений данного типа, оно должно *подписаться* в системе "*публикация/подписка*", регистрируя свой интерес. Как только издатель посылает сообщение данного типа, система извлечет список всех приложений, подписавшихся на сообщения этого типа, и доставит каждому из них по копии (*Рис. 2.20*).

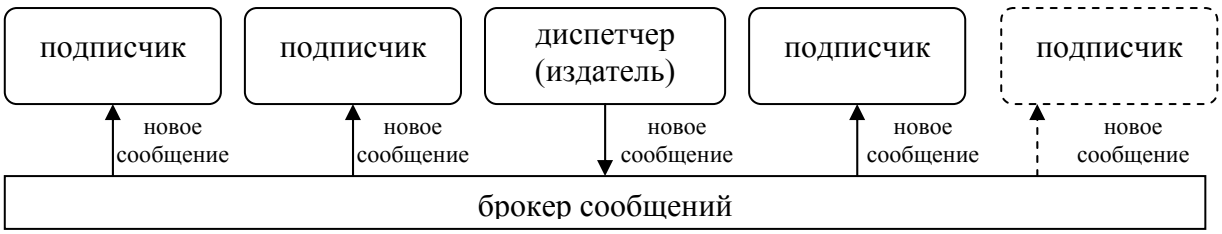


Рис. 2.20. Модели "публикация/подписка" повышают гибкость систем и их устойчивость к изменениям.

В модели "публикация/подписка" подписчики могут определять заинтересовавшие их сообщения двумя способами. Во-первых, они могут указывать тип сообщений (например, "Новый заказ"). В простейших случаях пространство именования типов довольно ограниченно и определяется символьной строкой. Более сложные системы допускают вводить структурные имена типов на основе иерархии типов/подтипов произвольной глубины. Используя структурированные типы, подписчики могут не только регистрировать свой интерес к сообщениям, имеющим некоторый тип, и подписываться на них, но также подписываться на сообщения, тип которых является прародителем в иерархии типов.

Вторая форма подписки основана на использовании параметров: подписчики специфицируют сообщения, которые они хотят получать, с помощью логических условий, вычисляемых над параметрами сообщений.

2.6.2. Распределенное администрирование брокера сообщений

В состав систем брокеров сообщений входит поддержка администратора, то есть выделенного пользователя, который имеет право определять:

(1) типы сообщений, которые можно отправлять и получать,
(2) пользователей, которым разрешено получать и/или получать сообщения и настраивать для себя логику маршрутизации.

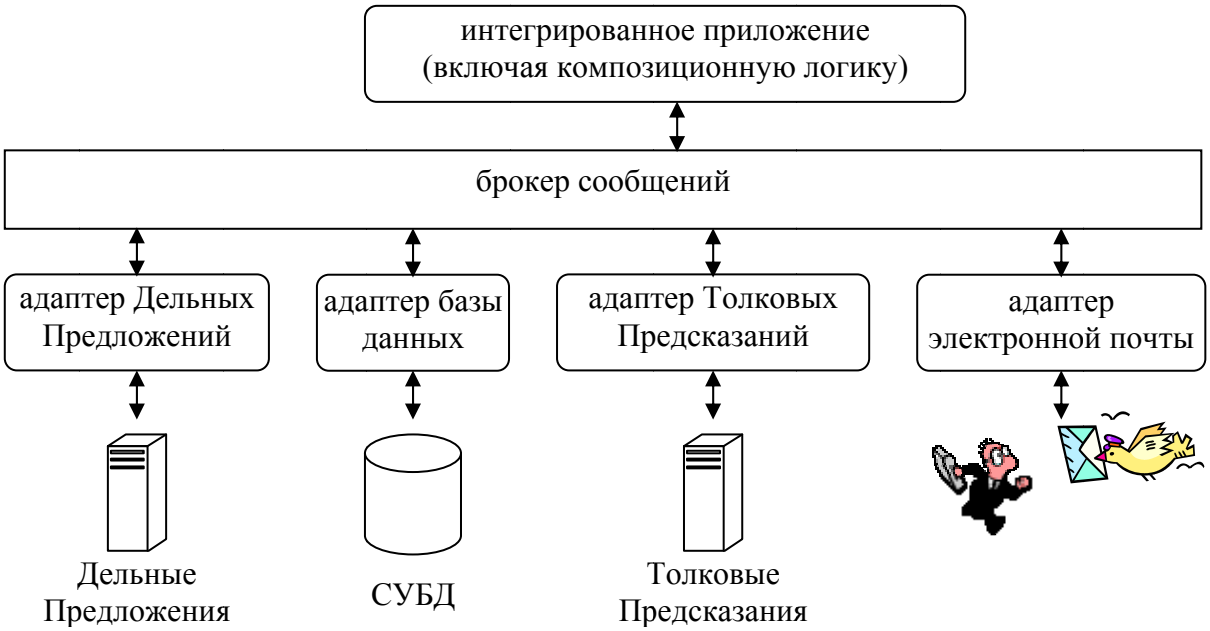
Администраторы присутствуют и в обычных системах обмена сообщениями, но в брокерах они более важны из-за слабой связи между отправителями и получателями, что, в общем случае, приводит к тому, что получатели не знают, какое приложение получит их сообщение. Системы "публикация/подписка", однако, разрешают издателям фиксировать ограничения на набор пользователей, которые могут получать определенные сообщения.

Архитектуры брокеров сообщений могут естественным образом приспосабливаться для удовлетворения потребностей расположенных в разных административных зонах приложений, ведущих интенсивный обмен сообщениями. Брокеры сообщений можно комбинировать. В такой архитектуре один брокер сообщений может быть клиентом другого. Если клиент хочет получить сообщение, посланное клиентом другого брокера, он подписывается у своего брокера, а тот, в свою очередь, подписывается на это же сообщение у другого брокера. С точки зрения брокеров другие брокеры выглядят точно так же, как и любые другие клиенты.

3. Основные виды прикладных систем

3.1. Комплексная интеграция приложений в рамках предприятия

Системы комплексной интеграции прикладных систем предприятий (*enterprise application integration, EAI*) – это эволюционный шаг в развитии системной поддержки, расширивший ее возможности по интеграции приложений. Современный интеграционные системы строятся с выраженной промежуточной платформой, что позволяет отделить слой прикладной логики от слоя управления ресурсами, добиться большей гибкости и органично интегрировать серверы.



3.1. Высокоуровневая модель архитектуры типичной системы интеграции приложений в рамках предприятия.

При комплексной интеграции прикладных систем особенно важно автоматизировать взаимодействие цепочек поставок, то есть передач информации от одной прикладной системы другой, которые обычно характеризуются большим разнообразием в используемых операционных системах, интерфейсах, форматах данных и моделях взаимодействия.

3.2. Модель комплексно интегрированного предприятия

Модель комплексно интегрированного предприятия базируется на двух фундаментальных компонентах (*Рис. 3.1*): адаптерах и брокерах сообщений. Адаптеры скрывают гетерогенность и формируют единый взгляд на внешний гетерогенный мир.

Брокер сообщений представляет собой инструмент для взаимодействия с адаптерами и, следовательно, с интегрируемыми системами. Некоторый ограниченный набор средств интеграции

функциональности приложений предлагают также транзакционные мониторы.

3.3. Системы управления рабочим потоком

3.3.1. Производственные рабочие потоки

Брокеры сообщений позволяют уменьшить влияние гетерогенности и распределенности систем автоматизации предприятий. Системы управления рабочим потоком (WfMS) пытаются преодолеть другую интеграционную проблему: они поддерживают непосредственно интегрирующие программы.

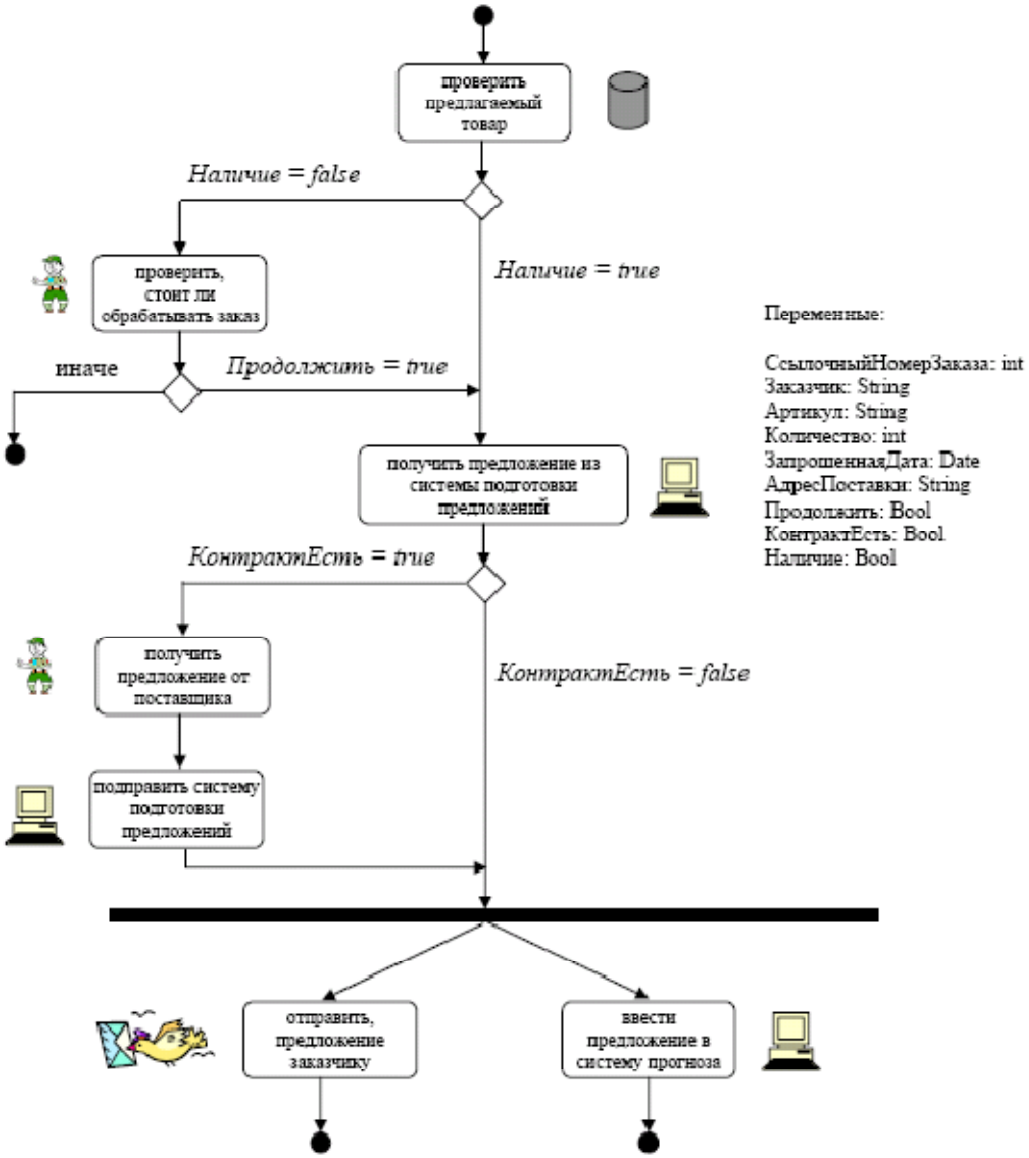


Рис. 3.2. Пример спецификации рабочего потока, моделирующего процесс подготовки предложений для покупателя.

Во многих отношениях системы производственного рабочего потока работают подобно системам интеграции приложений: они автоматизируют управление и поток данных между различными приложениями. Тем не

менее, интегрирующая часть ответственна за гетерогенность и поддержку способности к взаимодействию, а рабочий поток определяет бизнес логику, управляющую интеграцией. Это оказалось возможным благодаря тому, что системы WfMS описывали логику сложных приложений на языках высокого уровня (обычно графических), а не кодировали ее с помощью обычных языков программирования.

На *Рис. 3.2* показан рабочий поток, моделирующий процесс цепочки поставок, в частности, шаги, которые должен выполнить производитель некоторого товара, чтобы выставить предложение заказчику. Графическое представление основано на варианте диаграмм активности унифицированного языка моделирования (*Unified Modeling Language, UML*). С рабочим потоком связаны переменные, значения которых локальны для каждого его запуска. Переменные используются для обмена данными между рабочими узлами и для определения значений условий маршрутизации.

Рабочий поток выполняется *мотором*, который, по-существу, является планировщиком (*Рис. 3.3*): он составляет расписание работ, которые надо сделать, и приписывает работы соответствующим исполнителям (*ресурсам*). Систему не интересует, как ресурс выполняет заданную ему работу. В общем случае ресурсы обладают свободой выбора работ, которые им надлежит выполнять.

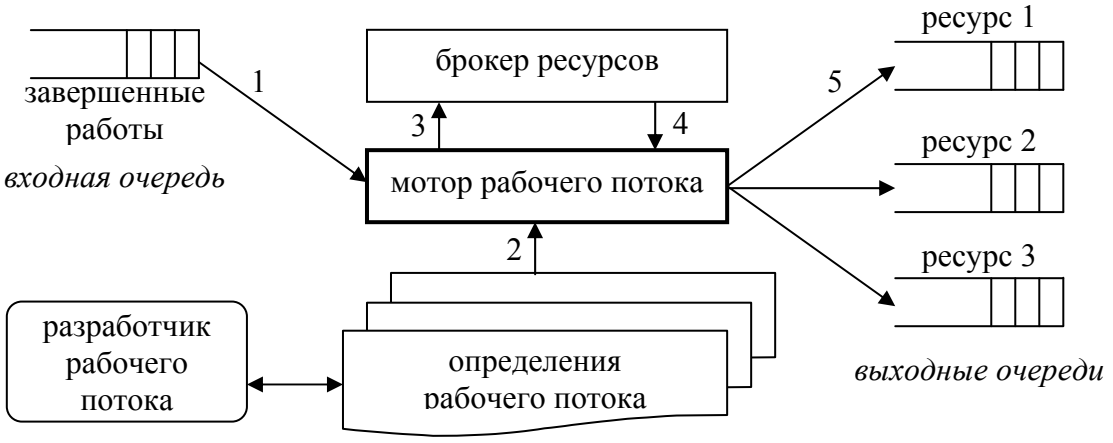


Рис. 3.3. Распределение работ в системе управления рабочим потоком.

Мотор непрерывно просматривает входную очередь для обработки сообщений рабочих узлов о завершении ими работы и для каждого сообщения во входной очереди проводит приписку ресурса.

3.3.2. Особенности рабочих потоков

Программирование рабочих потоков существенно отличается от программирования на обычных языках программирования. Главным отличием является масштаб работ, выполняемых процедурами. Рабочие

потоки обычно состоят из крупных структурных блоков активностей и приложений, которые могут работать часами или сутками. Другим важным отличием является степень модульности: системы рабочих потоков состоят из больших программных модулей, обычно целых приложений, и очень часто из сложных многоярусных систем.

При использовании рабочих потоков транзакционный откат части вычислений представляет собой серьезную проблему. Из-за длительности операций рабочего потока невозможно блокировать необходимые ресурсы базы данных, что может быть необходимо для поддержания возможности выполнения отката.

Еще одно отличие рабочих потоков от традиционного подхода связано с ресурсами, которые выполняют вызываемые процедуры. Обычно ресурсом считается вычислительная машина или ее компоненты. Для рабочего потока ресурс может быть разным в зависимости от выполняемой работы и от конкретного запуска потока, ресурсом может быть даже человек.

3.3.3. Интеграция рабочих потоков с другими системами

Технология рабочего потока по многим своим характеристикам напоминает технологию *транзакционного монитора*. Рабочие потоки, интегрируя крупные прикладные системы, могут выполнять распределенные транзакции, управлять именованием и связыванием ресурсов, обладают функциональностью по управлению производительностью и балансировкой нагрузки. Однако системы рабочих потоков фокусируются на динамическом выборе и на управлении исключительными ситуациями, используя преимущества объектно-ориентированных брокеров, систем обмена сообщениями, и моделей "публикация/подписка".

3.3.4. Достоинства и ограничения систем управления рабочим потоком

Функциональность систем управления рабочим потоком вполне достаточна для кодирования логики процессов при интеграции приложений, включая стыковки крупных приложений и работ, выполняемых вручную. Дополнительную помощь дает предоставляемое разработчиками графическое окружение разработки, которое помогает строить бизнес процесс, моделируя его на экране.

Системы рабочих потоков хорошо зарекомендовали себя при работе с циклическими, хорошо определенными процессами, но интеграция в

глобальной сети, как и интеграция существенно гетерогенных приложений, ими выполняется с трудом.

3.4. Серверы приложений

Серверы приложений эквивалентны традиционным системам поддержки распределенной обработки информации, отличаясь от них использованием глобальной сети, как основного канала доступа к службам системной поддержки.

Переход к сетевому доступу приводит к тому, что презентационный слой начинает играть гораздо более важную роль, чем в традиционных системах. Подготовка, динамическая генерация и управление документами стала значительной частью работы серверов приложений, которые расширяют возможности обычных систем доступом к глобальной сети.

Важнейшим свойством серверов приложений является аккумулирование внутри промежуточной платформы все большего объема функций. Если тенденция будет продолжена, станет непросто разобраться, что находится внутри сервера приложения, а что – нет. Во многих случаях имя, первоначально присвоенное серверу приложения, приобретает со временем новый смысл, обозначающее любой компонент промежуточного слоя, выпускаемый данной компанией (WebLogic, WebSphere).

3.4.1. Поддержка прикладного слоя

Содержащаяся в сервере приложений функциональность похожа на то, что предоставляется системами CORBA, транзакционными мониторами, брокерами сообщений. Это объясняет, почему серверы приложений не ограничиваются сетевой интеграцией, но могут использоваться и для интеграции прикладных систем предприятий (как обычных распределенных приложений, так и транзакционных). Их целью является создание единого окружения для всех видов прикладной логики, работающей с глобальной сетью, а также автоматическое получение функциональности серверов приложений (транзакции, безопасность, сохранность) при подключении приложения к данному серверу. Разработчикам приложения не требуется реализовывать всю эту функциональность самостоятельно, она становится доступной при подключении к серверу приложений.

В составе серверов приложений имеются различные виды компонентов, обеспечивающие различные методы управления состоянием и сохранностью (в рамках одной сессии взаимодействия или в рамках серии сессий с обеспечением сохранности данных). Допускается как синхронное, так и асинхронное взаимодействие с клиентом.

Компоненты серверов приложений могут создавать посредников между системными программами и приложениями, что позволяет предоставлять дополнительные услуги. Поддержка транзакционности освобождает разработчиков от определения транзакционных границ и реализации соответствующих программ. Другие службы контейнера включают обеспечение свойств сохранности и безопасности.

Привязка к серверу производится с помощью службы именования и каталогов, используя которую, клиенты могут привязываться к серверу по имени объекта. Каждое имя объекта действует в некотором контексте, внутри контекстов имена должны быть уникальными. Контексты могут строиться в иерархии, подобные иерархиям каталогов в файловых системах.

Серверы приложений работают также со слоем управления ресурсами. Обычно предлагается подход, основанный на использовании прикладных интерфейсов и стандартных архитектур, например сервер J2EE использует стандарты JDBC и J2CA. Такие стандарты определяют прикладные интерфейсы, дающие возможность доступа практически к любому источнику табличных данных, а также методы построения *адаптеров ресурсов*, то есть компонентов, дающих возможность единообразно подключать ресурсы к серверу и другим приложениям, что дает возможность поддерживать транзакционность и безопасность.

Серверы приложений предлагают и другие услуги, упрощающие администрирование и управление приложениями, обеспечивая высокую доступность и производительность. Ими может проводиться распределение нагрузки внутри наборов объектов, непрерывный контроль работы приложения и проведение перезапуска при сбоях. Проводится также администрирование объектов и безопасности: система следит за тем, какой пользователь обращается к какому приложению, и накладывает необходимые ограничения на этот доступ. То, что раньше могло делаться только вручную, теперь выполняется автоматически. Аналогичные свойства предоставляются (или разрабатываются) для других моделей распределенных объектов, например, для модели CORBA.

В настоящее время проводится работа по повышению производительности серверов приложений, которые пока не могут достичь показателей транзакционных мониторов. Но если транзакционные мониторы работают в приложениях с высокой нагрузкой, но статичных (при этом фаза их разработки может требовать больших затрат усилий и времени), то серверы приложений предназначены для того, чтобы сильно облегчить и упростить именно фазу разработки.

3.4.2. Поддержка презентационного слоя

Серверы приложений развивают возможности традиционных систем, с их помощью постепенно делаются все более сложные реализации, в которых передача информации между клиентом и сервером становится все более эффективной, гибкой и управляемой. Современный сервер приложений поддерживает различные типы клиентов.

Сетевые навигаторы. Программы сетевых навигаторов в настоящее время являются наиболее общим типом клиента. Они взаимодействуют с сервером приложения по протоколам HTTP или HTTPS и получают статически или динамически сформированные страницы HTML. Если клиентом является не навигатор, а аплет, взаимодействие с сервером приложения может происходить по другим протоколам (не HTTP).

Могут существовать и другие виды клиентов, подключающиеся через глобальную сеть. Примером могут служить ***различные современные устройства***, например, мобильные телефоны, в которых вместо HTTP используется протокол WAP, а языком презентации служит язык WML. Тем самым, разработчикам не требуется писать разные программы для разных клиентов, а некоторые особенно развитые инструменты позволяют динамически генерировать документы на различных языках разметки и проводить автоматическое конвертирование с одного языка на другой.

Электронная почта, как и другие устройства, имеет свой собственный протокол (SMTP) и свой собственный формат разметки. Серверы приложений поддерживают упаковку информации и ее доставку поверх SMTP.

Прикладные программы взаимодействуют с серверами приложений почти так, как это делают аплеты.

Клиент сетевой службы использует уникальный протокол SOAP, а также другие языки и инфраструктуры. Серверы приложений поддерживают взаимодействие с клиентами сетевых служб, обеспечивая создание, просмотр и проверку правильности документов, написанных на языке XML, а также упаковку и распаковку сообщений, доставляемых по протоколу SOAP.

3.5. Сетевые технологии для интеграции приложений

Первоначально большинство промежуточных платформ были спроектированы для работы в отдельной локальной сети. Однако впоследствии возникла необходимость взаимодействия (обмен транзакциями) различных промежуточных платформ друг с другом. Основная идея такого электронного обмена заключалась в полной автоматизации взаимодействия между предприятиями.

Технически электронное взаимодействие означает вызов службы, размещенной в другой компании. Этот подход был продемонстрирован при реализации спецификации CORBA, которая поддерживает доступ к удаленным объектам, размещенным внутри некоторого *домена*, то есть под управлением одного брокера объекта. Расширение на Интернет достигается соединением нескольких брокеров друг с другом. Делается это посредством обобщенного межброкерного протокола *GIOP*, определяющего, как вызовы одного брокера передаются другому и как отправляется ответ на вызов. Этот протокол был расширен до межброкерного протокола Интернета *IIOP*, в котором определено, как транслировать вызовы протокола GIOP в вызовы TCP/IP.

Однако брокеры часто соединены с Интернетом через межсетевые экраны, которые сильно ограничивают взаимодействие. Другим препятствием, которое мешает простым расширениям протоколов в виде соединения двух брокеров, является различие определений интерфейсов и форматов данных, применяемых в двух приложениях. Для помощи в поисках вызываемых служб также необходимо иметь справочный сервер.

Межсетевой экран – это барьер на пути нежелательного сетевого трафика, который блокирует многие коммуникационные каналы, в том числе почти все виды взаимодействия, предлагаемые традиционными продуктами интеграции приложений. Рассчитывать на использование протоколов GIOP/IIOP в распределенных системах удаленных вызовов процедур (RPC) и обращений к методам (RMI) нельзя.

При обычной интеграции приложений подобные проблемы не возникают, благодаря работе всех компонентов в единой сети, либо в нескольких сетях, соединенных прозрачным шлюзовым механизмом (*Рис. 3.4*). Основным принципом разработки в таких случаях является наличие доверительной зоны. Точно также, межсетевые экраны не являются проблемой в системах, основанных на обмене сообщениями. Протоколы, подобные простым почтовым (SMTP), могут проникать через экраны, поскольку ответственность за безопасное взаимодействие возлагается на программное обеспечение, обрабатывающее сообщения.

Межсетевые экраны меняют подходы к разработке систем. Во-первых, на основе тех протоколов, с которыми распространяются существующие промежуточные платформы, невозможно осуществлять прямое взаимодействие интегрируемых систем. Во-вторых, из-за наличия межсетевых экранов применяется *туннелирование*, суть которого заключается в том, что протоколы, которые могли бы быть заблокированными межсетевыми экранами, прячутся под протоколами,

которые этими экранами разрешаются. При туннелировании под протоколом HTTP некоторая посредническая программа упаковывает исходное сообщение в документ на языке HTML или XML, посылает документ в соответствии с протоколом HTTP, а затем после прихода документа к получателю извлекает сообщение из полученного документа, что позволяет общаться, не обращая внимания на межсетевые экраны, которые могли бы им помешать (*Рис. 3.5*).

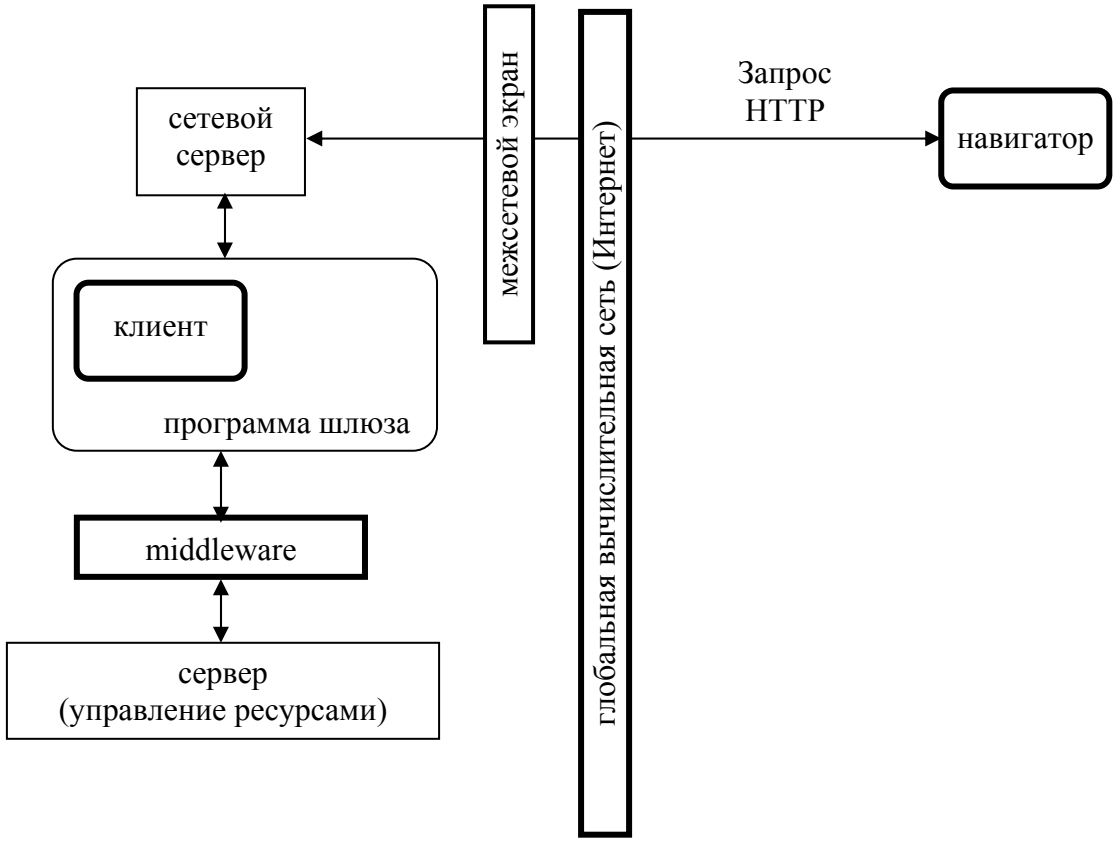


Рис. 3.4. Программы общего шлюзового интерфейса как способ взаимодействия с приложением, расположенным на серверной стороне.

В настоящее время туннелирование – это стандарт де-факто для решения проблемы преодоления межсетевых экранов. Оно используется не только традиционными промежуточными системами (например, GIOP/IIOP поверх HTTP), но и современными сетевыми службами (туннелирование удаленных вызовов процедур с помощью протокола SOAP поверх HTTP). Протокол HTTP наиболее часто используется как основа туннелирования, это один из очень немногих протоколов, которые проходят сквозь межсетевые экраны.

Возможность использования протоколов FTP, SMTP и HTTP для автоматического обмена сообщениями между приложениями поставила задачу выработки *единого синтаксиса и семантики* для данных, которыми приложения должны обмениваться.

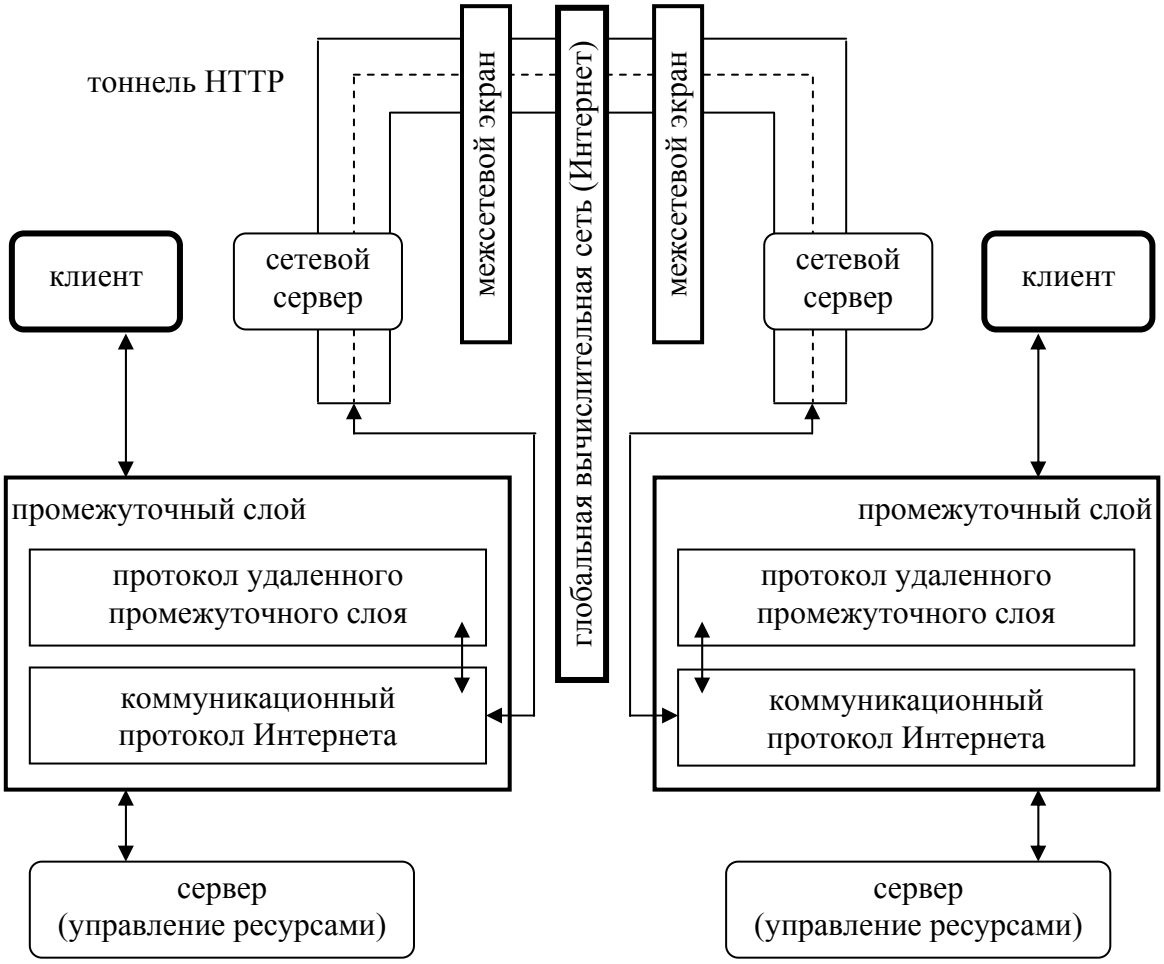


Рис. 3.5. Взаимодействие независимых прикладных систем на промежуточном уровне с туннелированием.

В традиционных системах представление данных скрыто в языке IDL, который выполняет две задачи: определение интерфейса и введение промежуточного машинно-независимого представления данных, преодолевающего различия между вычислительными архитектурами. В настоящее время широко используется язык разметки *XML* (*Extended Mark-up Language*), ориентированный на описание синтаксиса представления данных и предоставляющий стандартные правила определения структуры документов, пригодной для автоматического разбора.

4. Сетевые службы

4.1. Определение сетевых служб

Очень часто сетевой службой называют ***приложение, которое может быть доступно другим приложениям через Интернет***. Более точное определение может быть таким: ***самодостаточное модульное бизнес приложение, имеющее открытый стандартизованный интерфейс, ориентированный на Интернет***. Упор делается на согласованность со стандартами Интернета, от службы требуется открытость, то есть интерфейс, посредством которого она становится доступной в Интернете, должен быть опубликован. Однако некоторые вопросы остаются (что такое самодостаточность, модульность?).

Консорциум World Wide Web (W3C) дает такое определение: ***программное приложение, идентифицируемое с помощью универсального ресурсного идентификатора, интерфейс и способ связывания которого могут быть определены, описаны и выявлены как артефакты XML. Сетевая служба поддерживает прямое взаимодействие с другими программными агентами, используя обмен XML-сообщениями на базе Интернет-протоколов***.

Этим определением поясняется, что значит доступность службы (определение, описание, выявление), а что значит ее ориентированность на Интернет. Здесь же указывается, что сетевая служба должна быть "службой" в смысле, похожем на трактовку этого термина традиционными платформами промежуточных слоев. Она должна быть достаточно точно описана, чтобы для связывания и взаимодействия с ней можно было писать клиентские программы. Сетевые службы могут интегрироваться в более сложные распределенные приложения. Консорциум W3C также настаивает на том, чтобы в определение сетевых служб был включен язык XML. Этот язык в настоящее время также широко распространен, как и протокол HTTP, и сетевые серверы.

4.2. Сетевые службы и интеграция приложений

Проблемы автоматизации связей между предприятиями похожи на проблемы интеграции систем внутри отдельных предприятий, однако методы решения отличаются. Во-первых, совсем не очевидно, где может быть помещен слой промежуточного программного обеспечения, нужный для организации связей между предприятиями. Применение традиционного подхода потребовало бы от участников взаимодействия достичь соглашения по использованию и совместному управлению конкретной платформой, а также по реализации "глобального рабочего потока" (*Рис. 4.1*).

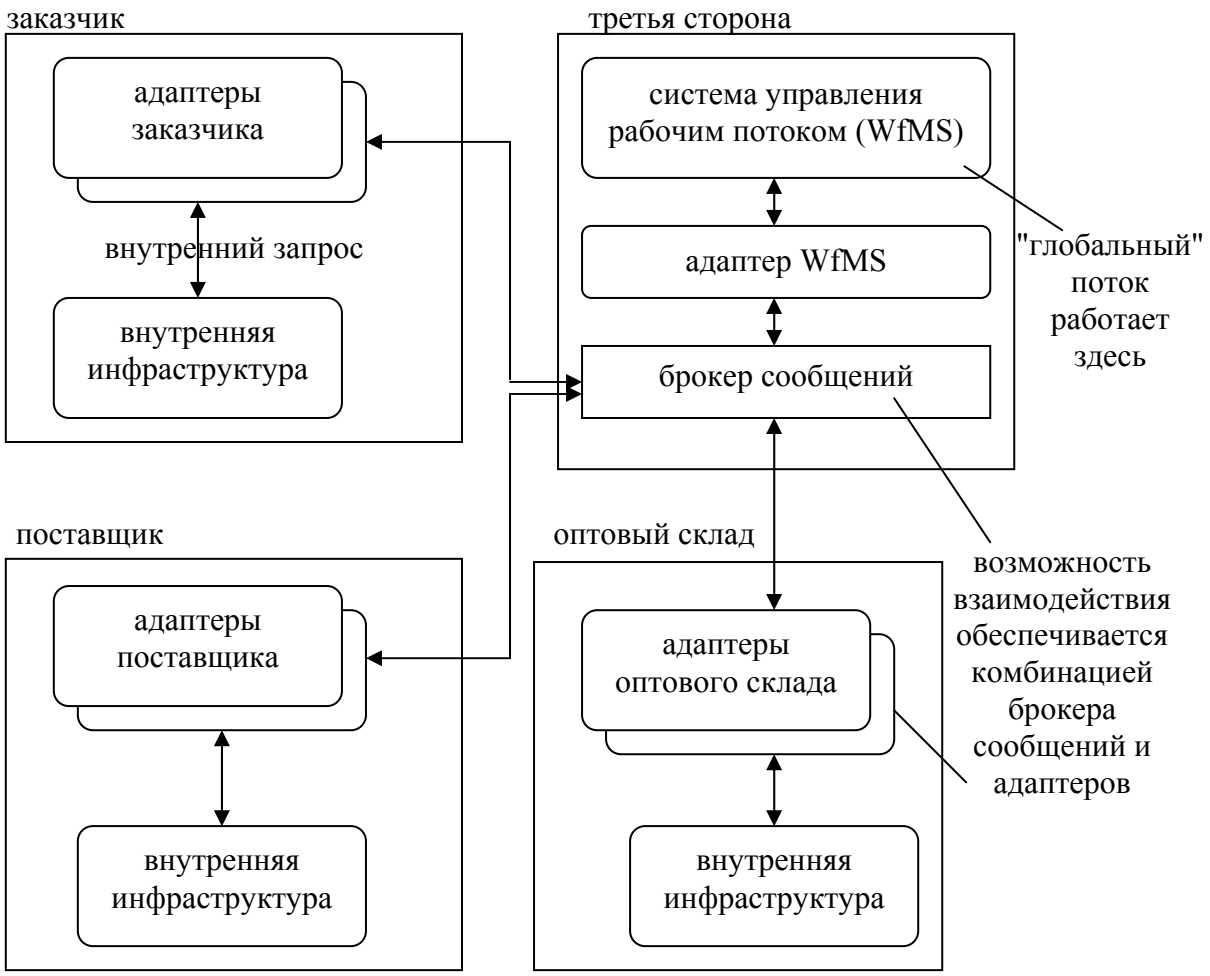


Рис. 4.1. Интеграция приложений различных предприятий выполняется так же, как и интеграция систем внутри отдельного предприятия. Технически возможная, она редко применяется на практике из-за недостатка доверия и необходимости большей автономности.

В некоторых случаях этот подход действительно можно применять, но достичь необходимого уровня доверия между предприятиями нелегко. Альтернативой могла бы быть организация взаимодействия по принципу "точка-точка", при которой проблемы интеграции различных партнеров решаются порознь. При этом подходе также требуется согласовывать используемые протоколы и инфраструктуры. Например, два предприятия могут установить у себя один и тот же брокер сообщений и организовать между собой связь, решая проблемы поддержки интеграции через Интернет (преодоление межсетевых экранов). Никакая третья сторона в такое взаимодействие не вклинивается, конфиденциальность не нарушается, транзакции попадают только к своим адресатам. Однако обычно взаимодействие предприятия происходит со многими его партнерами, поэтому для каждого из них на промежуточном слое потребуется устанавливать отдельное программное обеспечение. Тем самым возникает существенно гетерогенное окружение (*Рис. 4.2*).

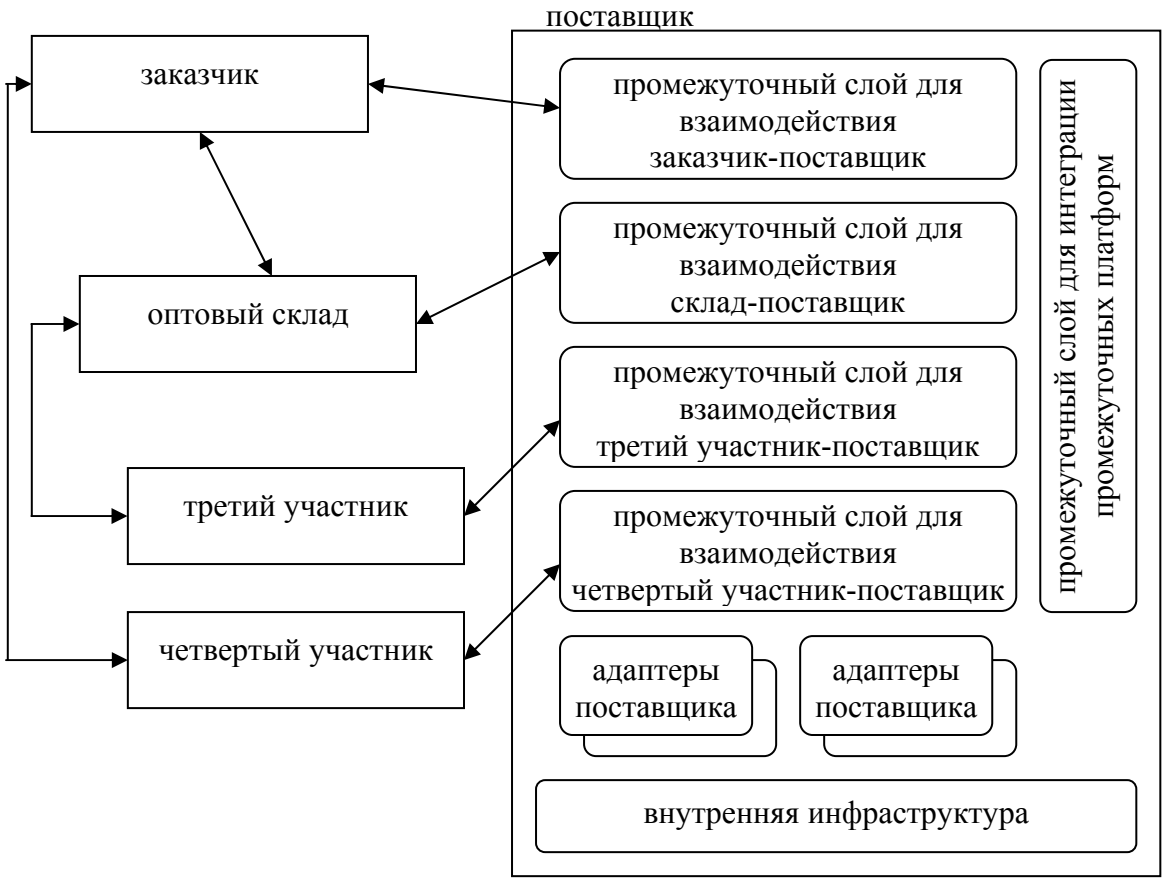


Рис. 4.2. Отсутствие центральной промежуточной платформы приводит к необходимости взаимодействия по принципу "точка-точка", при этом различные стороны (возможно) взаимодействуют с использованием разных программных платформ.

Другой причиной непригодности традиционного подхода является неверность многих предположений, делавшихся при внутренней интеграции приложений предприятий. Одно из таких предположений заключалось в том, что многие акты взаимодействия внутри предприятия обычно являются короткими, в то время как взаимодействия с внешними партнерами могут требовать более длительных транзакций (часов или дней). Длительность взаимодействия препятствует применению традиционных протоколов типа 2PC, поскольку они блокируют ресурсы на слишком большой срок, делая невозможным параллельное выполнение других операций.

Развитие глобальной сети привело к появлению и массовому принятию новых стандартов, протоколов (HTTP), форматов (XML). Однако сами по себе эти стандарты не решили всех проблем. Для интеграции приложений на новом уровне потребовалось развить архитектуру программных систем, основанную на понятии "службы", перестроить протоколы взаимодействия интегрирующих слоев и разработать новые дополнительных стандарты.

В терминах программного обеспечения ***служба – это процедура, метод или объект со стабильным общеизвестным интерфейсом***, который используется клиентами службы для обращения к ней (одна программа вызывает другую).

С точки зрения использования сетевые службы не отличаются от служб программных систем промежуточных слоев, за исключением того, что к ним можно обращаться через Интернет и из других предприятий. Службы являются слабо связанными системами, поскольку в общем случае они определяются, разрабатываются и управляются разными компаниями. Развитие такого подхода приводит к тому, что все начинает восприниматься как служба, а различные службы автономны и не зависят друг от друга.

Не все, что доступно через Интернет, представляет собой сетевую службу. Сетевая служба – это не набор страниц в Интернете, а программное приложение с общеизвестным и стабильным программным интерфейсом.

Традиционные протоколы (например, 2PC) проектировались в предположении, что работа будет осуществляться внутри предприятий. Эти протоколы предполагали наличие центрального транзакционного координатора, который обладает возможностями блокировать ресурсы по собственному усмотрению. Протокол подтверждения 2PC должен быть модифицирован, чтобы приобрести возможность работать в полностью распределенном окружении и более гибко проводить блокировки ресурсов. Аналогичным образом требуется переработать все протоколы взаимодействия и координации, улучшая и другие свойства системного программного обеспечения, например, надежность.

Ключом к возможности внедрения сетевых служб является стандартизация, которая и раньше помогла решить много проблем, возникавших при разработке традиционных программных платформ. Но для сетевых служб стандартизация не просто выгодна, а необходима. Наличия архитектур, ориентированных на работу со службами, и протоколов, пригодных для взаимодействия в Интернете, не всегда достаточно, чтобы решить все проблемы интеграции приложений, если все эти языки и протоколы не будут стандартными и всеми принятыми.

При интеграции приложений в Интернете с помощью сетевых служб каждая сторона представляет свои внутренние операции как некоторую (сетевую) службу, являющуюся точкой входа в локальную информационную систему. Работа независимых предприятий осуществляется в режиме равноправного взаимодействия. Все обмены

информацией проводятся на основе единых стандартизованных протоколов, разработанных так, чтобы децентрализованно обеспечивать все свойства традиционных протоколов. Сетевые службы сами исполняют эти протоколы и скрывают от программистов все сложные проблемы интеграции приложений (*Рис. 4.3*).

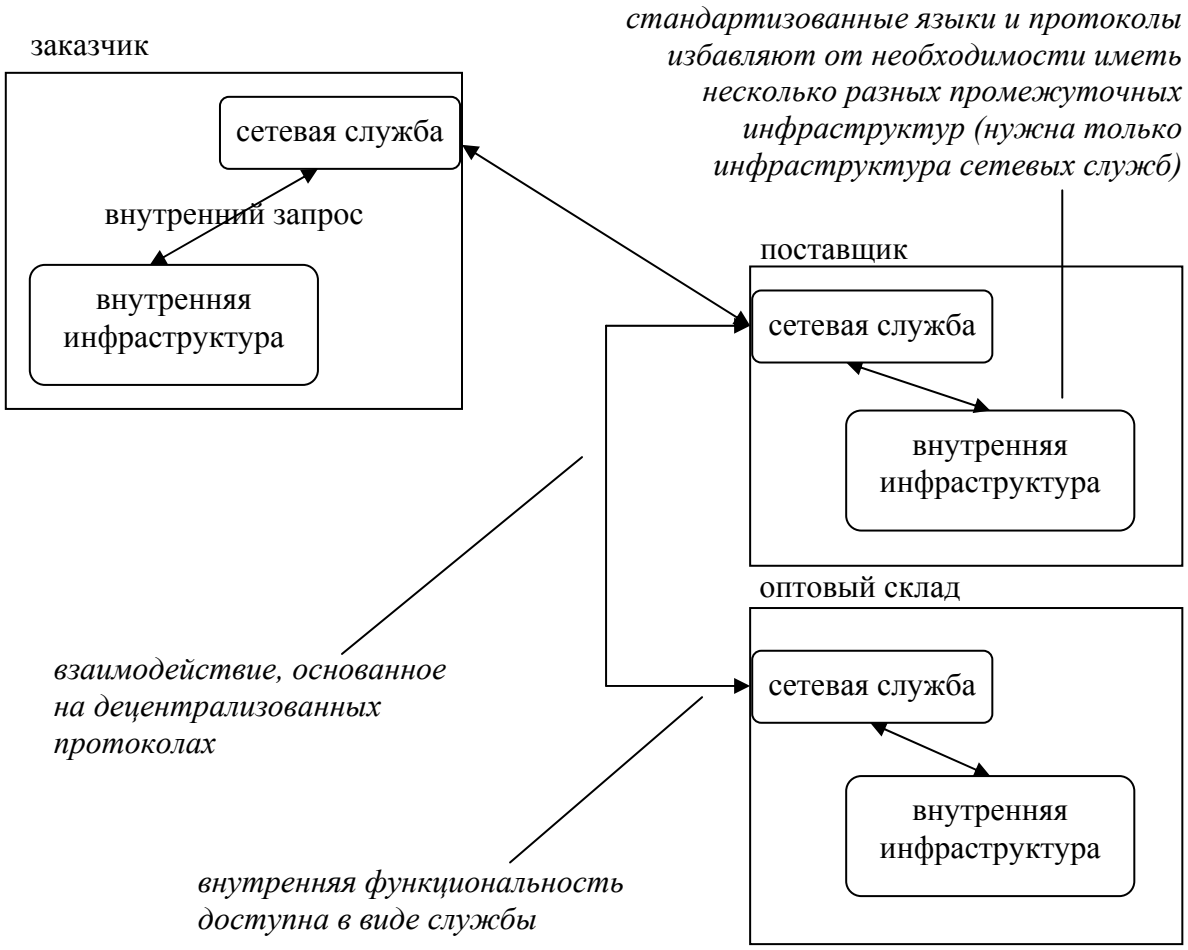


Рис. 4.3. Архитектура, ориентированная на сетевые службы, модернизированные (децентрализованные) протоколы и стандартизацию.

При разработке сетевых служб всегда имеется в виду, что эти службы представляют собой точки входа в локальные системы программного обеспечения. Поэтому основное использование сетевых служб заключается в представлении (через интерфейс сетевой службы) функциональности внутренних систем. В некотором смысле сетевые службы это аналог сложных оболочек, которые инкапсулируют приложения, создавая для них единый интерфейс и обеспечивая доступ к ним через Интернет.

Оболочки и адаптеры скрывают гетерогенность систем и являются ключами к интеграции приложений. С точки зрения клиента интеграции подлежат именно оболочки, поскольку только их может видеть интегрируемое приложение.

Гомогенность компонентов существенно снижает трудность интеграции. Это же относится и к сетевым службам, которые подчиняются единым стандартам. Сетевые службы создают фундамент, на котором можно строить программное обеспечение, поддерживающее интеграцию приложений в Интернете.

4.3. Основные технологии сетевых служб

4.3.1. Описание и поиск служб

Описание программной службы в традиционном понимании основывается на интерфейсе и языке описания интерфейса (IDL). Спецификации на IDL нужны для автоматической генерации переходников и установления основы для динамического связывания. Семантика различных операций, порядок, в котором к ним надо обращаться, и другие (возможно нефункциональные) свойства служб предполагаются известными разработчикам клиентских программ заранее. Это разумно, поскольку клиенты и серверы разрабатываются одними коллективами программистов. Кроме того, интеграционные платформы неявно определяют и вводят ограничения на многие аспекты описания служб и процесса связывания, которые поэтому не требуется считать частью описания службы. В сетевых службах такой неявный контекст отсутствует, поэтому описания должны быть точнее.

При описании сетевых служб применяется стек языков описания, в котором каждый элемент, расположенный на более высоком уровне, использует и уточняет описание, представленное элементами нижнего уровня.

- ***Общий базовый язык***. Самой первой проблемой, которую необходимо решать, это определение общего метаязыка, который можно было бы использовать как основу для спецификации всех языков, необходимых для описания различных аспектов служб. Для этих целей используется язык XML, который одновременно и широко распространен в качестве стандарта, и имеет достаточно гибкий синтаксис, позволяющий определять языки описания и протоколы служб.
- ***Интерфейсы***. Языки описания интерфейсов находятся в основе любой парадигмы, ориентированной на службы. Описания интерфейсов сетевых служб напоминают описания на языке IDL спецификации CORBA, хотя имеются и отличия, в частности, разрешается описывать типы систем с помощью схем XML. Кроме того, при описании службы необходимо указывать ее адрес и транспортный протокол (например, HTTP), иначе будет нельзя вызвать операции службы. В настоящее

время наиболее вероятным представляется использование языка описания сетевых служб WSDL.

- ***Бизнес протоколы***. Сетевая служба обычно предлагает несколько операций, которые должны вызываться клиентом в определенном порядке. Такие обмены между клиентами и службами называются *разговорами*. Набор правил, регулирующих разговоры, называется бизнес протоколом, поддерживаемым службой, что отличает их от коммуникационных протоколов. Для точного описания службы одного интерфейса недостаточно, и бизнес протоколы необходимы. Однозначного выбора стандарта описания бизнес протоколов к настоящему времени не произошло. Широко используются язык WSCL и язык BPEL.
- ***Свойства и семантики***. Традиционные интегрирующие платформы не содержат в описаниях своих служб ничего, кроме функциональных интерфейсов, но разработчики при связывании служб имеют возможность привлекать другую информацию, кроме того, традиционные службы являются тесно связанными компонентами. При описании автономных и слабо связанных сетевых служб надо учитывать, что клиенты принимают решения об использовании службы только на основании их описаний. В эти описания могут вставляться нефункциональные свойства (стоимость, качество и т. д.). Такая информация очень важна для служб, но она не входит в то, что обычно считается частью интерфейса. Дополнительная информация о службе присоединяется к ее описанию спецификацией универсальных средств описания, обнаружения и интеграции UDDI (*Universal Description, Discovery, and Integration*), в которой показано, как организуется информация о сетевой службе и как эта информация может регистрироваться и запрашиваться.
- ***Вертикальные стандарты***. Языки и свойства, описанные ранее, имеют обобщенный характер. Они не стандартизуют ни содержание службы, ни ее семантику (то есть смысл конкретных параметров или результат выполнения конкретной операции). Для того, чтобы определить конкретные интерфейсы, протоколы, свойства и семантики, предлагаемые сетевыми службами в конкретных приложениях, необходимы вертикальные стандарты. Например, стандарты RosettaNet описывают автоматизированный обмен коммерческой информацией. Эти стандарты дополняют ранее описанные слои, но связаны с определенными приложениями, они уточняют способы использования стандартного инструментария для управления обменами. Ими

руководствуются при написании клиентских приложений, которые могут осмысленно взаимодействовать с любой сетевой службой, совместимой с данными вертикальными стандартами.

После того, как служба правильно описана, она должна стать доступной для всех, кто ею интересуется. Для этого описания службы заносятся в справочник. Справочники позволяют разработчикам регистрировать новые службы, а пользователям отыскивать их. Поиск службы может осуществляться во время разработки или динамически. Ведением справочников заведуют либо доверенные организации (централизованный подход), либо произвольные компании, если централизованных серверов нет. В любом случае для взаимодействия с единой справочной службой или для взаимодействия между локальными справочниками необходимы протоколы и программные интерфейсы. Спецификация UDDI определяет стандартный прикладной интерфейс для опубликования и поиска информации в справочниках служб.

4.3.2. Взаимодействие служб

Взаимодействие служб проходит под управлением серии стандартов, определяющих это взаимодействие на разных уровнях. Каждый уровень характеризуется одним или несколькими протоколами, которые применимы ко всем сетевым службам, поэтому они реализуются в промежуточном слое сетевых служб. Протоколы взаимодействия невидимы для разработчиков, также как взаимодействия, проходящие между объектами CORBA, что позволяет сосредотачиваться на бизнес логике.

- ***Транспортные протоколы***. Эти протоколы скрывают от сетевых служб коммуникационные сети. Наиболее часто используется протокол HTTP.
- ***Сообщения***. На базе транспортных протоколов можно определять форматы и методы упаковки информации. Для сетевых служб используется протокол доступа к объектам (*Simple Object Access Protocol, SOAP*). Этот протокол задает шаблон обобщенного сообщения. Для указания конкретных свойств над протоколом SOAP делаются дополнительные надстройки. Например, протокол *WS-Security* описывает, как с помощью протокола SOAP осуществлять безопасные обмены.
- ***Инфраструктура протоколов (метапротоколы)***. Бизнес протоколы связаны с конкретными приложениями, но многое из нужной им поддержки может быть реализовано обобщенными инфраструктурными

компонентами. Эти компоненты могут, например, хранить сведения о состоянии разговора между клиентом и службой, ассоциировать сообщения с теми или иными службами, верифицировать сообщения на соответствие правилам, определенным в протоколах, выполнять метапротоколы, которые создаются с целью координации выполнения бизнес протоколов. Например, перед началом фактического взаимодействия клиенты и службы должны выбрать протокол, назначить ответственного за его выполнение и так далее. Эти метапротоколы, а также способы использования языка WSDL и протокола SOAP определяются в спецификации *WS-Coordination*.

- ***Промежуточные (горизонтальные) протоколы*** Сетевые службы и поддерживающая их инфраструктура по своей природе имеют распределенный характер, и их свойства, выходящие за рамки базовых коммуникационных свойств, реализуются с помощью децентрализованных протоколов. Эти протоколы называются горизонтальными, поскольку они применимы ко многим сетевым службам. Например, для надежности и транзакционности при взаимодействии требуется следовать некоторым протоколам (например, 2PC). Эти протоколы могут поддерживаться метапротоколами, но их лучше скрывать от разработчиков. Именно поэтому они включаются в стек взаимодействия, а не описания служб: их целью является не описание службы, а определение высокоуровневых свойств любого взаимодействия. Первым протоколом такого рода был протокол транзакционного взаимодействия сетевых служб *WS-Transaction*.

Сетевая служба (называемая в таком случае *базовой*) может получать запросы от других сетевых служб (называемых при этом *композитными*), возможно предоставляемых разными компаниями. С точки зрения клиента базовые и композитные службы неразличимы: это просто сетевые службы, описываемые и реализуемые одинаковыми способами. По мере роста числа используемых сетевых служб и возможности реализации, потребности в композиции служб растут. Стандартизация композитных служб находится в самом начале пути, а наиболее продвинутым стандартом является BPEL.

4.4. Внутренняя и внешняя архитектура сетевых служб

Сетевые службы имеют две архитектурные особенности (*Рис. 4.4*). Во-первых, они представляют собой окно, через которое через Интернет могут инициироваться внутренние операции. Сетевые службы должны получать запросы и передавать их нижним системным программам. В этом

их роль аналогична роли традиционных системных платформ. Эта часть инфраструктуры сетевых служб называется *внутренней*. С другой стороны архитектура сетевых служб должна обеспечивать интеграцию различных служб между собой, то есть в нее должна включаться *внешняя* инфраструктура.

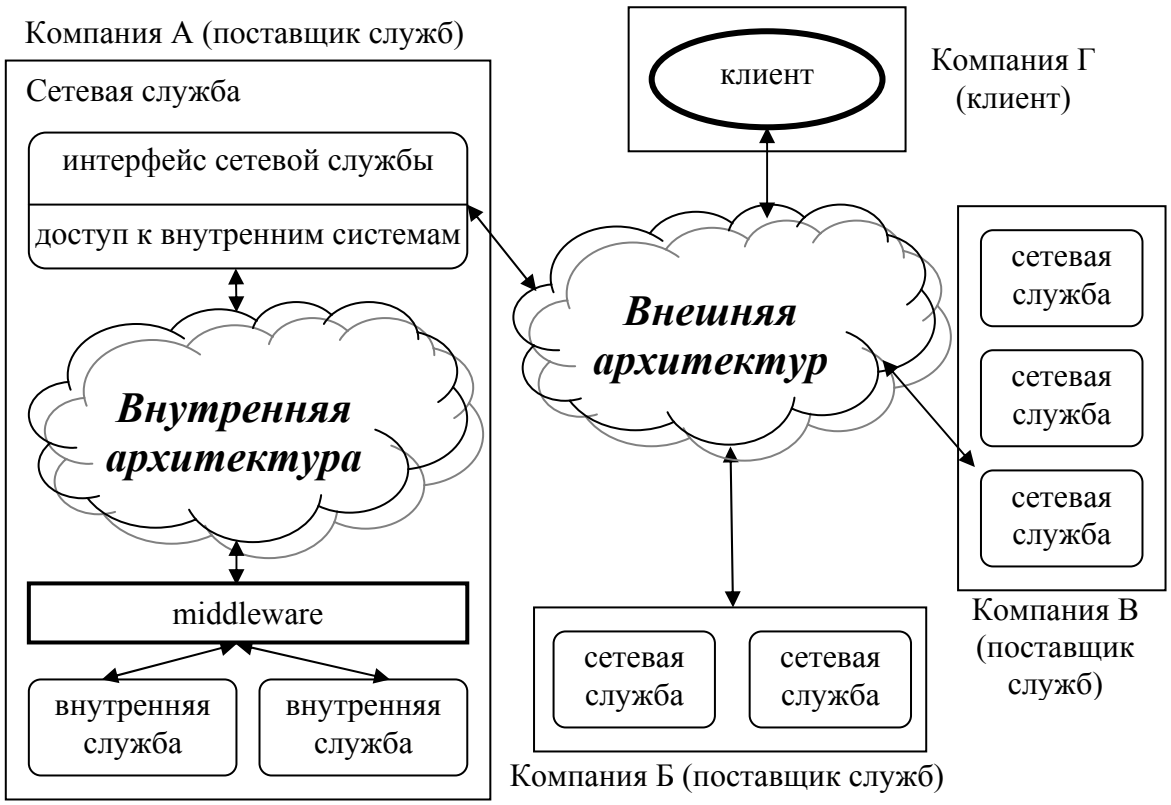


Рис. 4.4. Сетевые службы имеют внутреннюю и внешнюю архитектуры, опирающиеся на соответствующую системную поддержку.

Важно, что отнесение того или иного компонента к внутренним или внешним не связано с тем, установлен ли данный компонент у поставщика службы или где-то еще. На самом деле, разделение на внутреннюю и внешнюю архитектуры может быть проведено и в том случае, если сетевые службы используются для автоматизации внутри одного предприятия.

4.4.1. Внутренняя архитектура сетевых служб

Сетевые службы можно рассматривать как еще один ярус программного обеспечения. В системах интеграции приложений предприятия для построения многоярусных архитектур используются традиционные системы поддержки. В этих архитектурах программы скрываются за абстракциями, которые комбинируются в виде программ более высокого порядка, использующих их функциональность. Получающиеся более высокоуровневые программы в свою очередь могут упрятываться за новыми абстракциями и использоваться в качестве элементов новых служб. Этот процесс может повторяться многократно, в

результате получается архитектура, в которой службы находятся поверх других служб и базовых программ.

Программное обеспечение каждого промежуточного слоя не обязательно будет одним и тем же. Совместимость достигается введением оболочек. Промежуточные платформы склеивают отдельные уровни, формируя службы, используемые клиентами или более высокими уровнями иерархии.

Сетевые службы и технологии, поддерживающие сетевые службы, играют ту же роль, что и традиционные промежуточные слои, но имеют другой масштаб. В основе всего находится абстрактное понятие службы, напоминающее традиционные абстракции, поэтому реализация сетевой службы представляет собой реализацию еще одного (верхнего) яруса, обеспечивающего доступ с помощью стандартных протоколов сетевых служб (*Рис. 4.5*). Сетевые службы – это просто оболочки, они обращаются к внутренним службам, реализующим нужную прикладную логику, и забирают у них результаты работы.

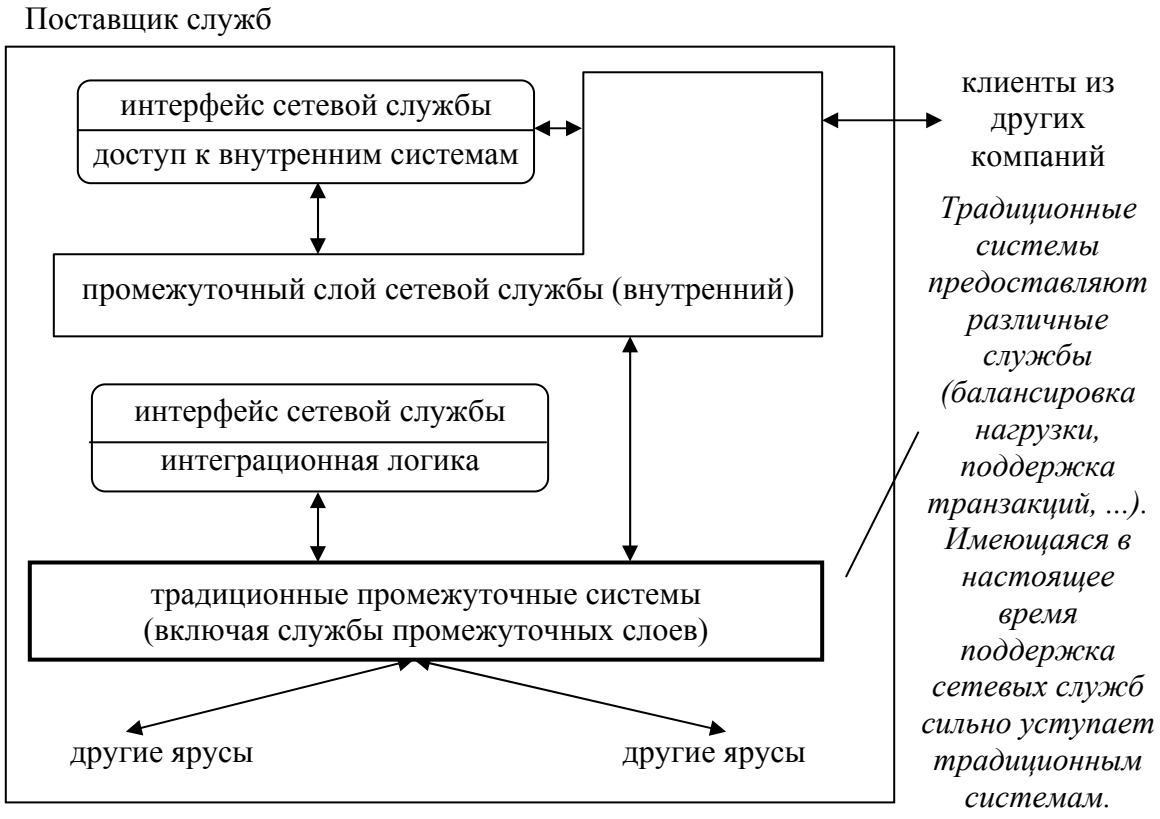


Рис. 4.5. Базовая архитектура набора сетевых служб, реализованная поверх ярусной архитектуры.

В настоящее время большая часть системной поддержки сетевых служб связана с упаковкой и распаковкой сообщений, пересылаемых между сетевыми службами, а также с преобразованием их в формат,

понятный внутреннему программному обеспечению. Это напоминает то, как сервер приложения преобразует данные в страницы HTML и обратно. Все эти преобразования приводят к росту накладных расходов на выполнение операций, поэтому сетевые службы чаще используются в крупноблочных приложениях, где такой накладной расход не так заметен.

4.4.2. Внешняя архитектура сетевых служб

Внешняя архитектура включает три важных компонента:

- ***Централизованные брокеры*** аналогичны централизованным компонентам традиционных платформ, они обеспечивают важнейшие свойства взаимодействий (журнализация, транзакционные гарантии, надежность, службы именования и справочников). На практике централизованной часто является только служба именования и ведения справочников.
- ***Инфраструктура протоколов*** относится к набору компонентов, координирующих взаимодействие сетевых служб друг с другом, в частности, реализующие децентрализованные протоколы (горизонтальные протоколы и метапротоколы), работающие там, где по каким-то причинам не могут использоваться централизованные платформы.
- ***Инфраструктура композиции служб*** относится к набору программных инструментов, полезных при определении и выполнении композитных служб.

К внешней архитектуре сетевых служб относится та сторона их работы, которая напоминает работу брокеров сообщений и систем управления рабочим потоком. Самой большой проблемой построения такой архитектуры является проблема поиска наиболее удобного места для размещения системной поддержки. В системах, работающих в локальных сетях, системная поддержка и приложения, построенные на ее основе, работают рядом, что делает простым осуществление посреднической деятельности для оказания услуг по поиску имен и справочников. Стороны, участвующие во взаимодействии сетевых служб, размещаются в разных местах, а это усложняет ситуацию.

Имеются два решения указанной проблемы. Первое заключается в реализации децентрализованной поддержки, когда все участники кооперируются и вместе выполняют функции службы именования. Это очень привлекательный подход, но остается неясным, как добиться того уровня надежности и доверия, который необходим в серьезных промышленных системах. Другое решение заключается во введении

некоторых посредников или брокеров, выполняющих нужные функции. Это может, например, быть некоторый сайт в Интернете. Если рассматривать такие серверы как часть инфраструктуры сетевых служб, это приводит к тому, что участники (и части) этой инфраструктуры могут размещаться в разных местах. В настоящее время существует только один тип брокера сетевых служб, стандартизованного и используемого на практике (хотя и очень ограниченно): сервер именования. На *Рис. 4.6* показана внешняя архитектура сетевых служб в ее централизованном варианте.

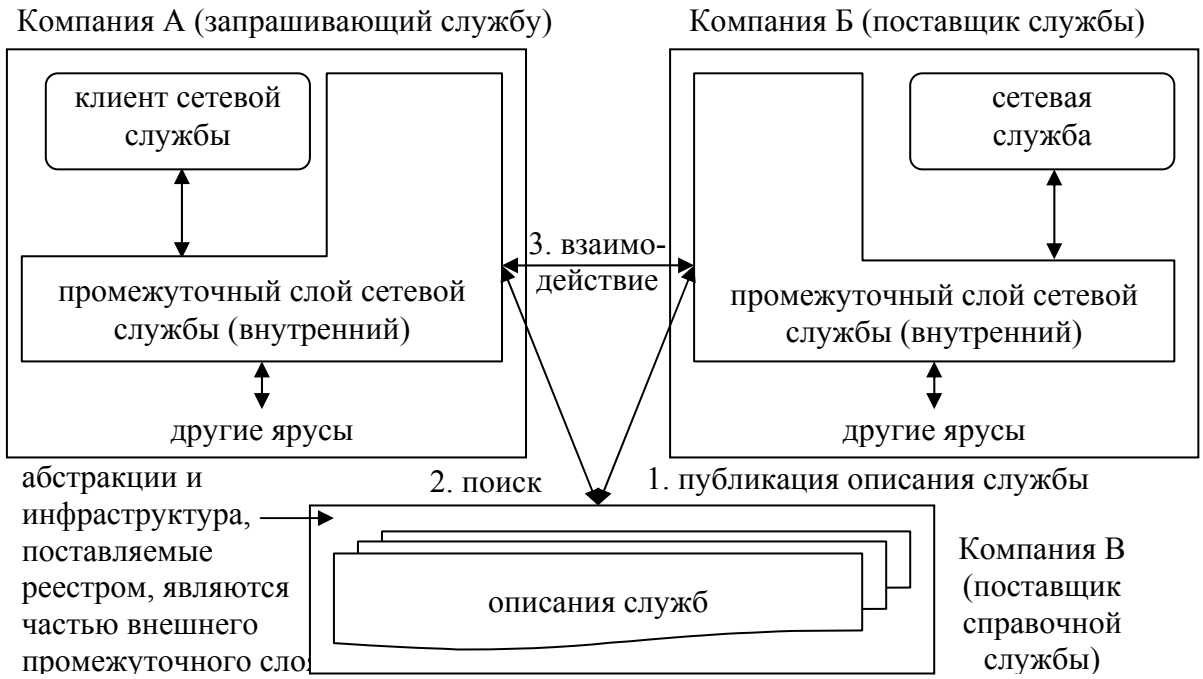


Рис. 4.6. Внешняя архитектура сетевых служб.

Идея заключается в том, что *поставщики служб* должны создавать сетевые службы и определять их интерфейсы, они также должны генерировать *описания* этих *служб* и делать их известными путем *публикации* в *реестре служб*. Информация из описания служб используется реестром для создания каталогов служб и поиска в этих каталогах по запросам от пользователей. Когда *запрашивающий службу* ищет ее, он обращается в реестр, который возвращает ему описание, из которого известно, где находится служба и как к ней обратиться. Реестр служб должен восприниматься всеми, как сетевая служба, адрес и интерфейс которой известен всем заранее.

Еще одна составляющая внешней архитектуры сетевых служб – это инструментарий для композиции служб. Композиция связана с интеграцией других служб и относится к внешней архитектуре. Технически она может быть централизованной, но так как реализации

являются частными и конфиденциальными, то может также находиться у поставщиков служб.

4.5. Базовые технологии сетевых служб

Многие архитектуры сетевых служб основаны в настоящее время на трех компонентах: запрашивающем службу, поставщике служб и реестре служб, что весьма близко к модели "клиент/сервер" с явно выделенной службой именования (регистрации). Хотя такие архитектуры имеют упрощенный характер, с их помощью можно проиллюстрировать самые основные компоненты сетевых служб: способ взаимодействия (SOAP), способ описания (WSDL) и сервер именования (UDDI).

Первое, что необходимо для всех спецификаций – это общий синтаксис их описания. В случае сетевых служб таким общим синтаксисом является XML. Все стандарты основаны на XML, а структуры данных и форматы описываются как XML-документы.

Во-вторых, необходим механизм, позволяющий удаленным сайтам взаимодействовать друг с другом. Спецификация этого механизма имеет три аспекта: общий формат данных для сообщений, которые будут использоваться при обменах, соглашение по поддержке специфических форм взаимодействия (посылка сообщений или удаленный вызов процедуры) и правила работы с сообщениями в терминах транспортного протокола. Сетевые службы могут пользоваться разными транспортными протоколами. В некоторых случаях подходит TCP/IP, для туннельного проникновения через межсетевые экраны необходим протокол HTTP, а для асинхронной отправки сообщений применяется протокол SMTP. Это означает, что механизм взаимодействия должен уметь работать с разными транспортными протоколами, а сообщения надо уметь преобразовывать, подстраиваясь под правила любого из них. Механизм также должен оставлять все взаимодействующие приложения слабо связанными. Именно поэтому он базируется не на процедурных вызовах как в RPC (хотя такие вызовы нужно обеспечивать для тех приложений, которые изначально проектировались с ориентиром на RPC), а *на обменах сообщениями*. Сетевые службы основывают свои взаимодействия на SOAP.

Сообщения протокола SOAP используются как конверты, куда приложения вкладывают отправляемую информацию. Они состоят из заголовка и тела сообщения (*Рис. 4.7*). Заголовок может отсутствовать, но тело сообщения присутствует в нем обязательно. И заголовок, и тело состоят из блоков. Предполагается, что у каждого сообщения есть отправитель, конечный получатель и некоторое число промежуточных узлов, которые обрабатывают сообщение и переправляют получателю.

Основная информация размещается в теле сообщения, заголовок служит для обработки на промежуточных узлах или для дополнительных служб (транзакционное взаимодействие, безопасность и т. д.). Протокол SOAP может использоваться разными способами. В одном случае происходит асинхронный обмен документами. Второй случай отличается от этого тем, как построено тело сообщения. В нем непосредственно содержится вызов процедуры (имя вызываемого метода и параметры). Все дополнительные свойства вызова RPC (например, указание транзакционного контекста) содержатся в заголовке.

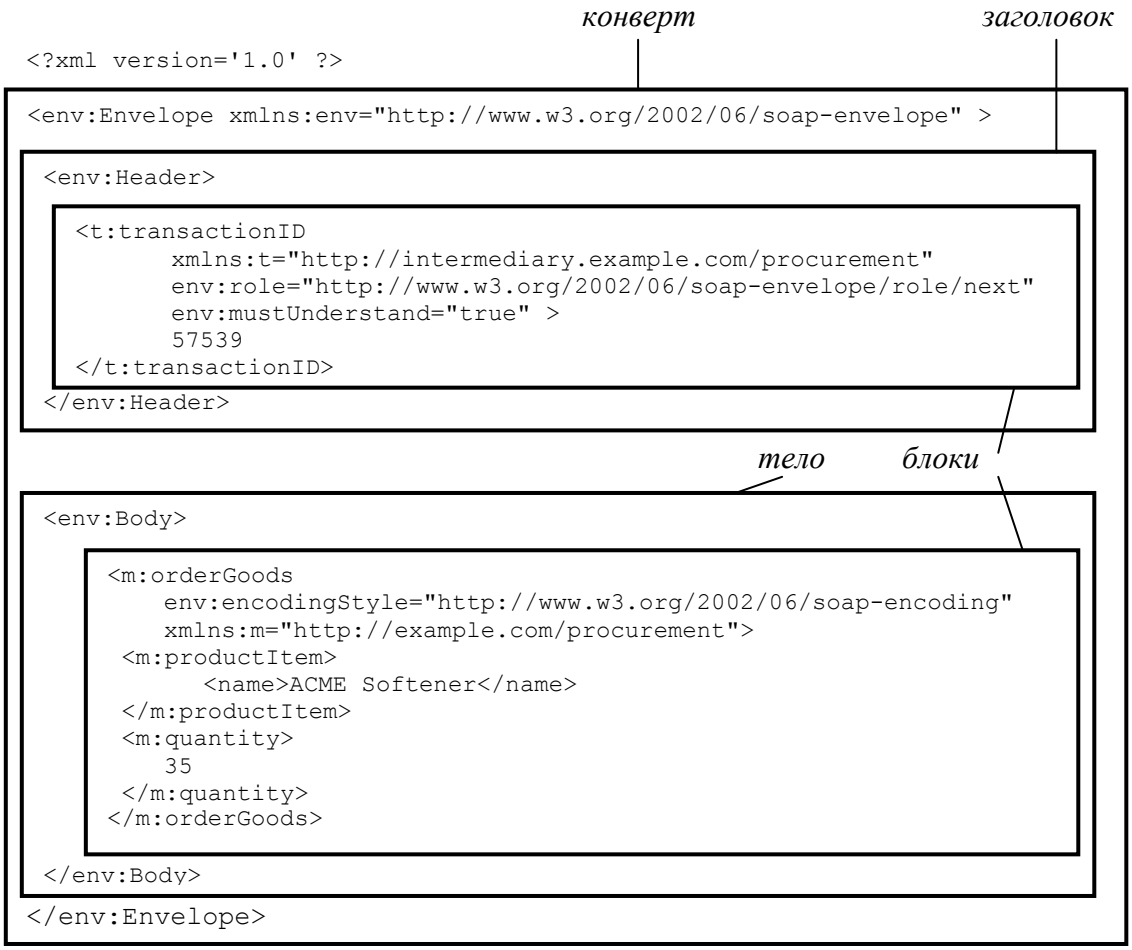


Рис. 4.7. Пример XML-сообщения протокола SOAP, имеющего в заголовке один блок, предназначенный для обработки промежуточным узлом.

Сообщения, пересылаемые по протоколу SOAP, подчиняются правилам кодирования, определяющим, как конкретная структура будет представлена в XML. Клиент и сервер должны заранее договариваться о выбранном способе передачи параметров сообщений (удаленных процедур) – в виде вложенных элементов или значениями атрибутов исходных элементов (*Рис. 4.8*).

Промежуточные узлы, которые поочередно обрабатывают SOAP-сообщения по мере их доставки получателю, обычно представляют собой

разные ярусы системной поддержки сетевой службы. Каждый узел может играть в процессе передачи ту или иную роль, в блоках заголовка сообщения можно указывать, для выполнения какой роли этот блок предназначен. Если блоку приписана роль "*никто*", блок не обрабатывается ни на одном узле на пути сообщения (блок можно только читать). Если блоку приписана роль "*конечный получатель*", ни один промежуточный узел обработкой блока заниматься не будет. Если блоку приписана роль "*следующий*", блок может обрабатываться на каждом узле, куда попадает сообщение (в том числе и на узле конечного получателя). Тело сообщения не имеет приписанной роли, оно всегда обрабатывается конечным получателем.

```
<ProductItem>                <ProductItem              <ProductItem name="…"
   <make> … </make>             name="…"                  <type>…</type>
   <name> … </name>             type="…"                  <make>…</make>
   <type> … </type>             make="…"               </ProductItem>
</ProductItem>               />
```

Рис. 4.8. Разные методы кодирования сообщений в XML.

Обработка блоков может быть самой разной (удаление из заголовка, внесение добавочной информации и т. д.). Блок может содержать признак, который требует обязательной обработки узлом с указанной ролью. Если такой признак не установлен, узел, играющий соответствующую роль, самостоятельно принимает решение обрабатывать данный блок или игнорировать его. Тело сообщения такого признака не имеет, но конечный получатель все равно должен его обработать или выработать ошибку.

Для того, чтобы механизм передачи сообщений SOAP функционировал (*Рис. 4.9*), необходимо связать его с транспортным протоколом, действующим в сети. Спецификация транспортного протокола, то есть *привязка*, определяет, как надо преобразовать сообщение к правилам транспортного протокола и как трактовать сообщение в терминах транспортного протокола (*Рис. 4.10*).

Привязка протокола SOAP к транспортному протоколу имеет и другую неявную функцию – *функцию адресации*. Указание адреса конечного получателя отсутствует в сообщении SOAP. Это связано с тем, что это сообщение является частью запроса HTTP или частью сообщения SMTP. В этих протоколах имеются необходимые средства для описания адресата (URL или поле "to"). Близкой проблемой является проблема *маршрутизации*. Протокол SOAP описывает путь доставки сообщений как набор узлов, механизма точной спецификации пути в сообщениях SOAP нет.

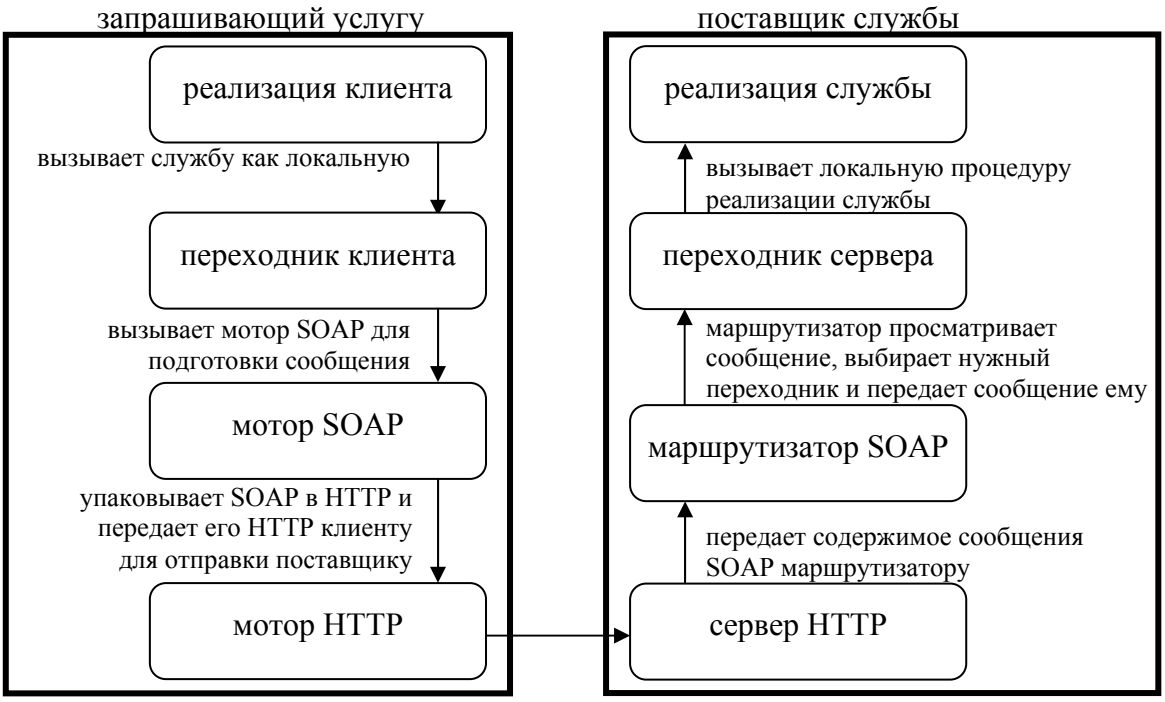


Рис. 4.9. Простейшая реализация протокола SOAP.

Протокол SOAP может работать в асинхронном режиме, пригодном для взаимодействия слабосвязанных партнеров. Вместо прямых вызовов при этом запросы отправляются с помощью системы очередей. Слабосвязанное взаимодействие обычно приводит к обмену сообщениями в стиле документов, выполняемому асинхронно и автоматически. Ответы передаются также асинхронно. Взаимодействующие объекты (в прежней терминологии – серверы и клиенты) остаются максимально независимыми друг от друга, что позволяет проводить их независимую разработку, сопровождение и внедрение. Обычно при этом в качестве транспортного протокола используется SMTP.

В-третьих, после выбора синтаксиса описаний и протокола взаимодействия для посылки сообщений нужна возможность описывать службы (и их интерфейсы) стандартным образом. Роль, отведенную в традиционных системах языкам IDL, играет для сетевых служб основанный на XML язык описания WSDL. Интерфейс описывается в терминах методов, которые поддерживаются сетевой службой, а каждый метод может принимать одно сообщение на входе и возвращать другое сообщение на выходе. В терминах RPC эти сообщения содержат входные и выходные параметры вызовов процедур. Используется WSDL так же, как и IDL. Файлы с описаниями транслируются в соответствующий язык программирования, при этом генерируются переходники и промежуточные слои, делающие обращения к сетевым службам прозрачными. Инфраструктура сетевой службы использует WSDL и SOAP для конструирования заместителей объектов на сторонах поставщика и

потребителя, поэтому разработчики могут программировать свои приложения, как если бы они выполняли локальные вызовы.

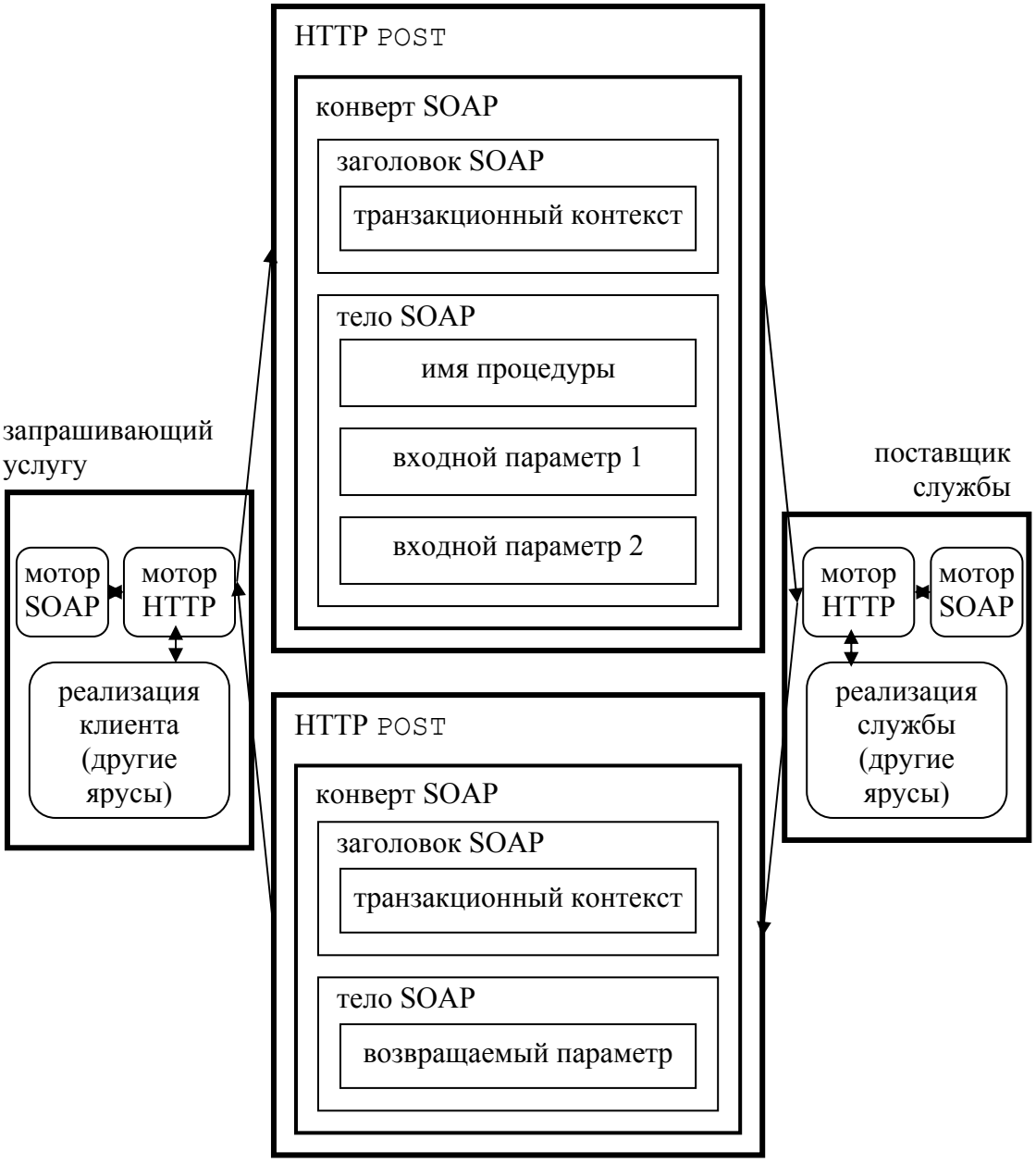


Рис. 4.10. Вызов удаленной процедуры с использованием протокола SOAP поверх HTTP.

Однако язык WSDL имеет ряд серьезных отличий от подхода, применявшегося в IDL. Во-первых, кроме спецификации операций, предлагаемых службой, необходимо дополнительно описывать механизм доступа к этой службе. Службы не привязаны друг к другу и не привязаны ни к какой системной платформе. Вызов процедуры можно специфицировать, описав лишь параметры вызова и возвращаемый результат. Вся остальная информация задается неявно, а механизм един для всех традиционных служб, пользующихся одной и той же системной

платформой. Сетевые службы делаются доступными посредством протоколов, а их надо специфицировать на WSDL.

```
<?xml version= "1.0"?>
<definitions name="Procurement"
   targetNamespace="http://example.com/procurement/definitions"
   xmlns:tns="http://example.com/procurement/definitions"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/" >

   <message name="OrderMsg">
       <part name="productName" type="xs:string"/>
       <part name="quantity" type="xs:integer"/>
   </message>

   <portType name="procurementPortType">
       <operation name="orderGoods">
          <input message = "OrderMsg"/>
       </operation>
   </portType>

   <binding name="ProcurementSoapBinding" type="tns:procurementPortType">
       <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
       <operation name="orderGoods">
            <soap:operation soapAction="http://example.com/orderGoods"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
       </operation>
   </binding>

   <service name="ProcurementService">
       <port name="ProcurementPort" binding="tns:ProcurementSoapBinding">
            <soap:address location="http://example.com/procurement"/>
       </port>
   </service>

</definitions>
```

абстрактная часть

сообщение

операция и тип порта

конкретная часть

привязка к протоколу

порт и служба

Рис. 4.11. Пример спецификации на языке WSDL.

Другим отличием, вызванным отсутствием общей системной платформы, является необходимость определения места, в котором доступна служба. В традиционных системах достаточно реализовать интерфейс и зарегистрировать его в системном слое, который сам, когда это требуется, заботится об активации объекта. Отсутствие централизованной системной платформы заставляет поставщиков служб иметь стандартизованные средства спецификации места их размещения.

Чтобы иметь возможность решать все проблемы, возникающие при описании служб, спецификации WSDL (*Рис. 4.11*) делят на две части –

абстрактную (концептуально похожую на традиционные описания на IDL) и конкретную (определяющую протоколы связывания и другую информацию).

Абстрактная часть построена на определениях *типов портов*, аналогичных интерфейсам традиционных IDL. Каждый тип порта есть логический набор связанных с портом *операций*. Каждая операция определяет простой обмен *сообщениями*, то есть коммуникационными единицами, представляющими обмен данными в одной логической передаче. Чтобы обмениваемые данные правильно интерпретировались на обоих концах взаимодействия, необходимо определить их *типы*. Все эти понятия определяются на XML.

Сообщения в WSDL представляют собой текстовые документы, разделенные на части, каждая из которых характеризуется именем и типом, который ссылается на тип, обычно определенный с помощью схемы XML. Например, вызывая процедуру с двумя параметрами, надо определить сообщение из двух частей, одна из которых будет содержать, например, целое, а другая, например, вещественное число. Эти типы входят в стандартный набор спецификаций схем XML, а если бы надо было использовать параметр сложной структуру, потребовалось бы дополнительно определить тип такой структуры.

В WSDL имеется четыре типа базовых операций: *односторонние*, *уведомление*, *запрос/ответ* и *просьба/ответ*. Первые два типа операций связаны с одиночными сообщениями, выполняемыми асинхронно, а последние два – с синхронными парными сообщениями. Сообщения типа уведомление и просьба/ответ инициируются службой, а других типов – клиентом.

Определение абстрактного интерфейса завершается в WSDL группированием операций с определением типов портов. Сложные типы портов могут опираться на ранее определенные типы портов. В таких случаях они содержат все операции, входящие во вложенные типы, а также дополнительные операции.

Все перечисленные определения (типы данных и портов, сообщения и операции) называются абстрактными потому, что в них не определяются ни конкретный транспортный протокол, ни тип кодирования информации, ни сама служба, которая реализует набор типов портов. Один и тот же тип порта может быть реализован с помощью разных протоколов, он же при формировании сетевой службы может комбинироваться с другими типами портов в самых разных вариантах. Аналогично, одни и те же типы данных

могут кодироваться в сообщения по разным правилам, а одни и те же сообщения могут пересылаться по разным протоколам.

Разделение на абстрактную и конкретную части спецификации WSDL имеет очень важное следствие – спецификации, описывающие абстрактную часть, становятся переиспользуемыми. Это означает, что одни и те же абстрактные спецификации могут использоваться с различными привязками к протоколам и по разным сетевым адресам. Возможность как SOAP, так и WSDL использовать много разных транспортных привязок серьезно повышает степень стандартизации сетевых служб, поскольку отрывает описания от различных реализаций.

Конкретная часть интерфейса WSDL определяется с помощью трех конструкций:

- ***Интерфейсное связывание***, определяющее тип кодирования сообщений и привязку к протоколу для всех операций и сообщений, заданных для данного типа порта. Интерфейсное связывание может определять, что сообщения некоторой операции должны пересылаться с использованием протокола SOAP с привязкой к протоколу HTTP. Здесь же специфицируются правила кодирования, на основе которых сообщения сериализуются в XML.
- ***Порты*** соединяют информацию интерфейсного связывания с сетевыми адресами, специфицируемыми с помощью URI. В традиционных системных платформах этого делать нет нужды, но для сетевых служб – необходимо.
- ***Службы***, являющиеся логическими группами портов. По крайней мере, в принципе, это означает, что некоторая WSDL-служба может оказаться доступной в Интернете распределенной сразу по нескольким адресам. На практике наиболее частой оказывается ситуация, в которой в одной службе группируются близкие по семантике порты, расположенные в одном месте. Часто также в службу включаются порты одного типа, но имеющие привязки к разным протоколам.

Одним из следствий структур данных, имеющихся в WSDL, является дальнейшее слияние понятий "клиент" и "поставщик службы". Как и всякий язык описания интерфейсов WSDL может использоваться в разных целях, важнейшими из которых можно считать такие:

- Традиционное использование в качестве языка описания службы.
- Использование в качестве входных данных для трансляторов переходников и других программных инструментов, которые по описаниям на WSDL могут генерировать переходники и другие

дополнительные данные, полезные как разработчикам служб, так и их пользователям.

Поскольку сетевые службы применяются для демонстрации внутренних операций через Интернет, часто при начале разработки службы приложения, на которые она опирается, уже существуют. Это позволяет автоматически генерировать тексты на WSDL по уже имеющимся описаниям прикладных программных интерфейсов. На *Рис. 4.12* изображена описанная схема, принципиально отличающаяся от схемы работы со спецификацией CORBA только отсутствием общей системной платформы.

В-четвертых, имея полное описание сетевой службы, чтобы действительно обратиться к сетевой службе, ее нужно отыскать. Чтобы службы имели глобальный характер и вызывали всеобщий интерес, процесс опубликования сведений о них и их локализация должны быть стандартизованы. Потребители должны иметь возможность искать интересующие их службы, понимать их свойства (интерфейсы и URI, по которым они могут оказаться доступными). Это делает важным стандартизацию реестра сетевых служб.

Рис. 4.12. Документы WSDL могут генерироваться на основе прикладных программных интерфейсов. Пунктиром показаны действия стадии разработки.

Проект *универсальных средств описания, обнаружения и интеграции* (*UDDI*) как раз и представляет собой такой стандарт. Фактически он

состоит из двух частей: реестра и прикладного программного интерфейса. Реестр (или регистр) эквивалентен серверу именования, имеющемуся в любой системной платформе. Это то место, где публикуются описания служб, которые могут затем отыскиваться в каталогах реестра. Прикладной интерфейс UDDI (*Рис. 4.13*) определяет, как опубликовать службу, какая информация необходима для регистрации службы, как делать запросы к службе.

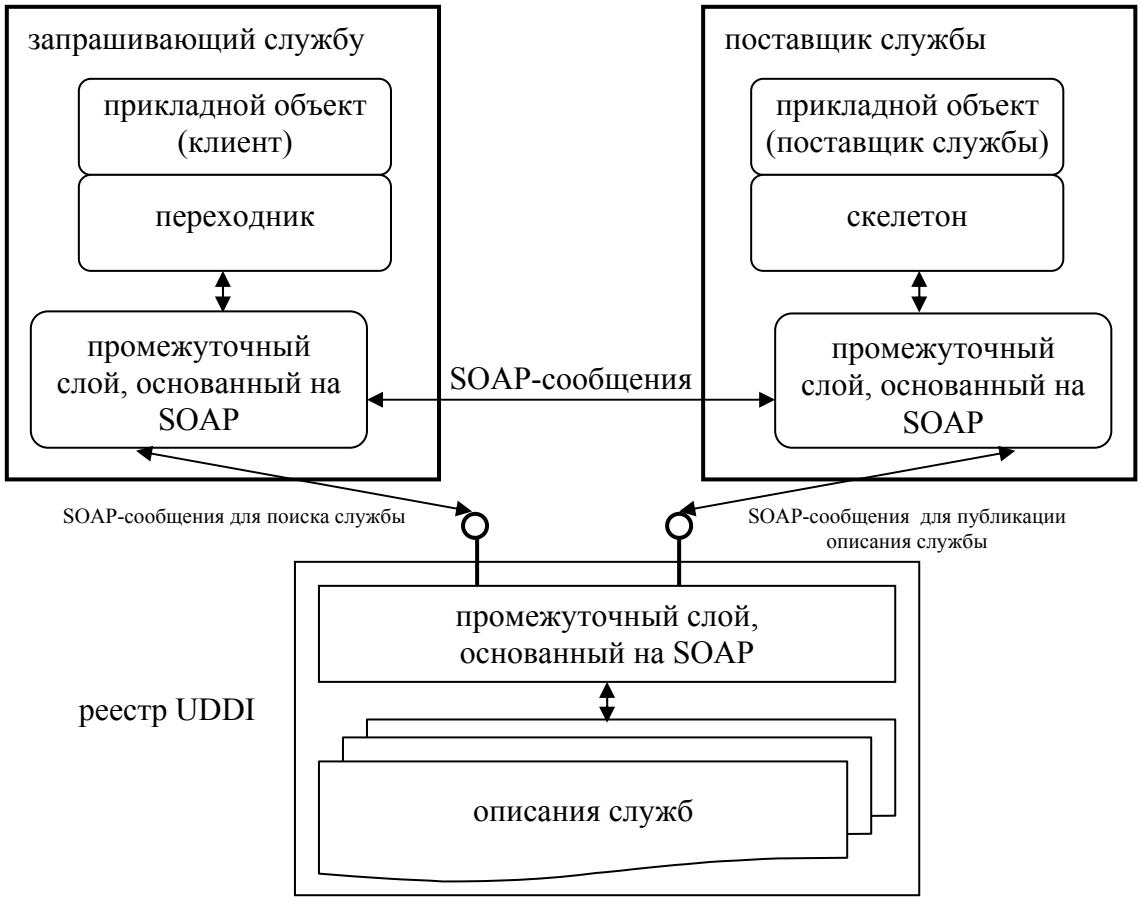


Рис. 4.13. Поставщики сообщают о своих службах в реестр UDDI. Клиенты (во время разработки или выполнения) ищут службы в реестре, а затем обращаются к ним.

Поиск информации в реестре может преследовать две цели: во-первых, необходимо понять, что надо сделать, чтобы написать клиентскую программу для доступа к службе, во-вторых, после того, как эта клиентская программа написана, ей надо обеспечить возможность динамического поиска службы. Реестр UDDI также может использоваться в качестве *универсального бизнес реестра*, где каждый производитель программного обеспечения может свободно и бесплатно регистрировать свою продукцию, а пользователи могут искать интересующие их программы. Информация, занесенная в реестр UDDI, может группироваться там в алфавитном порядке имен предприятий, поставляющих сетевые службы для всеобщего использования, по

тематике, к которой относится деятельность предприятий поставщиков и сами службы, а также по способам вызова сетевых служб.

При занесении в реестр сетевая служба сопровождается различной информацией:

- ***предприятие*** включает сведения об имени компании, ее адресе и другую контактную информацию.
- ***служба*** описывает набор сетевых служб, предлагаемых данным предприятием. *Предприятие* может поставлять несколько *служб*, но *служба* может поставляться только одним *предприятием*.
- ***шаблон использования*** описывает техническую информацию, необходимую для использования конкретной службы: адрес, по которому доступна сетевая служба, а также детализированный набор ссылок на документы (*технические модели*), где описываются интерфейсы служб и другие их свойства. Одна *служба* может содержать несколько *шаблонов использования*, но *шаблон* может относиться только к одной *службе*.
- ***техническая модель*** может содержать произвольную информацию: WSDL-интерфейс службы, классификацию, протокол взаимодействия, сведения о семантике службы. На технические модели могут ссылаться любые объекты реестра (предприятия, службы, шаблоны), они не привязаны ни к одному из них.

Для опубликования службы в реестре сначала необходимо определить технические модели, причем иногда может оказаться, что они уже существуют в реестре, и на них ссылаются другие объекты. Затем публикуются сведения о предприятии, общая информация о поставляемых службах, и, наконец, техническая информация о каждой отдельной реализации и о точках доступа к службам со ссылками на технические модели.

Структура технической модели очень проста (*Рис. 4.14*). Фактическое описание находится в документах, на которые в модели только приводятся ссылки. Содержимое обзорных документов обычно не структурировано и предназначено для чтения людьми. Каждая техническая модель, зарегистрированная в реестре, имеет уникальный ключ, который помогает разработчикам получать доступ к моделям и, таким образом, получать информацию, нужную для разработки клиентских программ. Зная уникальный ключ моделей, разработчики имеют возможность организовывать динамический поиск служб, ссылающихся на них.

Технические модели могут использоваться не только для описания интерфейсов и протоколов, но также для идентификации и классификации. Каждый может определить собственную классификационную схему (например, по географическим признакам). Полное описание классификации помещается в обзорные документы, а в технических моделях остаются только классификационные признаки, которые можно использовать как поисковые ключи. Статически или динамически можно искать службы, относящиеся к данной классификационной категории (например, искать предприятие или службу, ссылающиеся на модель 211 и имеющие классификационный признак "Москва").

обзорный документ (ссылается на спецификацию WSDL и спецификацию API)

```
<tModel tModelKey="uddi:uddi.org:v3_publication">
   <name>uddi-org:publication_v3</name>
   <description>UDDI Publication API_V3.0</description>
   <overviewDoc>
       <overviewURL useType="wsdlInterface">
          http://uddi.org/wsdl/uddi_api_v3_binding.wsdl#UDDI_Publication_SoapBinding
       </overviewURL>
   </overviewDoc>
   <overviewDoc>
       <overviewURL useType="text">
          http://uddi.org/pubs/uddi_v.3.htm#PubV3
       </overviewURL>
   </overviewDoc>

   <categoryBag>
       <keyedReference keyName="uddi-org:types:wsdl"
          keyedValue="wsdlSpec"
          tModelKey="uddi:uddi.org:categorization:types"/>
       <keyedReference keyName="uddi-org:types:soap"
          keyValue="soapSpec"
          tModelKey="uddi:uddi.org:categorization:types"/>
       <keyedReference keyName="uddi-org:types:xml"
          keyValue="xmlSpec"
          tModelKey="uddi:uddi.org:categorization:types"/>
       <keyedReference keyName="uddi-org:types:specification"
          keyValue="specification"
          tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

классификационная информация (специфицирует, что данная техническая модель имеет отношение к XML, WSDL и SOAP)

Рис. 4.14. Пример технической модели в реестре UDDI, описывающей прикладной программный интерфейс самого реестра UDDI.

Технические модели, как и другие объекты в реестре, могут ссылаться на технические модели. Можно, в частности, ссылаться на модели, содержащие различные виды классификации одних тех же объектов, вводя собственные таксономии с помощью различных схем категоризации.

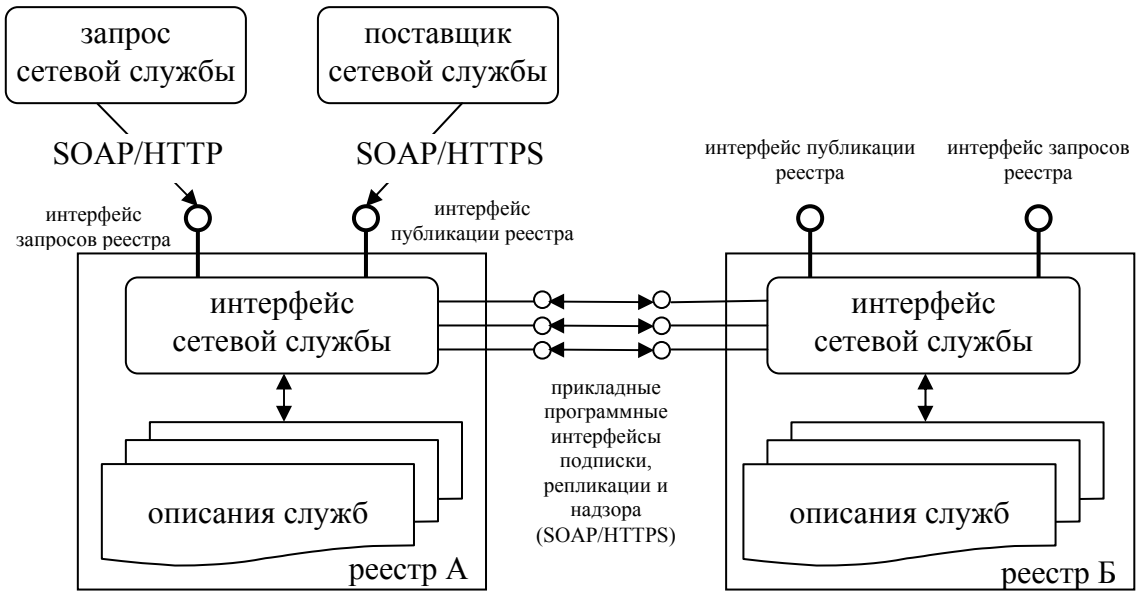


Рис. 4.15. Взаимодействие с реестрами UDDI и между ними.

Работу с реестром UDDI могут проводить пользователи разных типов: поставщики служб, публикующие сведения о них, клиенты, ищущие службы для выполнения собственных задач, и другие реестры, осуществляющие обмен информацией с данным реестром (*Рис. 4.15*). Для разных типов пользователей реестры поддерживают разные точки входа, взаимодействие с которыми осуществляется по протоколу SOAP обменом XML-документами.

Всего реестр имеет несколько различных видов прикладного программного интерфейса, важнейшим из которых является прикладной интерфейс запросов, сознательно спроектированный очень простым, что сделано для облегчения процессов стандартизации и всеобщего признания. Существующий упрощенный подход приводит даже к тому, что в реестрах не проводится распределенный поиск.

Использование реестров и размещенных в них описаний служб совсем не отменяет те описания этих служб, которые на языке WSDL делаются разработчиками. Определения WSDL-интерфейса (точнее типы портов и привязка к протоколам, но не порты и адреса) регистрируются в качестве технических моделей. В полях, предназначенных для обзорных документов этих моделей, находятся ссылки на соответствующие документы WSDL.

4.6. Работа сетевой службы

И описания, и поиск сетевых служб в реестрах проводятся для того, чтобы, в конце концов, правильно организовать работу сетевых служб. Информация об интерфейсе, описывающем имя процедуры, передаваемые ей параметры и возвращаемый ею результат, может автоматически

транслироваться в описание сетевой службы (шаг 1 на *Рис. 4.16*). В большинстве существующих систем привязка проводится к протоколу SOAP и сетевому адресу машины, на которой работает система управления базы данных.

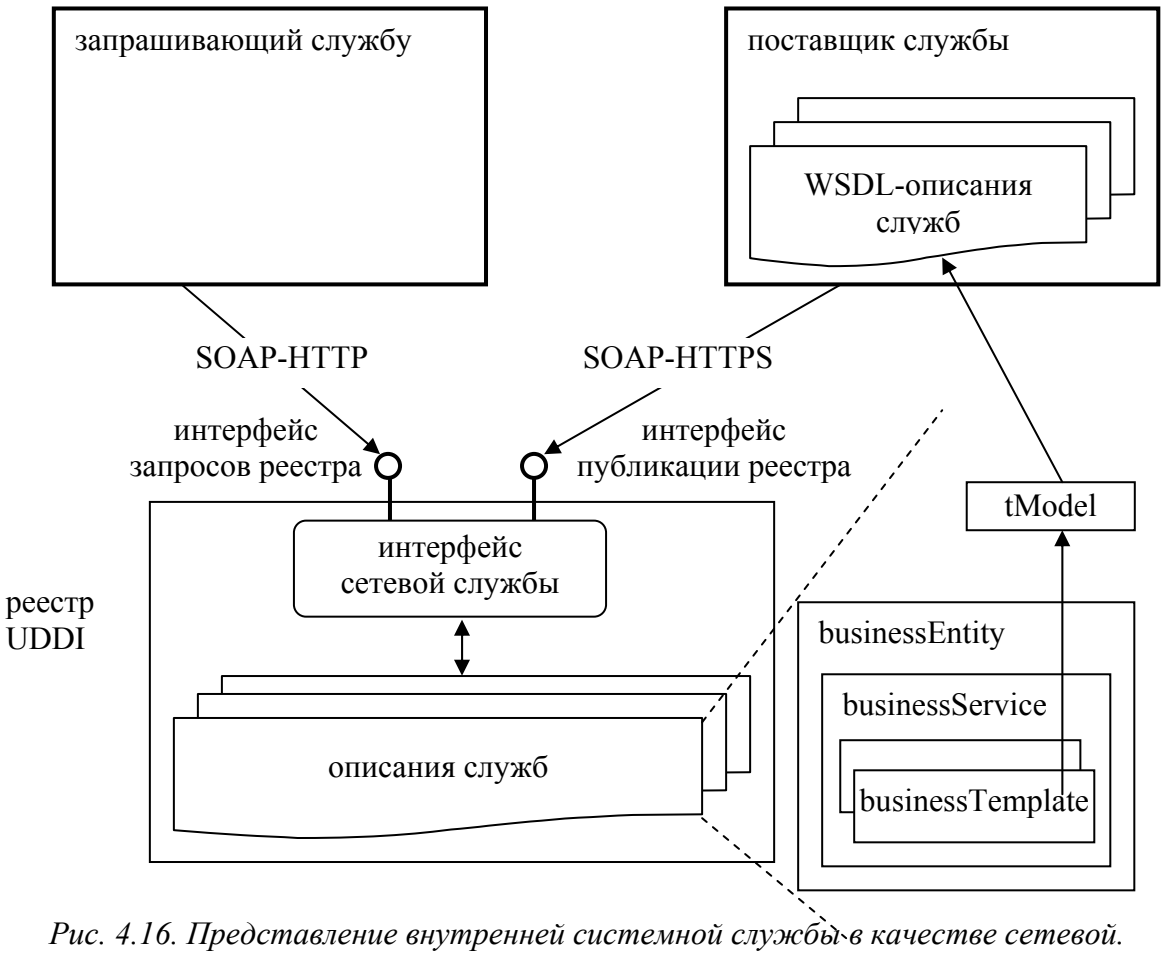


Рис. 4.16. Представление внутренней системной службы в качестве сетевой.

Полученный текст на WSDL хранится у поставщика службы. Компилятор языка WSDL создает серверный переходник (обычно в виде сервлета) и регистрирует его на маршрутизаторе SOAP. Теперь маршрутизатор знает, что при обращении к определенному ресурсу он должен вызвать зарегистрированный переходник (шаг 2 на *Рис. 4.16*). В свою очередь переходник будет вызывать исходный объект (в этом случае – хранимую процедуру базы данных). *Этим завершается реализация сетевой службы*. Служба стала вполне работоспособной, ее можно вызывать, но клиентское приложение реестра UDDI должно выполнить последний шаг (шаг 3 на *Рис. 4.16*), состоящий из двух этапов. Первый этап заключается в публикации в некотором реестре технической модели, ссылающейся на автоматически полученное описание на языке WSDL. Реестр должен быть доступен тем клиентам, для которых готовится служба. Второй этап заключается в публикации полноценной записи в

реестре, в которой должны содержаться сведения об адресе, по которому служба доступна, и ссылка на техническую модель.

Сетевая служба становится видной клиентам, которые получают возможность искать ее и взаимодействовать с ней. Разработчики могут просматривать реестры, выявлять соответствующие ей описания на языке WSDL, автоматически генерировать клиентские переходники своими WSDL-компиляторами, создавать клиентские приложения. После завершения разработки приложений их нужно привязывать к соответствующим портам (статически или динамически), что выполняется с помощью реестров сетевых служб.

Описанный процесс иллюстрирует, как можно пользоваться сетевыми службами полностью автоматически. Многие поставщики системного программного обеспечения поставляют моторы SOAP, компиляторы WSDL и другой инструментарий, делающий процесс вызова службы прозрачным для разработчиков и клиентских и серверных частей сетевых служб, то есть позволяющий обращаться к ним, как к локальным службам.

Другой типичный сценарий работы с сетевыми службами возникает при использовании стандартизированных интерфейсов сетевых служб. Отличается он тем, что тексты на WSDL извлекаются из реестров UDDI. Для этого нужно использовать технические модели соответствующих обзорных документов реестров. По извлеченным описаниям компилятор WSDL может генерировать программы, требующиеся на серверной стороне. В мире Java это приводит к генерации сервлета, который нужно зарегистрировать в маршрутизаторе SOAP, и интерфейса на языке Java, который реализуется службой. После реализации службы объект регистрируется в маршрутизаторе SOAP, и служба становится доступной. Ее публикация в реестре UDDI проводится как и раньше, только теперь не требуется публиковать техническую модель, которая уже имеется в реестре.

4.7. Координация работы сетевых служб

В реальном мире работа по распределенной обработке информации проходит в гораздо более сложном режиме, чем простая активация одиночной сетевой службы. Переход от одиночных, независимых вызовов к сериям операций, порядок выполнения которых важен для получения правильного результата, оказывает серьезное влияние, как на реализацию, так и на процесс взаимодействия с сетевыми службами. Чтобы взаимодействие служб стало организованным (и в принципе возможным), его надо проводить, следуя специальным протоколам. Протоколы

взаимодействия сетевых служб строятся на основе понятия "*разговора*", который можно описывать диаграммой состояний некоторой абстрактной машины.

Для описания протоколов координации служб важным является понятие "*роли*". Участники взаимодействия "играют" свои роли, обмениваясь сообщениями. Протокол накладывает ограничения на порядок, в котором такой обмен может происходить. В отличие от термина "*разговор*", который относится ко всякой последовательности операций (обменов сообщениями), могущей возникнуть между клиентом и службой в процессе обращения к службе, термин "*координационный протокол*" относится именно к спецификации набора правильных (допустимых) разговоров. Важно иметь стандартизованный способ описания координационных протоколов, который позволил бы разрабатывать программное обеспечение для автоматической поддержки и управлять разговорами с сетевыми службами. Существующие попытки стандартизации, в частности, стандарт языка WSCL пока не получили достаточной поддержки, поэтому для описания координационных протоколов применяются различные модели.

Среди используемых моделей выделяются две: модель диаграмм последовательности и модель диаграмм активности. На *Рис. 4.17* приведено описание разговора трех сетевых служб.

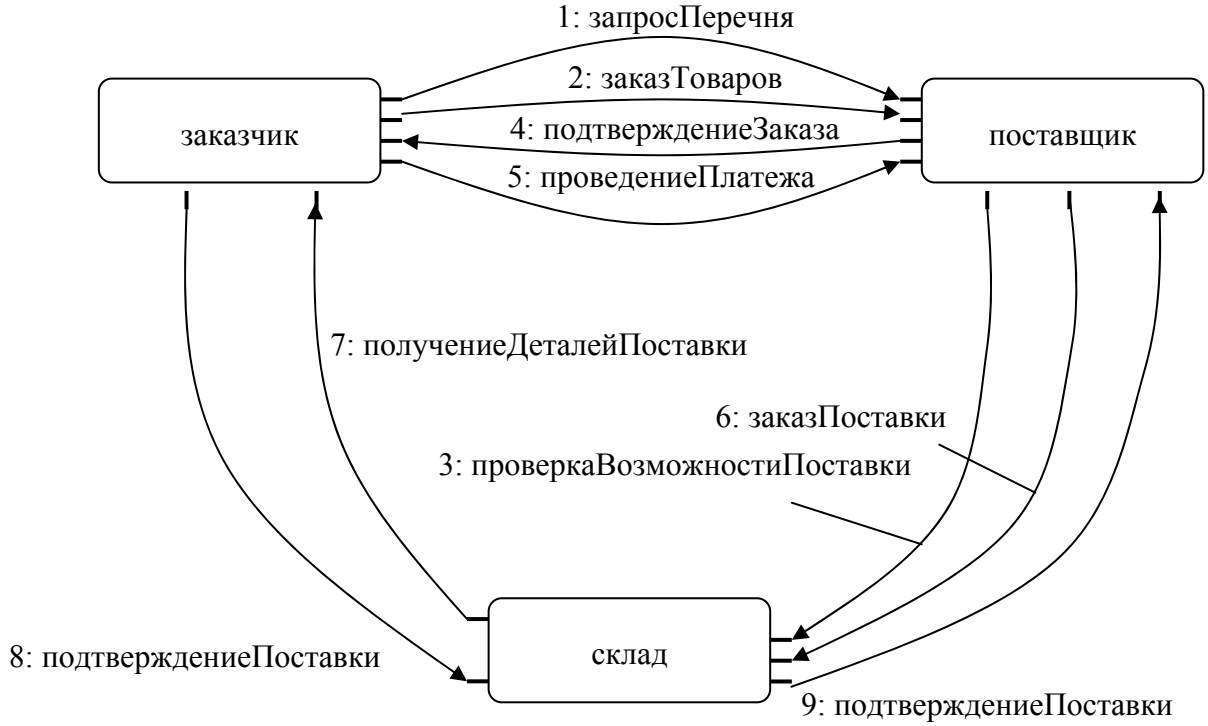


Рис. 4.17. Последовательность обмена сообщениями трех сетевых служб при отработке заказа на приобретение товара.

Диаграммы последовательности (*Рис. 4.18*) распространены именно ввиду их простоты и интуитивной понятности, однако, чем сложнее протокол, тем сложнее становится разобраться в этих диаграммах. Например, если бы потребовалось ввести еще одного участника взаимодействия (на стадии выяснения наличия товара на складе им мог бы быть еще один склад, оптовый) или провести дополнительные переговоры с заказчиком об изменении даты поставки, возникли бы еще несколько разговоров, параллельных первому. Иногда так и поступают, прорисовывая несколько диаграмм (такая практика используется также при описании протокола TCP), но это удобно лишь при наличии небольшого количества альтернатив, иначе диаграмм становится слишком много. В последнее время все большую популярность приобретают *диаграммы активности* (*Рис. 4.19*), с их помощью удобно демонстрировать альтернативные и параллельные ветви разговоров.

Координационные протоколы имеют большое значение для описания сетевых служб. Они похожи на описания интерфейсов служб, которые тоже призваны разъяснять, как организовать взаимодействие со службой.

Рис. 4.18. Диаграмма последовательности, соответствующая разговору трех служб, показанному на Рис. 4.17.

Зная координационный протокол и роль сетевой службы в протоколе, можно проектировать взаимодействия с этой службой. Координационные протоколы помогают также организовать динамическую привязку к службе, поскольку приложения, разработанные для участия в некотором протоколе, могут ограничить поиск сетевых служб в реестрах только теми, которые играют определенные роли.

Координационные протоколы обычно хранятся в реестрах в виде технических моделей, а роли, исполняемые службой, могут указываться в *шаблонах использования*, ссылающихся на эти технические модели.

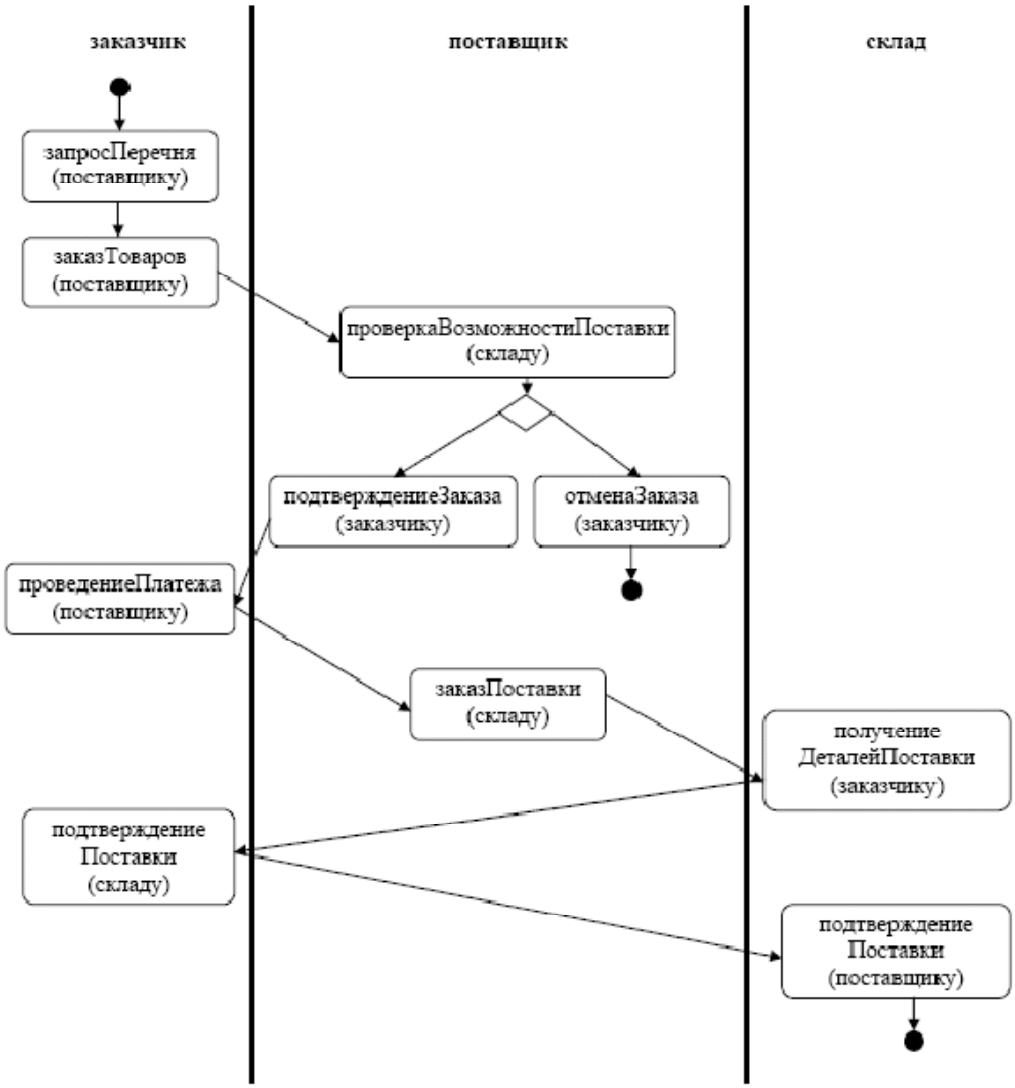


Рис. 4.19. Диаграмма активности разговора, показанного на Рис. 4.17.

Координационные протоколы (как и интерфейсы) не раскрывают деталей реализации, что облегчает внесение изменений в программы взаимодействия служб (до тех пор, пока изменения в программах не затрагивают протокол). Еще один аспект сокрытия информации связан с генерацией ролевых протоколов, являющихся подмножествами полных протоколов, но содержащих только те операции и сообщения, которые необходимо знать исполнителю определенной роли.

Координационные протоколы сетевых служб делятся на *вертикальные* и *горизонтальные*. *Вертикальные протоколы* относятся к отдельным отраслям (и называются бизнес протоколами). Обычно в них описывается, каким образом следует строить конкретные бизнес транзакции, определяются форматы соответствующих документов,

семантика содержимого этих документов. Протокол, проиллюстрированный на *Рис. 4.17-4.19*, тоже является вертикальным.

Многие важные детали реализации (как проводить обмен сообщениями) часто в вертикальных протоколах отсутствуют, они больше концентрируются на семантике обменов, а также на наборах правильных разговоров. Детали – это то, с чем имеют дело *горизонтальные протоколы*, в которых определяется общая инфраструктура, независящая от прикладной области. Однако сетевые службы предназначены для взаимодействия не внутри, а между предприятиями, и многие ранее разработанные методы для них не пригодны. В частности, из-за длительности взаимодействия протоколы подтверждения типа 2PC использоваться не могут, так как они в момент этого подтверждения блокируют ресурсы. Следует разрабатывать новые стандарты (уже разработаны стандарты *WS-Coordination*, *WS-Transaction*).

4.7.1. Инфраструктура координационных протоколов

Программы, предназначенные для выполнения разговоров служб, называются *контроллерами разговоров*. Их функциональность имеет двойную направленность: они *маршрутизируют разговоры* и *верифицируют их соответствие протоколу*. Обычно контроллеры разговоров включаются в состав маршрутизаторов SOAP. Маршрутизация разговоров связана с проблемой диспетчеризации сообщений: необходимо, чтобы эти сообщения попадали нужным внутренним объектам. При получении сообщения служба определяет, к какому разговору сообщение относится, и как это сообщение надо обрабатывать, что зависит от состояния разговора. Контроллер разговоров может справляться с этим, включением в заголовки каждого пересылаемого сообщения уникального для данного разговора идентификатора. Такой идентификатор должен генерироваться каждый раз в начале нового разговора, и при получении контроллером сообщения должен в нем отыскиваться, указывая на объект (например, компонент EJB сервера приложений J2EE), которому сообщение направлено.

Этот подход применим для любых протоколов – горизонтальных и вертикальных, лишь бы сообщения имели в заголовке правильный идентификатор. Однако он требует стандартизации, поскольку все взаимодействующие участники должны знать, как идентификатор размещен в сообщении, и вставлять его в каждое отправляемое сообщение. Стандартный подход не требует введения в описания WSDL данных для сопоставления сообщений, оно будет проводиться автоматически.

Другой функцией контроллеров разговоров является верификация разговоров на соответствие протоколу, описание которого должно передаваться контроллеру на некотором языке протоколов. Находясь на пути всех сообщений, контроллер при обнаружении несоответствий может вместо передачи сообщения на обработку возбуждать сообщение об ошибке. Эта функция контроллеров также упрощает разработку сетевых служб.

Контроллеры разговоров создают обобщенную протокольную инфраструктуру, которая может поддерживать любые протоколы. Кроме этого обобщенного компонента, системная часть сетевой службы может содержать модули, реализующие специфические координационные протоколы, то есть зависящие от конкретных протоколов программы, генерирующие сообщения в соответствии с правилами, определенными именно для этих протоколов. Такие модули называются *модулями управления протоколами*. Эти модули могут поддерживать выполнение протоколов двумя способами:

1. Модуль получает, интерпретирует и отправляет протокольные сообщения автоматически без вмешательства сетевой службы. Таким образом могут реализовываться протоколы надежной доставки. При этом программы хранения сообщений до момента их доставки адресату, выполнения повторных отправок, отправки и получения подтверждений о доставке целиком размещаются в системной части.
2. Модуль управления протоколами и сетевые службы совместно реализуют протокол. Например, в случае протокола 2PC программы доставки/получения подготовительных сообщений, подтверждений и отмен реализуются системными средствами. Принятие решения о подтверждении или отмене транзакции и программы, реализующие эти операции (например, что означает "подтвердить"), выполняются сетевыми службами, которые должны понимать настоящую логику процесса.

При своей работе контроллер разговоров направляет сообщения бизнес протокола соответствующей реализации сетевой службы, содержащей прикладные программы, реализующие протокол. Сообщения, относящиеся к горизонтальному протоколу, вместо этого направляются в модуль управления. В принципе реализация службы и модуль управления протоколом выглядят для контроллера разговоров одинаково. Контроллеры разговоров и модули управления протоколами можно совмещать.

Реализация горизонтальных протоколов в слое системной поддержки, а не в самой сетевой службе, накладывает ряд дополнительных требований. Для обмена сообщениями службы должны знать порты друг друга. Эти порты могут разыскиваться в процессе привязки (например, с помощью реестров UDDI). Однако, если реализация протокола передана в системный слой, розыском портов должны заниматься модули управления протоколами. Необходимо иметь механизм передачи в систему сведений о портах.

Сетевые службы при выполнении протоколов потенциально могут играть различные роли. Если реализация протокола выполняется системой автоматически, в систему должны передаваться сведения о роли, выполняемой в данном разговоре.

Независимо от выполняемых функций (маршрутизация разговоров, верификация протоколов или реализация горизонтальных протоколов) системная поддержка сетевых служб должна опираться на стандарты. Во-первых, для сопоставления сообщений разговорам и направления их объектам, управляющим разговорами, необходим способ генерации и транспортирования в заголовках сообщений SOAP уникальных идентификаторов разговоров. Во-вторых, для согласования вопросов выбора и координации протокола необходим способ этого согласования и набор протоколов (называемых метапротоколами), поскольку без этого сетевые службы не смогут договориться друг с другом. В-третьих, необходима стандартизация горизонтальных протоколов, без чего их выполнение должно оставляться разработчикам сетевых служб, затрудняя разработку этих служб и увеличивая гетерогенность систем. Наконец, для того, чтобы контроллеры разговоров могли интерпретировать спецификации протоколов и верифицировать согласованность сообщений, необходимы стандартизованные языки протоколов. Все это и многое другое отсутствует в спецификациях SOAP, WSDL и UDDI.

Таким образом, широкое использование средств системной поддержки взаимодействия сетевых служб может начаться с полноценного всеобщего принятия стандартов, определяющих это взаимодействие. Одним из таких стандартов, предложенным компаниями IBM, Microsoft и BEA в августе 2002 года, является *стандарт WS-Coordination*, определяющий:

- Метод передачи уникальных идентификаторов сетевым службам, взаимодействующим между собой. В частности, стандарт определяет координационный контекст и то, как он должен включаться в заголовки сообщений SOAP.

- Метод передачи контроллерам разговоров сведений о портах участников разговоров. Для этого стандарт определяет интерфейс регистрации портов.
- Метод передачи контроллерам разговоров сведений о ролях, которые они должны играть в разговоре. Для этого стандарт определяет интерфейс активации.

Важнейшее место в стандарте WS-Coordination занимают понятия "*координатор*" и "*участник*". Для описания взаимодействий между координатором и участниками стандарт WS-Coordination вводит три абстракции:

- ***Координационный протокол*** есть набор правил управления разговорами координатора с участниками (например, 2PC).
- ***Координационный тип*** представляет собой набор логически связанных друг с другом координационных протоколов. Например, координационный тип атомарных транзакций будет включать в себя группу из двухфазного подтверждения и протокола уведомления, выполняющегося между участниками, желающими получить информацию о результате 2PC.
- ***Координационный контекст*** – это структура данных, используемая для отметки сообщений, относящихся к одному разговору (в протоколе WS-Coordination это называется "к одной координации").

Стандарт WS-Coordination определяет три формы взаимодействий между координатором и участниками:

- ***Активация***. Участник требует от координатора создать новый координационный контекст. Новые контексты создаются всякий раз, когда участник создает новый экземпляр координационного типа (разговор), например, когда сетевая служба начинает атомарную транзакцию.
- ***Регистрация***. Участник регистрируется у координатора как участник координационного протокола. Эти сетевая служба объявляет, что участвует в выполнении протокола и ее следует уведомлять о выполнении определенных шагов протокола.
- ***Взаимодействие, определяемое протоколом***. Координатор и участники обмениваются сообщениями, специфичными для данного прикладного протокола.

Взаимодействия, проводимые для активации и регистрации транзакций, не зависят от типа координации (*горизонтальны*). Это

позволяет реализовывать их как координаторами, так и участниками в качестве части спецификации WS-Coordination. С другой стороны интерфейсы, нужные для проведения взаимодействия, специфичного для конкретного протокола, оказываются разными для разных протоколов и не могут быть определены в рамках WS-Coordination.

4.7.2. Централизованная координация

В процессе обмена сообщениями при централизованной координации (с одним общим координатором) выдаются сообщения следующих трех типов (*Рис. 4.20*):

- ***Операционные сообщения***. Эти сообщения передаются между взаимодействующими службами.
- ***Сообщения протокола WS-Coordination***. Эти сообщения передаются между службами и координатором при проведении активации или регистрации.
- ***Протокольные сообщения***. Эти сообщения передаются между службами и координатором как часть основного протокола.

Из этих трех типов сообщений протокол WS-Coordination определяет только сообщения второго типа. Другие сообщения зависят только от самих сетевых служб и от координационных протоколов, которые они используют. Предполагается, что координатор знаком с основным протоколом и знает, как поддержать его выполнение.

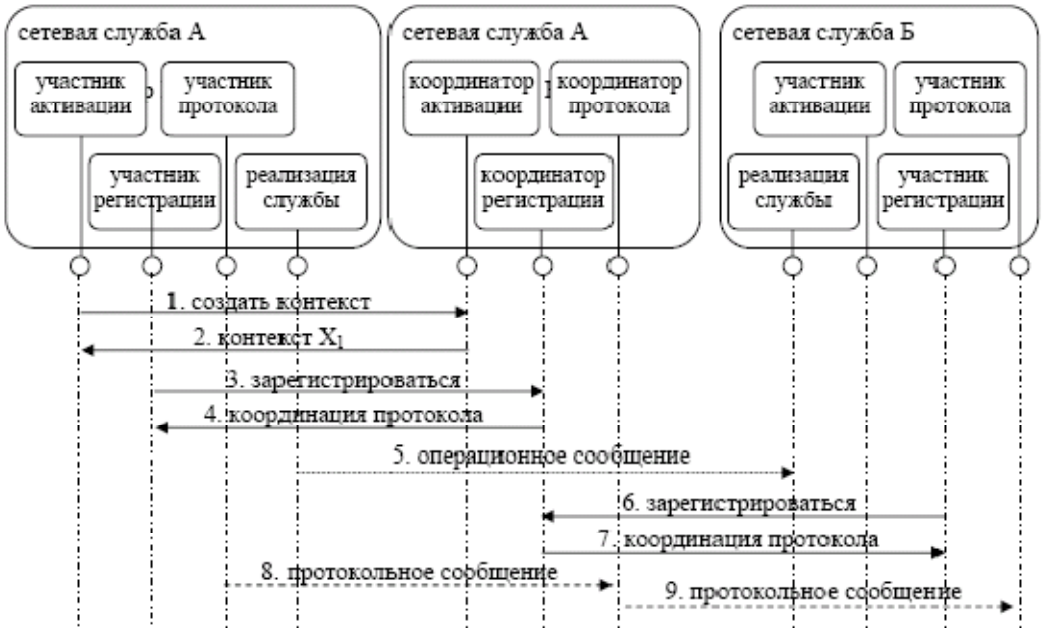


Рис. 4.20. Обмен сообщениями при выполнении разговора с централизованной координацией (после сообщения № 7 все стороны знают, кто будет координировать протокол, и кто будет в нем участвовать).

Реализация различных типов координаций проводится протоколом WS-Coordination в следующей последовательности. Во-первых, все сетевые службы, участвующие в разговоре, должны согласовать между собой, кто будет координировать данный разговор. Протокол WS-Coordination решает эту проблему распространением структуры данных для координационного контекста и указанием метода передачи этой структуры между сетевыми службами. Указанная структура данных содержит ссылку на координаторный регистрационный порт, с помощью которого все участники могут регистрировать свой интерес к разговору у одного и того же координатора. Во-вторых, многие типы координаций требуют передачи между участниками уникального идентификатора, который обеспечит автоматическую маршрутизацию верификацию протокола на уровне системной поддержки. Протокол WS-Coordination определяет общий механизм определения таких идентификаторов и их передачи между службами, что также выполняется с помощью координационной структуры данных. В-третьих, в любом разговоре возникает необходимость связывания координаторов и участников. Без такого связывания координатор не узнает ссылку на протокольный интерфейс участника (и наоборот).

4.7.3. Децентрализованная координация

Протокол WS-Coordination позволяет участникам взаимодействовать с использованием персональных координаторов, что является обычным режимом в Интернете, где взаимодействие часто децентрализовано (*Рис. 4.21*).

В отличие от централизованного варианта участники могут регистрироваться у разных координаторов, причем каждый из них создает свой координационный контекст, а координаторы должны регистрировать друг друга, причем один из них объявляет себя координатором всего процесса взаимодействия, а другие участвуют во взаимодействии в качестве посредников.

Распределенность координации достигается построением цепочек координаторов. Один координатор работает как заместитель (*proxy*) другого. Все сообщения между координатором одной службы и другой службой проходят через координатор этой другой службы. Для того чтобы реализовался такой сценарий, в приведенном примере задействованы два механизма. Во-первых, один координатор, чтобы действовать в качестве координатора-заместителя другого, должен знать свою роль в протоколе. Во-вторых, любой координатор должен иметь возможность переправлять

полученные сообщения от своей службы к другому координатору и обратно.

Рис. 4.21. Обмен сообщениями при выполнении разговора с децентрализованной координацией (после сообщения № 11 известно, кто будет координировать протокол, и кто будет в нем участвовать).

Активация или создание координационного контекста имеет двойную цель: для участника она состоит в получении нового контекста, а для координатора в осознании собственной роли в протоколе. Если существующий координационный контекст передается как часть сообщения создания координационного контекста, координатор понимает, что он является заместителем некоторого другого координатора. Ссылка на порт регистрации координатора уже включена в переданный координационный контекст. С ее помощью координатор-заместитель узнает ссылку на первичный координатор. С другой стороны, если сообщение с запросом на создание координационного контекста не имеет существующего координационного контекста, координатор действует как первичный.

Построение цепочек координаторов может привести к получению произвольного уровня сложности связей, однако цепочками система координаторов может не ограничиваться. В соответствии со стандартом WS-Coordination координаторы могут транслировать сообщения одного протокола, получаемые от их участников, в сообщения другого протокола, передаваемые другому координатору. Естественно, что это требует

реализации зависящих от конкретных протоколов компонентов, выполняющих активацию и регистрацию.

4.8. Транзакции в сетевых службах

Протокол WS-Coordination создает основу реализации других протоколов сетевых служб, в том числе для тех важнейших протоколов, которые смогут поддерживать транзакционный обмен. В августе 2002 года фирмы IBM, Microsoft и BEA предложили на базе протокола WS-Coordination набор спецификаций, создавших новый стандарт *WS-Transaction*.

Отсутствие централизованной системной платформы – это не единственная особенность в работе с сетевыми службами. Еще одно их отличие от традиционных систем заключается в длительности операций, выполняемых с их помощью. Интегрируя работу различных прикладных систем, сетевые службы при выполнении транзакций могут требовать в отдельных случаях выполнения ручных операций. Следствием этого может оказаться, что сохранение свойств обычных транзакций, то есть – атомарности, непротиворечивости, изолированности и долговечности, при выполнении двухфазного подтверждения их завершения станет невозможным.

Дополнительным отличием является недостаточная проработка моделей ресурсов и операций. В системах управления базами данных имеются четкие определения того, что означают термины "ресурс", "блокировка", "подтверждение" и "откат". Модель базы данных очень точно описывает, что блокируется в период выполнения транзакции, что происходит в момент подтверждения. В отношении сетевых служб ничего подобного не существует. Операции WSDL могут быть произвольными, от вставки записи в базу данных до отправки письма заказчику. Откат транзакции может означать самое разное, в зависимости от того, что это за транзакция. Например, откат отправки письма заказчику может заключаться в отправке еще одного письма тому же заказчику с просьбой, считать первое письмо недействительным.

Таким образом, работая с сетевыми службами, приходится переходить к выработке *компенсационных механизмов*. Идея, лежащая в основе этого подхода заключается в том, что любая сетевая служба, участвующая в транзакционном обмене может писать в сохранную память (менять свое состояние) после выполнения очередного шага транзакции, делая тем самым, результаты этого шага видимыми за пределами транзакции. Это нарушает свойства атомарности и изолированности. Если по каким-то причинам транзакцию надо будет отменить, сетевая служба

выполнит компенсационную операцию, которая семантически отменит (частичные) результаты выполнения транзакции. Каждая операция может иметь свою собственную компенсационную логику, каждый поставщик служб может иметь собственное мнение и собственный способ выполнения компенсационных операций.

Независимо от того, как проводится внутренняя реализация транзакций и компенсаций, всем участникам транзакций необходимо следовать некоторым стандартным протоколам, которые помогают определять, что некоторый шаг транзакции не удался, и просить остальных участников выполнить компенсационные операции. Стандарт *WS-Transaction* определяет стандартный протокол для долгих транзакций, называемых *бизнес активностями*. Чтобы сетевые службы не потеряли возможности работать с короткими транзакциями в доверительных зонах, стандарт WS-Transaction определяет также атомарные транзакции.

Стандарт WS-Transaction определяет набор протоколов, требующих скоординированной работы нескольких сторон, следовательно, он строится на базе протокола WS-Coordination. В частности, стандарт WS-Transaction предполагает существование набора сетевых служб, участвующих в транзакции и одного или нескольких координаторов, централизованных или распределенных. Стандарт WS-Transaction также определяет структуру координационного контекста и стандартные WSDL-интерфейсы, которые должны реализовываться участниками и координаторами. Транзакционная семантика реализуется комбинацией двух протоколов WS-Coordination и WS-Transaction, выполняемых с поддержкой со стороны координаторов.

4.8.1. Атомарные транзакции

Первым из координационных типов, определяемых стандартом WS-Transaction, является тип *атомарных транзакций*. Этот тип состоит из нескольких координационных протоколов, исполняемых последовательно или альтернативно сетевыми службами участниками или координаторами, в зависимости от того, что должно делаться на протяжении той или иной фазы распределенной транзакции. Всего в этот координационный тип входит пять протоколов: завершения, завершения с уведомлением, двухфазного подтверждения, нулевой фазы и уведомления о результате.

Когда сетевая служба хочет завершить транзакцию, она выполняет протокол завершения с координатором. Ее целью при этом является информирование координатора о том, что он должен инициировать протокол двухфазного подтверждения для проверки результата транзакции и опросить всех участников об успешности проведения транзакции. После

окончания двухфазного подтверждения и завершения транзакции, ее результат передается сетевой службе.

В некоторых случаях координатор до двухфазного подтверждения проводит с участником обмен по протоколу нулевой фазы. Это бывает нужно, чтобы оповестить участника о предстоящем начале двухфазного подтверждения. Участник может при этом провести свои подготовительные работы, например, записать информацию из оперативных буферов в память. Такое действие необходимо в тех реализациях баз данных, в которых после начала двухфазного подтверждения изменения в базы вносить не разрешается. В любой точке в процессе или после выполнения протокола двухфазного подтверждения участник может запросить координатора о результате транзакции, что делается выполнением протокола уведомления о результате. Наконец, протокол завершения с уведомлением может инициироваться сетевой службой как альтернатива протоколу завершения в тех случаях, когда нужно, чтобы координатор хранил результат транзакции, не уничтожая, до получения специального уведомления от сетевой службы о том, что этот результат ею получен.

Для каждого из пяти протоколов стандарт вводит (в дополнение к портам, специфицированным стандартом WS-Coordination) по два типа портов, которые должны быть реализованы координатором. Один из пары типов портов позволяет координатору выполнять протоколы в качестве координатора, а второй – в качестве участника (это необходимо для организации координационных цепочек, когда координаторы взаимодействуют друг с другом). Сетевые службы должны реализовывать только половину типов портов, поскольку всегда являются только участниками взаимодействий.

Сетевые службы не обязаны реализовывать все типы портов. Номенклатура портов зависит от той роли, которую сетевая служба предполагает играть в транзакционных обменах.

Протоколы завершения и завершения с уведомлением нужны сетевым службам, которые обращаются к координаторам с целью подтвердить транзакцию или выполнить ее откат. Протокол двухфазного подтверждения нужен тем сетевым службам, которые вносят изменения в состояния баз данных, требующие подтверждения или отката. Если сетевая служба выполняет другие формы изменения состояния (для повышения производительности отражаемые в оперативной памяти или в буферах) и нуждается в предварительном оповещении о предстоящем выполнении протокола двухфазного подтверждения, ей нужно реализовывать порт

нулевой фазы. Получив соответствующее сообщений, сетевая служба сможет заблаговременно (до начала выполнения протокола 2PC, некоторые реализации которого препятствуют внесению изменений в базу данных после начала протокола) внести все необходимые изменения в базу данных. Наконец, если сетевая служба желает проверять результаты транзакций, она должна иметь порт протокола уведомления. На *Рис. 4.22* приведена диаграмма состояний наиболее сложного протокола атомарных транзакций – двухфазного подтверждения.

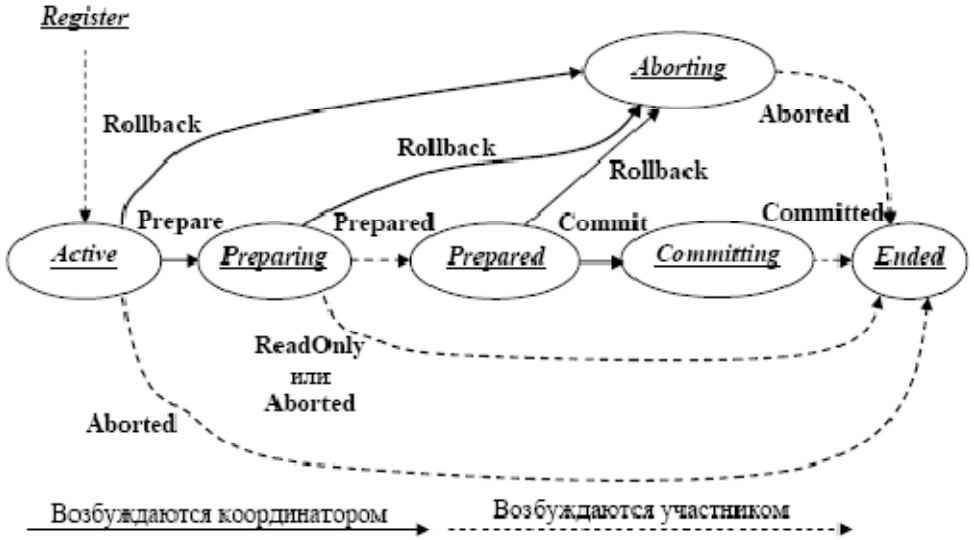


Рис. 4.22. Диаграмма состояний протокола 2PC атомарной транзакции.
Сообщение **ReadOnly** *показывает, что участник голосует за подтверждение, но сам не нуждается в дальнейшем участии в протоколе.*
Сообщение **Aborted** *показывает, что участник голосует против подтверждения и не нуждается в дальнейшем участии в протоколе.*

Кроме протоколов, стандарт WS-Transaction определяет структуру транзакционного контекста. Именно эта структура возвращается координатором в ответ на запрос о создании координационного контекста. Она же передается как часть заголовков сообщений SOAP и показывает, что сообщение посылается в рамках некоторого разговора.

Дополнительные координаторы, составляющие цепочки произвольной длины, работают как посредники, передавая получаемые ими от главных координаторов сообщения ранее зарегистрированных протоколов службам, зарегистрировавшимся у них для участия в координационных протоколах.

Необходимо помнить, что стандарт WS-Transaction специфицирует только часть бизнес логики. Семантика понятий "подтверждение" и "отказ от подтверждения" определяется неформально, и хотя их общий смысл

понятен, поведение различных сетевых служб при подтверждении или при отказе от него может существенно различаться.

4.8.2. Бизнес активности

Чтобы иметь возможность управлять длительными бизнес транзакциями, выполняемыми сетевыми службами, и при этом не блокировать ресурсы использованием протокола двухфазного подтверждения, стандарт WS-Transaction определяет другой координационный тип, называемый *бизнес активностью*. Этот координационный тип содержит два протокола: *бизнес соглашение с завершением участника* и *бизнес соглашение с завершением координатора*.

Протокол бизнес соглашения с завершением участника инициируется сетевой службой участника с целью проинформировать координатора о состоянии выполнения, которое может, например, иметь значения: *прервано*, *завершено*, *ошибка*. После достижения консенсуса, продолжать ли выполнение транзакции или прервать ее, координатор отвечает всем участникам сообщениями типа *закрыть*, *завершить*, *компенсировать, отменить* или *ошибка*.

Протокол бизнес соглашения с завершением координатора похож на протокол бизнес соглашения с завершением участника, он отличается тем, что каждый участник в данном случае уверен, что координатор предварительно оповестит его, что получил все запросы на выполнение работ в рамках данной бизнес активности. Это дает возможность участнику подготовиться к следующим за этим операциям завершения или компенсации. На *Рис. 4.23* и *Рис. 4.24* приведены диаграммы состояний протоколов бизнес активностей.

Координатор в соответствии со стандартом должен иметь две пары типов портов (одна пара – для соглашения с завершением участника, а вторая для соглашения с завершением координатора). Один из пары типов портов позволяет координатору выполнять протоколы в качестве координатора, а второй – в качестве участника (как и раньше, это необходимо для организации координационных цепочек, когда координаторы взаимодействуют друг с другом). Сетевые службы должны реализовывать только один или два типа портов, в зависимости от типа используемого протокола.

Реализация компенсационных операций зависит от конкретной сетевой службы и не рассматривается как часть бизнес логики. Более того, для каждой бизнес активности и для каждой сетевой службы может определяться только одна компенсационная операция. Это означает, что

если сетевая служба в рамках некоторой бизнес активности выполняет в некотором порядке серию заданий, все эти задания должны отменяться в совокупности.

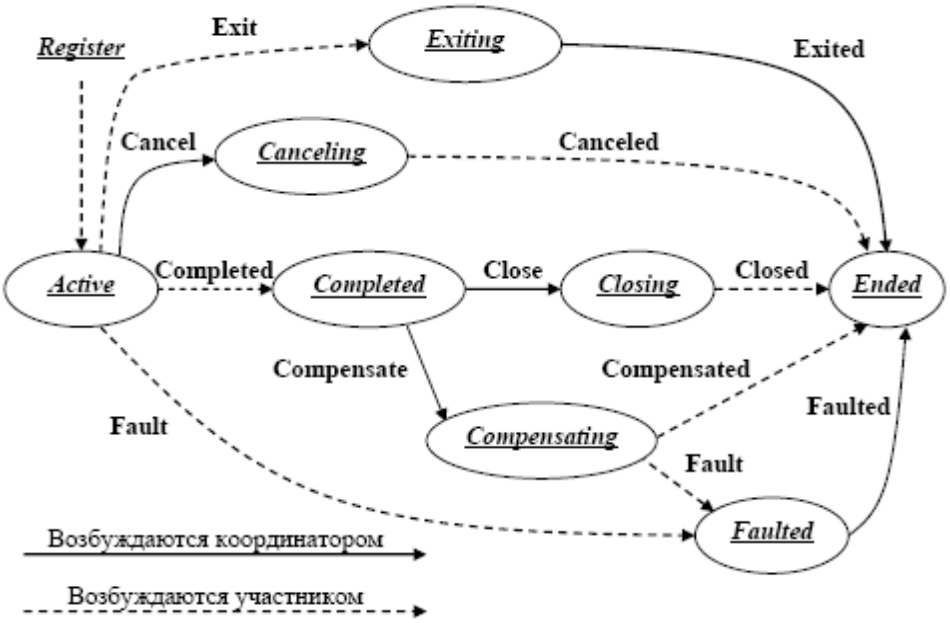


Рис. 4.23. Диаграмма состояний протокола бизнес соглашения с завершением участника.

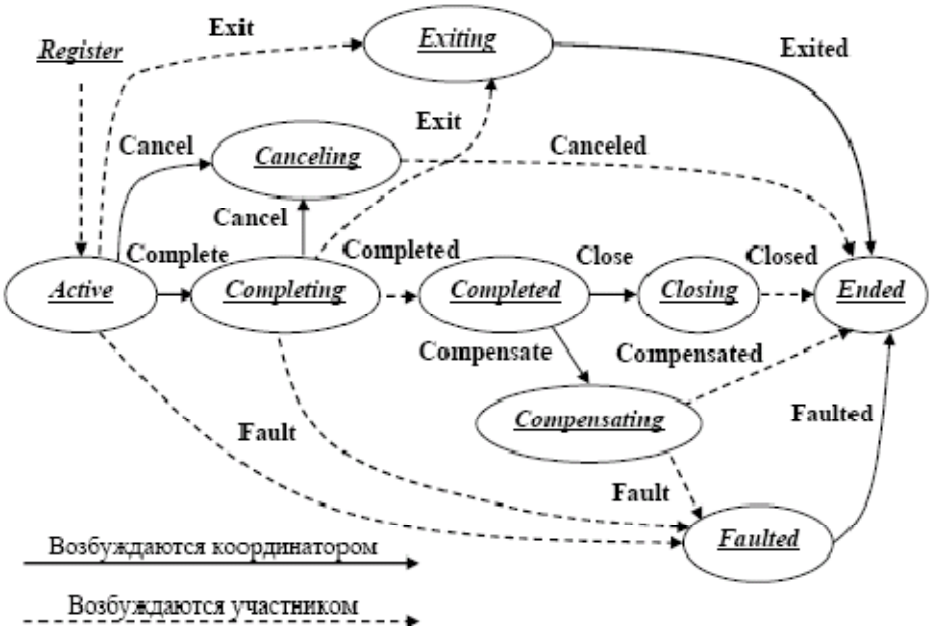


Рис. 4.24. диаграмма состояний протокола бизнес соглашения с завершением координатора.

5. Композиция сетевых служб

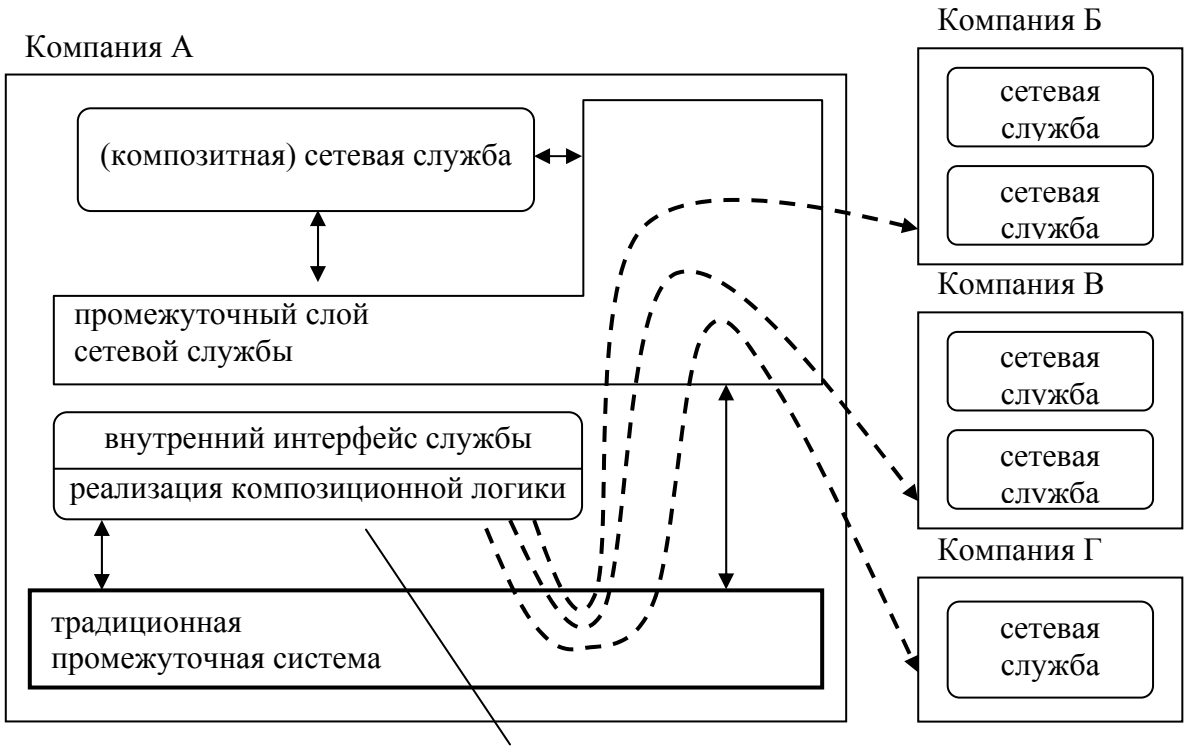
Если координация служб позволяет нескольким службам участвовать в одном разговоре между собой, то их композиция обеспечивает службам возможность одновременно вести несколько разговоров с разными службами (*Рис. 5.1*). Композиция сетевых служб напоминает процессы, происходящие в системах управления рабочим потоком, где бизнес логика интегрированного приложения реализуется композицией других, автономных крупноблочных приложений. При этом, в общем случае, различные сетевые службы могут поддерживать разные протоколы, следовательно, клиент должен сам реализовывать все протоколы, необходимые для обращения к службам.

Рис. 5.1. Клиенты могут участвовать в различных разговорах с разными сетевыми службами.

Одно из наиболее интересных свойств композиции сетевых служб состоит в том, что добавлением составных компонентов она позволяет создавать сколь угодно сложные приложения. Таким образом, композицию служб можно рассматривать в качестве средства управления сложностью, в котором службы строятся из других служб, находящихся на более низком уровне абстракции.

В отличие от традиционных композиционных структур (как в программировании, так и производственной деятельности) композиция сетевых служб не предполагает физической интеграции компонентов. Сетевые службы это не библиотеки прикладных программ, которые надо

транслировать и объединять с другими частями приложений. Напротив, их надо рассматривать как интерфейсы, которыми надо пользоваться, обращаясь к ним. Как и в системах интеграции приложений на уровне предприятий, композиция сетевых служб эквивалентна указанию, к каким службам надо обратиться, в каком порядке это сделать, как надо управлять исключительными ситуациями. Базовые компоненты остаются отделенными от композитных служб.



Внутреннее приложение реализует композиционную логику, обращаясь при необходимости к сетевым службам. В данном случае какая-либо поддержка в промежуточном слое отсутствует.

Рис. 5.2. В отсутствие системной поддержки композиции сетевых служб, реализация композитной службы проводится традиционными средствами.

Для композиции сетевых служб требуется системная поддержка. В настоящее время наиболее широко распространен подход, когда программирование композитных сетевых служб ведется средствами традиционных языков, например, языка Java. Сетевые службы более всего используются в качестве мостов между гетерогенными системными платформами, а в подобном окружении разработка служб и их композиция традиционно выполняется непосредственно программированием на некотором языке. Однако языки программирования разрабатывались, не имея в виду потребности композиции сетевых служб. В получающихся программах бизнес логика оказывалась перемешанной с низкоуровневыми деталями, что означало сложности не только в процессе разработке композиции, но и при сопровождении (*Рис. 5.2*). Сложившаяся ситуация

привела к осознанию необходимости моделей высокого уровня, и для композиции сетевых служб было предложено много разных языков (XL, WSFL, BPML, BPEL).

5.1. Основные элементы системной поддержки композиции сетевых служб

Композиция служб – это технология реализации. Она определяет, как надо реализовывать сетевую службу, соединяя между собой другие службы. Соответственно, системная поддержка должна заключаться в определении абстракций и инструментария, которые помогут четко описать и исполнить запрос к сетевой службе, позволив разработчикам концентрироваться не на низкоуровневых деталях, а на бизнес логике. Системная поддержка включает (*Рис. 5.3*):

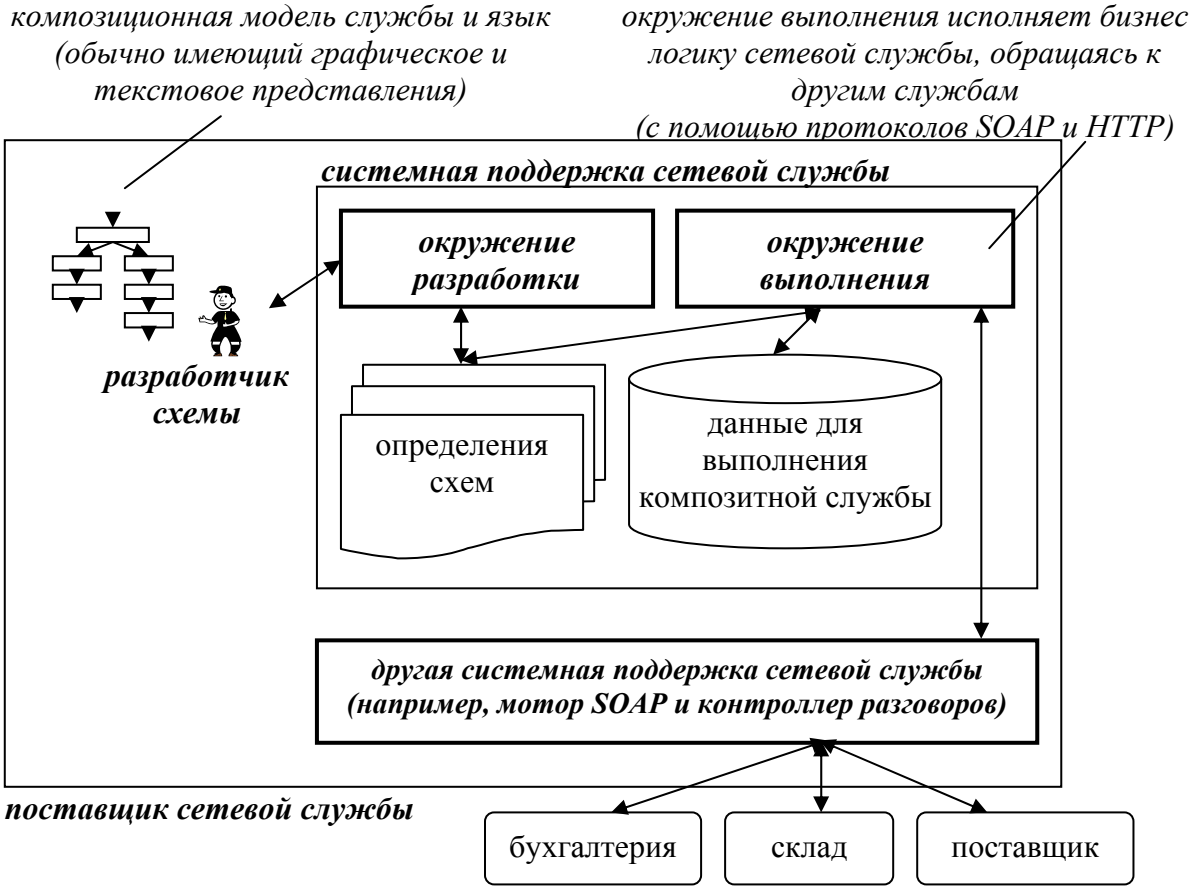


Рис. 5.3. Архитектура, ориентированная на сетевые службы, модернизированные (децентрализованные) протоколы и стандартизацию.

- ***<u>композиционную модель и язык</u>***, позволяющие описывать объединяемые службы, порядок в котором к ним надо обращаться, а также способ определения параметров обращений к службам. Спецификация композитной службы, выраженная на языке композиции, называется композиционной схемой. Схема определяет бизнес логику

композитной сетевой службы, она может выглядеть как программа, написанная на языке, специально разработанном для композиции.

- ***<u>окружение разработки</u>***, обычно характеризующееся графическим пользовательским интерфейсом, с помощью которого разработчики могут создавать композиционные схемы, а затем создавать графы зависимостей, обозначая порядок, в котором надо обращаться к службам. Графы и другая описательная информация транслируются в текстовые спецификации (композиционные схемы).
- ***<u>окружение выполнения</u>***, называемое иногда композиционным мотором. Это окружение выполняет бизнес логику композитной службы, обращаясь к службам компонентам, определенным в схеме. Каждое отдельное выполнение композитной службы называется композиционным примером.

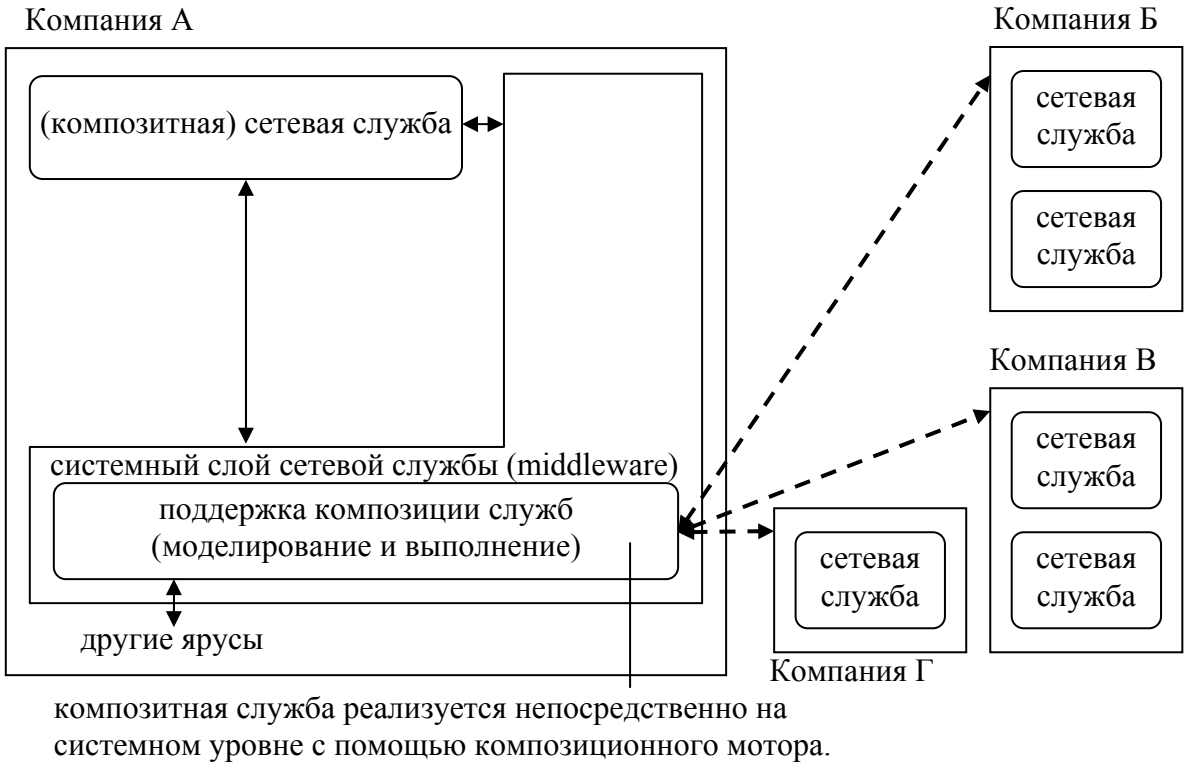


Рис. 5.4. Используя системную поддержку композиции сетевых служб, реализация композитной службы проводится внутри системного слоя.

Применение системной поддержки для композиции приводит к реализации композитной службы на системном уровне сетевых служб, а не на уровне традиционной системной поддержки (*Рис. 5.4*).

5.2. Системная поддержка композиции и координации

Спецификации композитных служб выполняются разработчиками и являются их собственностью. Их не сообщают клиентам и нигде не

регистрируют. Спецификации композиций предназначены для системных слоев сетевых служб, которые автоматизируют композицию, обращаясь к операциям, предлагаемым другими сетевыми службами в соответствии с композиционной схемой (*Рис. 5.5*).

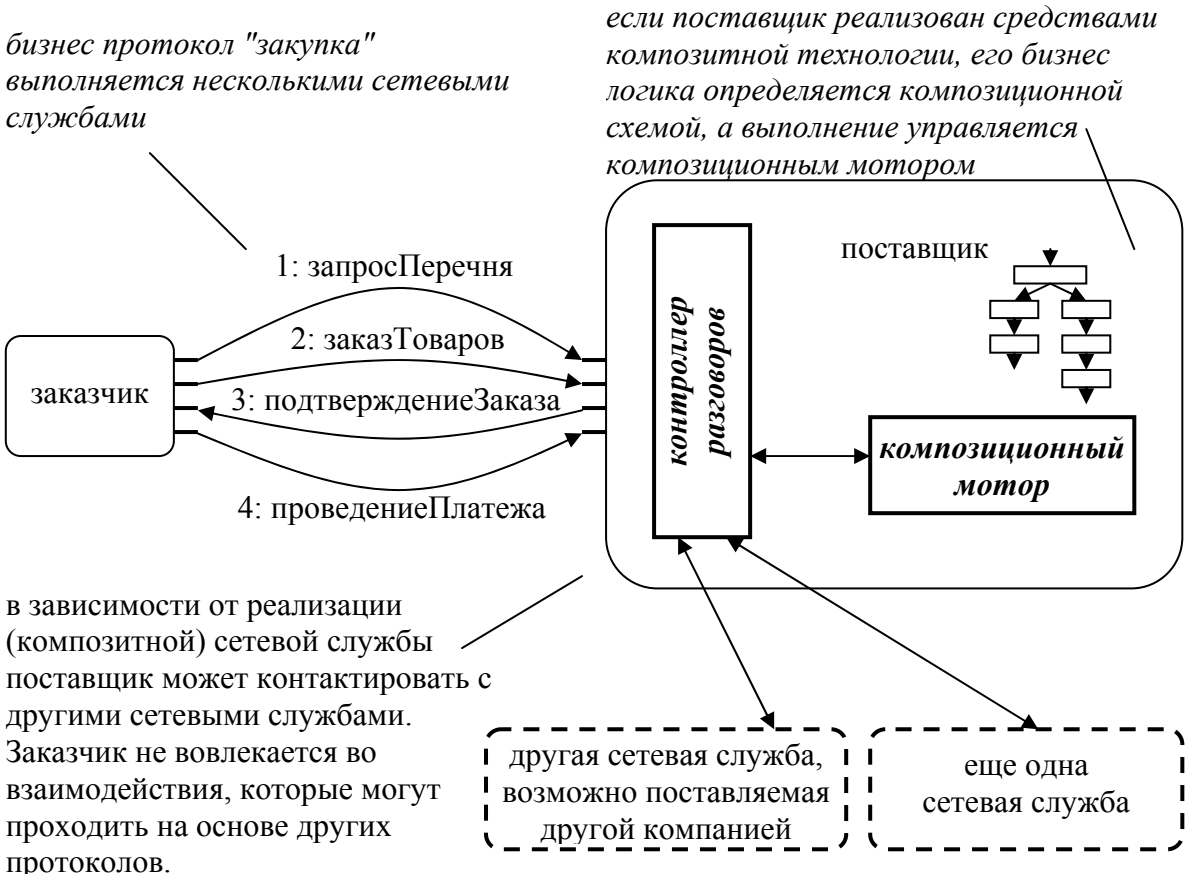


Рис. 5.5. Композиция и координационные протоколы имеют разные области применения: внешние взаимодействия и внешняя реализация.

С точки зрения клиента все равно, является сетевая служба композитной или нет. Клиент не вникает в то, как реализована служба, с помощью традиционных языков программирования или на основе технологии композиции служб.

Тем самым, область применения и цели композиции резко контрастируют с целями координации. Координационные протоколы – это общедоступные документы, создаваемые стандартизирующими консорциумами и на основе стандартных языков. Эти документы регистрируются в реестрах сетевых служб, их целью является поддержка поиска при разработке и привязки при выполнении. Разговоры, подчиняющиеся координационным протоколам, поддерживаются контроллерами разговоров, цель которых связана не с выполнением бизнес логики, а с диспетчеризацией сообщений, приходящих для внутренних

объектов, и с верификацией правил протоколов. Контроллеру все равно, с кем ведется разговор, с базовой службой или с композитной.

Итак, имеется четкое различие между *внутренней композицией* и *внешней координацией* сетевых служб.

5.3. Композиционные модели сетевых служб

Композиция сетевых служб по смыслу выполняемых мероприятий очень близка понятию рабочих потоков, но выполняется на другом уровне стандартизации интегрируемых объектов. Терминология, применяемая при описании композиционных моделей, близка к терминологии систем управления рабочими потоками. Термин *определение процесса* (или просто *процесс*) относится к *композиционной схеме*, *пример процесса* – это конкретное, индивидуальное выполнение определения процесса. Термин *схема оркестровки* или просто *оркестровка* относится к части композиционной схемы, описывающей порядок, в котором должны вызываться отдельные компоненты службы.

В качестве первого шага определения композиционной модели обычно вводятся следующие определения:

- ***Компонентная модель***. Определяет природу объединяемых элементов в терминах предположений, которые делаются моделью по поводу таких компонентов.
- ***Оркестровая модель***. Определяет абстракции и языки, используемые для определения порядка, в котором должны вызываться службы. Имеются различные варианты моделей: диаграммы активности, сети Петри, π-исчисление, диаграммы состояний, иерархии активностей, оркестровка на основе правил.
- ***Модель данных и доступа к данным***. Определяет методы описания данных и обмена данными между компонентами.
- ***Модель выбора службы***. Определяет способ статической или динамической привязки, то есть, каким образом в качестве компонента выбирается та или иная конкретная служба.
- ***Транзакции***. Определяет, какая транзакционная семантика может быть ассоциирована с композицией и как это делается.
- ***Управление исключениями***. Определяет, как можно управлять исключительными ситуациями, возникающими при выполнении композитной службы, с целью предотвращения прерывания работы.

5.3.1. Компонентная модель

Типы компонентов, включаемых в сетевую службу, и предположения, делаемые о них, в существенной степени отличают одни

службы от других. Одна крайность заключается в том, что модель может предполагать, что компоненты реализуют определенный набор стандартов сетевых служб, например, HTTP, SOAP, WSDL и WS-Transaction. Такие предположения снижают гетерогенность системы. Другая крайность заключается в ограничении самыми общими предположениями. Например, можно ограничиться лишь тем, что компоненты взаимодействуют, обмениваясь XML-сообщениями синхронно (в стиле RPC), либо асинхронно. Модель становится более общей, но следствием может стать усложнение работы по созданию службы.

Промежуточным решением может быть одновременное следование нескольким разным моделям и разработка специальных средств для компонентов, не входящих ни в одну из поддерживаемых моделей. Такая открытость приводит к использованию более сложных языков и систем, которым требуется поддерживать множество форматов и протоколов.

В настоящее время наиболее перспективный язык композиции BPEL предполагает, что компонентами являются службы, описанные на WSDL. Он также полагается на другие стандарты, вроде XPath и WS-Addressing. Будущие версии могут быть интегрированы с протоколом WS-Transaction.

5.3.2. Оркестровая модель

Оркестровка позволяет различным службам организоваться в единое целое. В этой модели описывается порядок, в котором вызываются службы, а также условия, в соответствии с которыми определенная служба может вызываться или не вызываться.

Предположим, поставщик сетевой службы позволяет заказчикам размещать заказы вызовом операции *заказТовара*. Поставщик, реализованный на основе технологии композиции служб, выполняет операцию, вызывая другие службы. Его бизнес логика, следовательно, определяется композиционной схемой. На *Рис. 5.6* приведена оркестровка композиционной схемы, моделируемой средствами диаграмм активности унифицированного языка моделирования UML. Диаграммы активности являются наиболее широко используемой парадигмой моделирования, как в традиционных рабочих потоках, так и в сетевых службах. В этой парадигме предполагается, что оркестровки от начала выполнения до самого конца определяются описанием последовательности операций.

Бизнес логика сетевой службы, рассматриваемой в качестве примера, такова: когда заказчик вызывает операцию *заказТовара*, организуется новый запуск композитной службы, которая вызывает операцию *проверитьСклад*, выполняемую локальной сетевой службой. Эта операция используется поставщиком для проверки наличия товара на складе. Если

товар обнаружен, поставщик подтверждает заказ заказчику вызовом операции *подтвердитьЗаказ*, которая выполняется сетевой службой заказчика. В противном случае поставщик входит в контакт с оптовым складом и проверяет наличие товара там. Если товар на складе есть, следует подтверждение заказчику, в противном случае заказ не принимается.

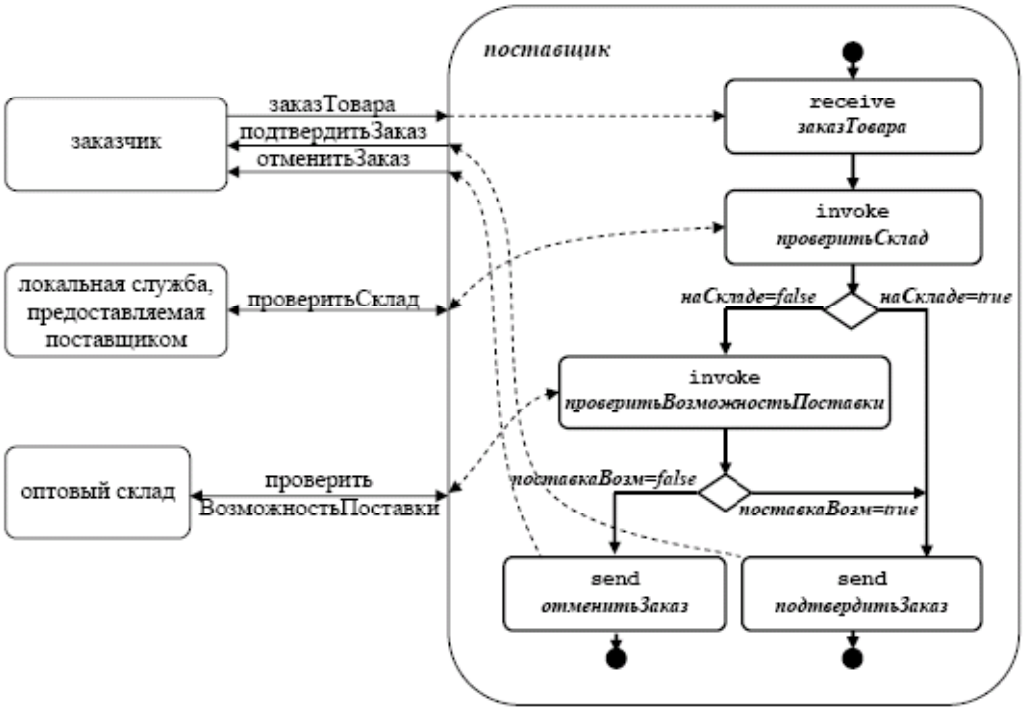


Рис. 5.6. Модель сетевой службы поставщика в виде диаграммы активности. Пунктирными линиями отмечены отношения между (внутренними) активностями и (внешними) протокольными сообщениями.

Диаграмма *Рис. 5.6* предполагает, что активности всегда моделируют уведомления о сообщениях (получение сообщений), приходящих к (поступающих от) сетевой службе:

- Уведомления о сообщениях другим сетевым службам (вызовы односторонних операций, предлагаемых компонентами сетевых служб) моделируются средствами активности `send` (*послать*), например, `send` *отменитьЗаказ* в примере. Такие активности не являются блокирующими.
- Обращения к синхронным (запрос/ответ) операциям, предлагаемым другой сетевой службой, моделируются активностью `invoke` (*вызвать*), в примере - `invoke` *проверитьСклад*. Эта активность является блокирующей, поскольку она ждет ответа от вызываемой службы.

- Получение сообщений, относящихся к компонентам служб, вызывающим односторонние или двухсторонние операции, предлагаемые композитной службой, моделируются активностью `receive` (*получить*), например, `receive` *заказТовара*. Это тоже блокирующие активности, поскольку выполнение композитной сетевой службы не может быть продолжено до получения сообщения.
- Если полученное сообщение вызывает двухстороннюю операцию, композиционная схема будет включать активность `reply` (*ответить*), которая будет посылать ответ клиенту.

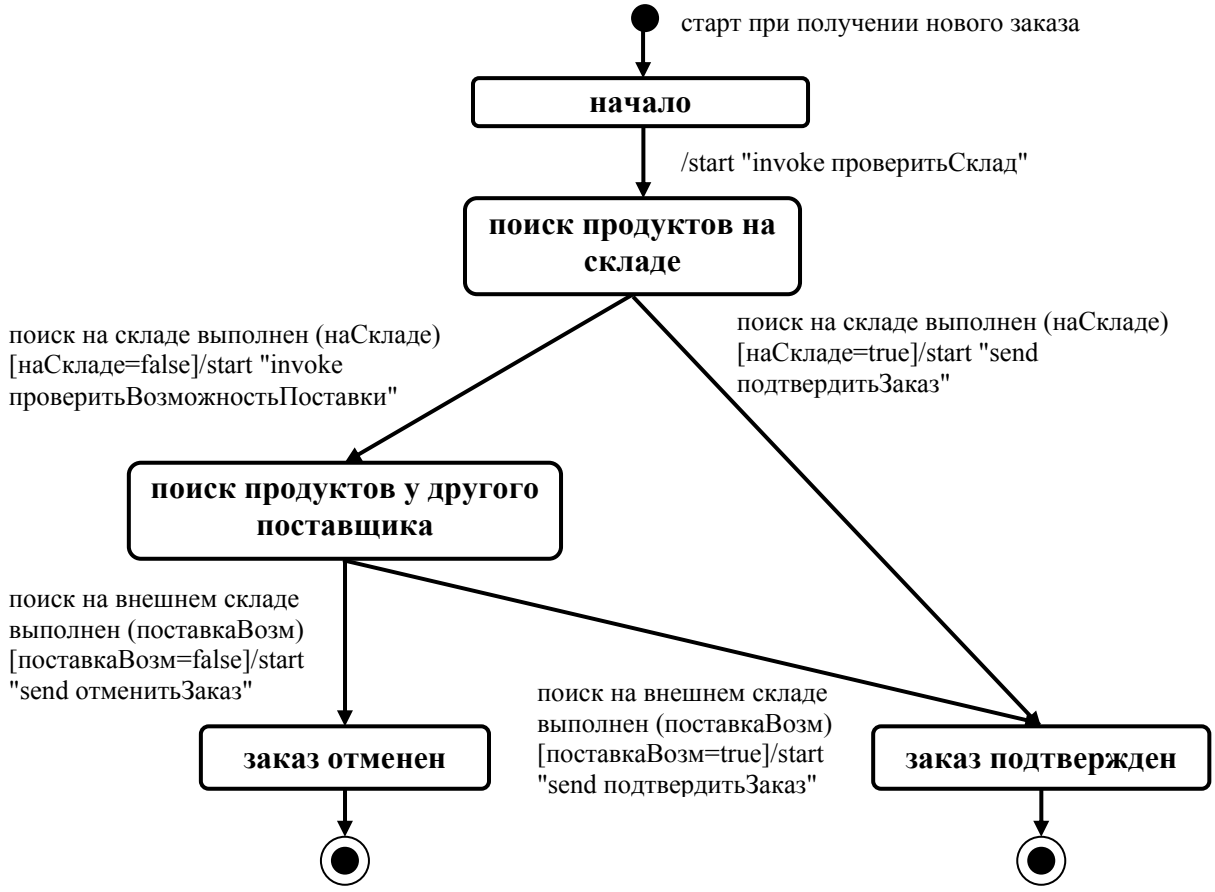


Рис. 5.7. Описание процесса с помощью диаграммы состояний.

Кроме посылаемых и получаемых сообщений, диаграммы активности обычно содержат множество деталей, связанных с особенностями конкретной модели. Туда могут входить определения URL служб, которым отправляются сообщения, способы построения сообщений на основе результатов предыдущих активностей, способы обработки исключительных ситуаций, которые могут возникать при выполнении активностей. В отличие от диаграмм активностей в описаниях протоколов, в этих диаграммах полностью специфицируются все условия и элементы данных, поскольку внутренние спецификации служб не доступны.

Существуют альтернативные способы описания оркестровой модели.

Диаграммы состояний. Этот формализм основан на расширенной автоматной схеме, что дает возможность определять активности, выполняемые при входе в состояние, при выходе из него или при нахождении в состоянии. Диаграммы состояний также описывают события, условия и действия, связанные с переходами, например, определяют переход, который нужно сделать при возникновении события, если истинно некоторое условие. В этом случае выполняется также и связанное с состоянием действие. Имеются и другие расширения, которых так много, что почти стирается грань между диаграммами состояний и диаграммами активностей (*Рис. 5.7*).

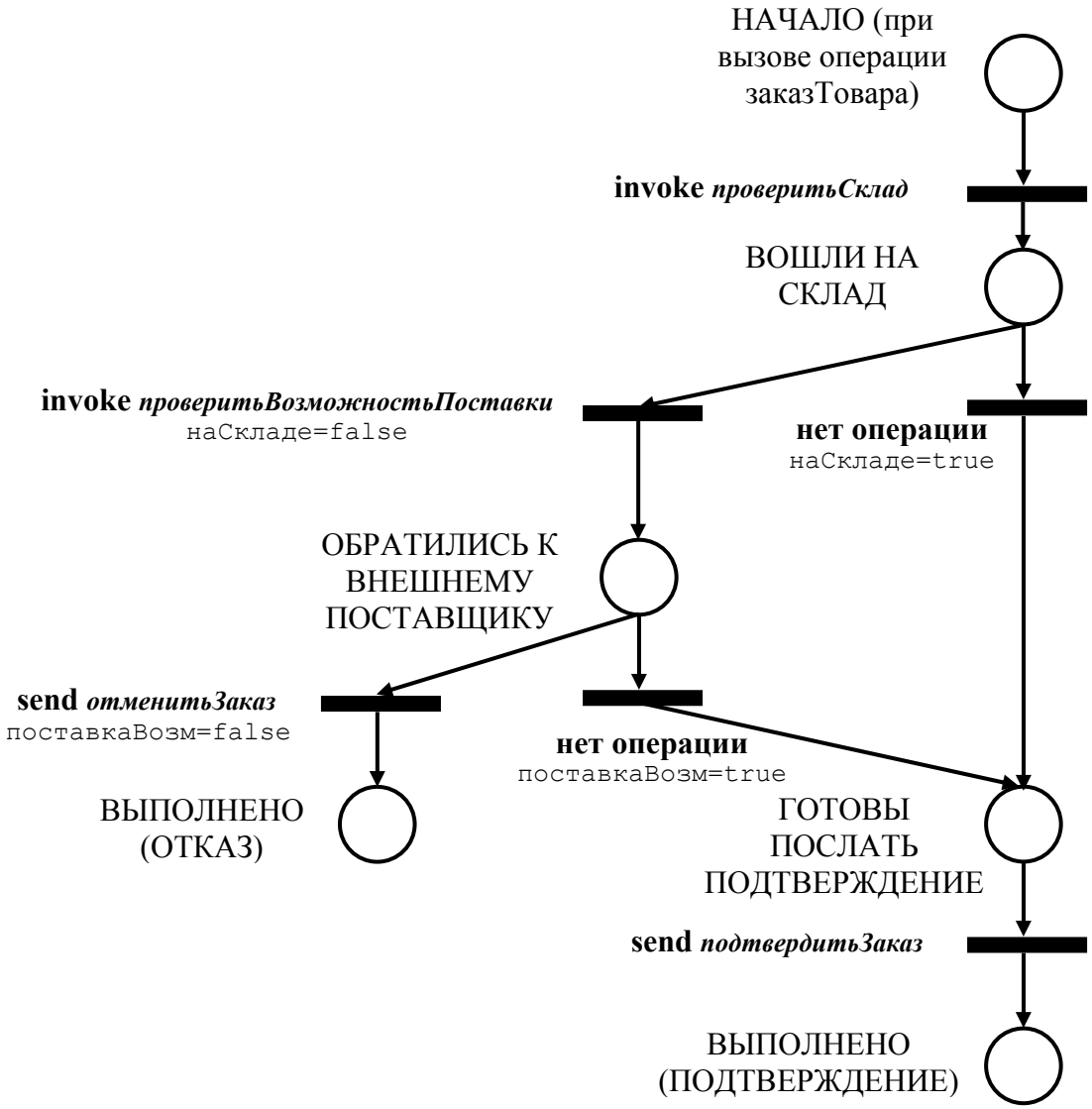


Рис. 5.8. Описание процесса с помощью сети Петри.

Сети Петри. Оркестровая модель, базирующаяся на сетях Петри, объединяет две парадигмы – диаграммы активности и определение состояний процесса с помощью диаграммы состояний. Свойства сетей

Петри четко определены и имеют понятную семантику. Для работы с сетями Петри созданы многочисленные системы автоматического анализа. С помощью этих систем пользователи могут исследовать свойства спецификаций и обнаруживать потенциально ошибочные места. На *Рис. 5.8* приведен все тот же пример, но в виде сети Петри. Некоторые переходы помечены логическими предикатами, управляющими переключением состояний. На основе сетей Петри создано несколько коммерческих моделей и исследовательских прототипов.

π-исчисление есть алгебра процессов, на основе которой созданы современные языки композиции, например, язык BPEL. π-исчисление можно рассматривать как попытку разработки формальной теории моделирования процессов, аналогичной той, которую реляционная алгебра предлагает для реляционной модели. Как и в случае сетей Петри достоинством модели является наличие точного и хорошо изученного формализма, который способен верифицировать свойства изучаемого процесса.

С точки зрения оркестровой модели π-исчисление вводит конструкции для композиции активностей в терминах последовательного, параллельного или условного выполнения. Запись A.B означает, что активность A происходит раньше активности B, A|B означает, что A и B происходят параллельно, A + B означает, что выполняется либо A, либо B (выбор недетерминирован), а запись [переменная=значение]A означает, что A выполняется, если и только если переменная имеет указанное значение. В примере на *Рис. 5.9* дано описание процесса закупки на основе облегченного синтаксиса.

```
A=receiveЗаказТовара, invokeПроверитьСклад
B=[поставкаВозм=false]sendОтменитьЗаказ+
  [поставкаВозм=true]sendПодтвердитьЗаказ
C=invokeПроверитьВозможностьПоставки.B

Закупка=A.( ([наСкладе=false]C) +
            ([наСкладе=true]sendПодтвердитьЗаказ)
          )
```

Рис. 5.9. Описание процесса с помощью π-исчисления.

Иерархии активностей (*Рис. 5.10*) представляют собой еще один подход к описанию оркестровки. В этой модели процессы описываются в виде постоянно уточняемой активности верхнего уровня с помощью дерева активностей. Конечные узлы (листья дерева) представляют собой фактически исполняемые активности, а промежуточные узлы определяют порядок следования активностей более низких уровней.

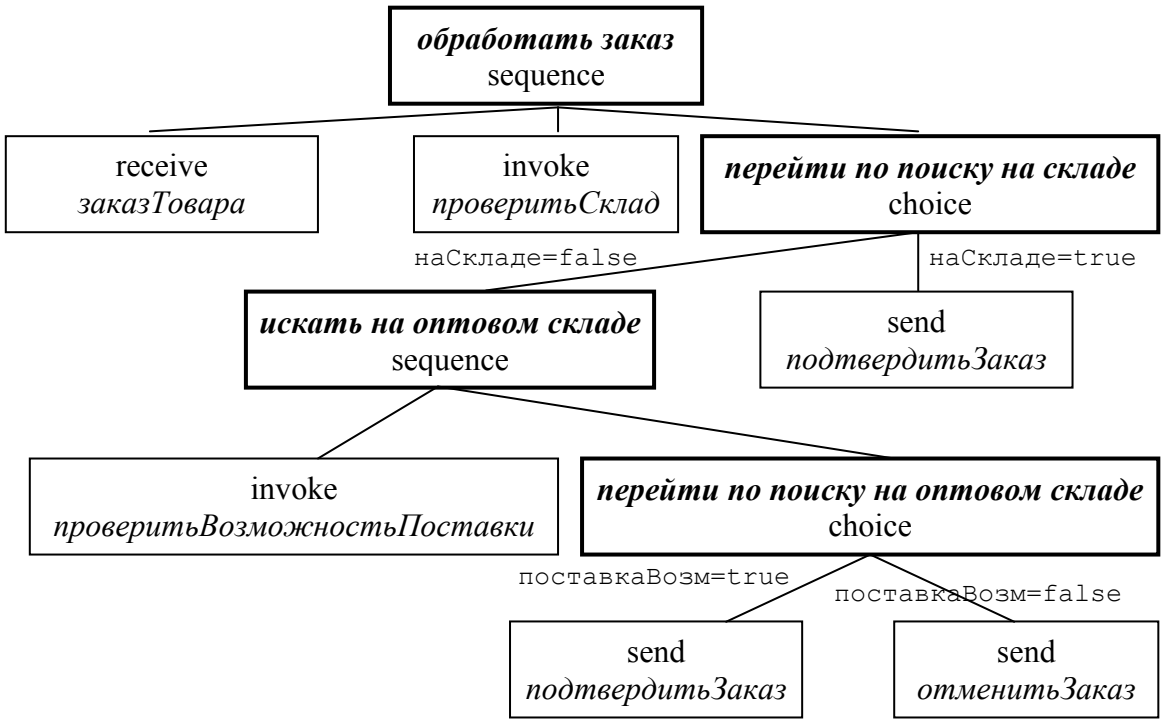


Рис. 5.10. Описание процесса с помощью иерархии активностей.

Эта формализация не очень практична и недостаточно интуитивно понятна разработчикам, как из-за необходимости осуществлять "искусственные" шаги, так и из-за необходимости думать в терминах хронологического порядка следования шагов (что эквивалентно применению при разработке иерархии и раскрытии левых поддеревьев метода "*сначала вглубь*"). У этой модели имеются и достоинства. Создаваемая структура позволяет взглянуть на оркестровку с разных уровней абстракции: более высокие уровни абстракции располагаются ближе к корню дерева, более детализированные уровни могут получаться постепенным перемещениям к листьям. Это естественным образом делает определение модульным, позволяя вести параллельную разработку процесса разным группам разработчиков.

Оркестровка на основе правил часто используется в реактивных системах, то есть в системах, которые отслеживают в одном или нескольких приложениях факты возникновения интересующих их *событий*, сигнализирующих о критических условиях. Фиксация события приводит к выполнению действия, управляющего ситуацией. Для этого правила в программных системах записываются в виде пар <событие-действие>. В некоторых моделях правило может содержать еще и условие, то есть логический предикат над параметрами события, вычисляемый при обнаружении события. Предикат определяет, нужно ли выполнять

указанное действие. В таких случаях говорят, что модель следует парадигме *событие-условие-действие*.

В качестве реактивной системы можно рассматривать композиционный мотор, который реагирует на сообщения, получаемые клиентами или другими системами (события) продвижением по оркестровой схеме (которая может включать и условия) и, соответственно, отправкой других сообщений (действия). На *Рис. 5.11* приведен все тот же пример оркестровки, записанный с помощью правил. Хотя правила записаны в некотором порядке, между ними нет отношения упорядоченности.

В отличие от иерархий активностей модели, основанные на правилах, гораздо менее структурированы и меньше отражают порядок в общем потоке действий. Они больше подходят для тех моделей оркестровки, в которых имеется не очень много ограничений на активности, и где, следовательно, небольшое число правил может определять всю схему.

```
ON receive заказТовара
IF true
THEN invoke проверитьСклад;

ON complete(проверитьСклад)
IF (наСкладе==true)
THEN send подтвердитьЗаказ;

ON complete(проверитьСклад)
IF (наСкладе==false)
THEN invoke проверитьВозможностьПоставки;

ON complete(проверитьВозможностьПоставки)
IF (поставкаВозм ==true)
THEN send подтвердитьЗаказ;

ON complete(проверитьВозможностьПоставки)
IF (поставкаВозм==false)
THEN send отменитьЗаказ;
```

Рис. 5.11. Описание процесса с помощью правил.

Правила позволяют также моделировать асинхронные события, то есть события, которые могут произойти на любой стадии процесса, что делает их пригодными для определения логики управления исключительными ситуациями, которые по своей природе асинхронны.

Недостатком модели является трудность понимания логики процессов, описанных большим числом правил.

5.3.3. Модель данных и доступа к данным

В самом общем виде данные, используемые во время композиции служб, можно разделить на данные, связанные с отдельными приложениями, и данные управляющего потока. Прикладные данные – это параметры, посылаемые или получаемые в сообщениях. Управляющие данные используются для вычисления условий перехода, а в общем случае это те данные, которые используются композиционным мотором при выполнении процесса (*переменные процесса*). В большинстве систем управляющих данных немного, их типы ограничены строковыми, целыми или вещественными, могут использоваться массивы и структуры. Значения управляющих данных обычно извлекаются из сообщений, получаемых сетевыми службами.

Прикладные данные более сложны и разнообразны. Один из подходов для работы с ними заключается в их трактовке как черных ящиков, которые можно только передавать от одной активности другой. Тем самым, вместо обмена текстовыми документами или изображениями сетевые службы обмениваются только указателями (например, URL) на места размещения данных. Другой подход пытается сделать все данные явными, вставляя определения данных в композиционную схему.

Черные ящики имеют свои преимущества. Одно из них – в том, что композиционная модель может игнорировать обмены сложными данными между активностями.

Многие системы в настоящее время поддерживают обобщенный тип XML, который в части систем может быть связан с некоторой схемой. Однако представление в виде XML не очень наглядно, кроме того, много времени тратится на просмотр документов и преобразования данных. Многим приложениям XML просто не нужен. Выражается это в присоединениях двоичных файлов к сообщениям SOAP. Такая практика приводит к тому, что XML становится лишь контейнером для хранения прикладной информации, снова превращающейся в черный ящик.

Передача данных. Для передачи данных между активностями, существуют два подхода: журнальный подход и подход явного потока данных. Журнальный подход аналогичен традиционным языкам программирования: все данные, используемые в сетевой службе, должны явно именоваться и перечисляться. Журнал – это набор всех переменных, в который активности (обращения к операциям) заносят свои результаты, откуда они берут свои входные параметры. Модификации переменных выполняются при получении сообщения "атомарно", прежние значения переменных стираются. Активности могут иметь разный уровень доступа к

переменным (чтение/запись или только чтение). Каждый запуск активности заводит свой отдельный журнал, как каждый запуск программы заводит свой набор переменных.

Подход явного потока данных требует, чтобы поток данных (в дополнение к потоку управления) был явным элементом композиции. На диаграммах активностей явно прорисовываются соединительные линии, с помощью которых разработчик указывает, что входные данные активности должны браться из результатов другой, ранее выполненной активности.

Выбор того иного подхода зависит от многих факторов. Потоки данных гибче и богаче журналов, но работать с ними сложнее: они создают неявные управляющие зависимости, поскольку активности, являющиеся источниками данных должны завершаться до начала работы активностей, эти данные получающих. Более того, в случаях, когда некоторые входные данные могут поставляться несколькими различными потоками, этот подход может приводить к возникновению соревнований между активностями. Журнальный подход более естественен, так описывают передачи данных языки программирования.

5.3.4. Модель выбора службы

Композиционная схема описывает посылаемые и получаемые сообщения, а также порядок, в котором проводятся обмены. Для выполнения композиционной логики мотор должен в дополнение к схеме знать еще и то, какая конкретно служба (например, URL службы) является получателем посылаемого сообщения. В композиционных схемах эта информация указывается абстрактно (вместо номера порта на схеме указывается тип порта, на который отправляются сообщения, или роль, которую играет получатель). Однако во время выполнения перед отправкой сообщения все типы портов должны заменяться точными номерами портов, чтобы мотор знал, куда направить сообщение. Другими словами композитная служба должна выбирать сетевую службу. Существуют четыре способа привязки:

- статическое связывание,
- динамическое связывание по ссылке,
- динамическое связывание с поиском,
- динамический выбор операции.

Самый простой способ выбора службы – вставка указателя на нее (URL) в спецификацию композитной службы, что эквивалентно *статическому связыванию*. Особенно этот способ полезен при построении прототипов и тестировании композиций. Очевидный недостаток –

трудность отслеживания изменений URI службы (определение процесса приходится модифицировать и переустанавливать), а также то, что все отдельные запуски композитной службы всегда обращаются к одной и той же службе, то есть никакого "выбора" служб не происходит.

Для устранения этих недостатков используют подход *ссылок*, при котором определения службы берутся из переменных процесса. Этот метод *динамического связывания по ссылке* тоже прост. Обычно присваивание указателя переменной происходит в результате выполнения предыдущей операции, его можно извлекать из указателя клиента, обратившегося к композитной службе, он может быть явно задан при установке службы на машине. Если указатель должен быть динамически извлечен из справочника, этому сопоставляется отдельная активность. Такой активностью может быть операция некоторого реестра (чей указатель определяется статически или сам, в свою очередь, определяется по ссылке), которая определяет указатель службы и записывает его в некоторую переменную, используемую в следующей активности.

При *динамическом связывании с поиском* системные композиционные программы позволяют для каждой активности описывать запрос к некоторому справочнику. Результат обработки запроса используется для определения службы, которую надлежит вызывать. Например, язык WSFL (один из предшественников BPEL) имеет возможности описывать подобные обращения в виде стандартных обращений к реестрам UDDI, используя для этого форматы прикладного программного интерфейса UDDI. Это может приводить к множественному выбору служб, а, значит, и к необходимости уметь формулировать критерии уточнения выбора одной службы из целого списка. В частности, можно организовывать простой случайный выбор.

Как и в спецификации CORBA, и в других традиционных системных платформах, композиционные модели могут проводить динамическое связывание не только на уровне целых служб, но и на уровне отдельных операций. Такой подход называется *динамическим выбором операции*. Например, при организации дальних поездок активность "заказ билетов" может обращаться к разным операциям (и к разным службам), в зависимости от того, хочет ли заказчик ехать автобусом, поездом, лететь на самолете или плыть на корабле. Выбор можно моделировать на оркестровом уровне, вводя условия активации одной из нескольких активностей, определяющие выбор на основе предпочтений заказчика. Этот подход усложняет оркестровку, которой, при росте вариантов выбора становится трудно управлять. Более гибкое решение связано с

определением *абстрактных* активностей, которые явно не специфицируют никаких операций. Вместо этого операции выбираются во время выполнения, вместе со службой. Динамические операции могут быть полезны в тех случаях, когда набор параметров операции меняется, в зависимости от выбранной службы.

5.3.5. Транзакции

Транзакционное поведение композитных служб определяется внедрением в оркестровую схему *атомарных областей*. На графических диаграммах такие области окружают наборы активностей, обладающие свойством *все-или-никто*. Атомарность достигается выполнением с вызванными службами протоколов двухфазного подтверждения, возможно основанных на спецификации WS-Transaction. Это поведение полностью реализуется в системном слое, не требуя от разработчика никакого программирования.

Иногда требуется менее строгая транзакционная семантика, не требующая досконального выполнения транзакционных свойств, что связано с длительностью блокировки ресурсов при выполнении композитных служб. Алгоритм 2PC для подтверждения атомарности *наборов операций* атомарных областей применять нельзя. Решение этой проблемы находится при применении *компенсаций*, когда результаты подтвержденных операций отменяются выполнением других операций. Применение такого подхода означает, что при возникновении ошибки для осуществления частичного отката операций атомарной области системный слой сетевых служб должен инициировать и выполнить протокол компенсации подтвержденных активностей. Компенсация может управляться мотором, который в совокупности с системными программами сетевых служб, реализующими спецификации WS-Transaction (в частности, протокол бизнес активностей), выполняет компенсационный протокол и информирует службы о необходимости компенсаций.

Многие композиционные модели позволяют разработчикам явно определять компенсационную логику в форме оркестровой схемы, описывающей способы компенсации атомарных областей. В этом случае при возникновении ошибок композиционный мотор прерывает активные операции и выполняет пользовательскую компенсационную логику.

Ответственность за реализацию компенсации все чаще возлагается на разработчика сетевой службы, то есть на стороны компонентов, а не на сторону композиции. Если компонент имеет транзакционную и компенсационную логику, то разработчик композиции в большинстве

случаев от реализации компенсационной логики освобождается. Разработчики композиции, если им нужно компенсировать операцию, могут просто обращаться к компенсирующей операции. Если компонент, кроме того, поддерживает стандартные компенсационные механизмы, легко может быть получен сценарий, в котором транзакции (с точки зрения композиции служб) поддерживаются.

5.3.6. Управление исключениями

В контексте композиции сетевых служб термин *исключение* относится к отклонениям от ожидаемого или желаемого выполнения композиции. Исключения обычно вызываются ошибками в системе или в вызываемых приложениях, либо могут быть ситуациями, хотя и предусмотренными семантикой сетевой службы, но нечастыми, например, когда заказчик отменяет ранее выданный заказ. Одним из способов управления исключениями являются транзакции, хотя транзакционная отмена приводит к потере части ранее выполненной работы.

Подходы, основанные на потоках. Такие методы применяются в отсутствие специальных конструкций для управления исключениями. Они аналогичны методам, применяемым в языках третьего поколения, которые не имеют поддержки исключений: в конце каждой операции результат проверяется на наличие ошибок, и, если обнаруживается ошибка, предпринимаются соответствующие действия. Когда вызванная операция не возвращает никакого результата, для каждой активности вводят таймауты, при срабатывании которых мотор останавливает активность и выполнение продолжается.

Исключение может явно возбуждаться композиционным мотором или сетевой службой. В таких случаях исключение подхватывается активностью, связанной с операциями, инициируемыми при получении сообщения.

Подходы "попытка-перехват-возбуждение". Эта методика похожа на то, как делается в программах на языках Си++ и Java операторами ***try***, ***catch*** и ***throw***, но адаптирована для сетевых служб. Для исключения вводится логическое условие, связанное с композиционными данными, обычно с результатом сообщения о выполнении операции. Если условие выполнено, выполняется и часть программы, управляющая исключением. После этого процесс может повторить активность, повторить обработку исключения или просто остановиться. Иерархия позволяет определять обработчики исключений на разных уровнях абстракции. Первыми вызываются обработчики исключений, определенные внутри группы активностей, в которой возникло исключение. После окончания их работы

у них есть возможность снова возбудить исключение, чтобы оно могло быть обработано объемлющей активностью данной иерархии. Если внутри активности нет обработчика исключений, подходящий обработчик ищется переходами вверх по иерархии.

Подход "попытка-перехват-возбуждение" имеет несколько преимуществ. Во-первых, он отделяет обычную логику от логики исключений. Это помогает структурировать композицию, облегчить ее проектирование и сопровождение. Во-вторых, если композиция структурирована в соответствии с деревом декомпозиции, аналогично организуют и исключения. В-третьих, этот подход позволяет описать *стратегию продолжения*, то есть указать, что произойдет с вызовом композиции и с "исключительной" активностью, после обработки исключения.

Подходы, основанные на правилах. Эти подходы описывают логику управления исключениями на основе правил ***event-condition-action***, с помощью которых исключительное *событие* (например, отказ заказчика), должно быть облечено в форму сообщений, посылаемых клиентом композитной службе, или в форму таймаутов. *Условие* – это логическое выражение над сообщением, которое проверяет, действительно ли *событие* относится к исключительной ситуации, которую надо обработать (например, оно может проверять, разрешено ли данному заказчику иметь отрицательный баланс). Затем на исключение реагирует *действие*, вызывая операцию или прерывая транзакцию. Правила обычно определяются в некотором текстовом виде, дополняя графическую нотацию, определяемую в обычной оркестровке.

Подходы, основанные на правилах, четко разделяют нормальное и исключительное поведение процесса. Эти подходы хороши, если число правил невелико, иначе трудно анализировать и понимать совместное поведение набора правил и взаимодействие внутри набора между правилами и потоками.

5.4. Координация композитных служб

5.4.1. Зависимости между координацией и композицией

Основные отношения между координационными протоколами и композицией связаны с тем фактом, что определение протокола накладывает ограничения на композиционную схему сетевой службы, реализующей логику протокола. Если сетевая служба играет роль в некотором протоколе, а реализация сделана на основе композиционных методов, эта схема должна включать активности, которые получают и отсылают сообщения, предписанные протоколом.

Чтобы создать сетевую службу, которая сможет играть роль поставщика, сначала надо создать ролевой фрагмент протокола. Этот фрагмент должен включать все обмены сообщениями, затрагивающие данную роль поставщика, то есть выделенный фрагмент протокола. Следующий шаг состоит в переходе от ролевой части протокола к определению процесса обмена сообщениями, предписанному ролевой частью, с целью определения процесса, включающего все активности, отправляющие и получающие сообщения на основе протокола.

Созданный фундамент послужит отправной точкой для разработчиков сетевой службы, которые добавят к нему необходимую бизнес логику и получат композиционную схему, которая в протоколе закупки сможет играть роль поставщика. Чтобы такой фундамент построить, каждой вызываемой операции, отмеченной в роли, надо поставить в соответствие активности процесса.

Созданный абстрактный процесс есть полностью эквивалентное представление ролевого фрагмента, но описанное несколько с другой точки зрения. Здесь определяется видимое поведение сетевой службы, за что эти процессы и называются открытыми. Выполняться абстрактный процесс не может, его определение может только передаваться контроллеру разговоров, который проверяет, что обмен сообщениями происходит в соответствии с протоколом. Композиционный мотор не сможет с ним работать потому, что ему нужно знать, как строить сообщения и как вычислять условия ветвления.

Преимущества введения абстрактных процессов в том, что они облегчают понимание того, как протоколы ограничивают композицию, и как определить композиционную схему, реализующую протокол. Расширение абстрактного протокола необходимыми деталями легко приведет разработчиков к композиционной схеме. Обычно приходится добавлять дополнительные активности, вызывающие другие службы, и другие детали, отсутствующие подробности, например, условия ветвления, присваивания данных и правила передачи данных. На практике сетевые службы должны поддерживать несколько протоколов одновременно и вести сразу несколько разговоров.

Языки, ориентированные на процессы, и предложения по стандартам по-разному подходят к решению проблем композиции, протоколов и их взаимоотношениям. Некоторые современные языки (BPEL, ebXML) могут описывать и внешнее поведение (абстрактные процессы) и внутреннюю реализацию (выполняемые процессы).

5.4.2. Контроллеры разговоров и композиционные моторы

Разработка архитектуры композитной службы на основе композиционного мотора сталкивается с проблемами маршрутизации (*Рис. 5.12*). Системная поддержка сетевых служб, включающая контроллер разговоров и композиционный мотор, работает так, что контроллер проверяет соответствие протоколу и направляет сообщения в мотор. Мотор представляет собой внутренний объект, реализующий разговор. Он выполняет множество композиционных запусков, к которым поступают все сообщения, относящиеся к этим запускам, поэтому должен уточнять, к какому конкретно запуску надо направить каждое конкретное сообщение.

Способ, которым это делается, зависит от деталей работы контроллера и мотора, а также от выбранной композиционной модели. Если контроллер разговоров и маршрутизатор SOAP при передаче сообщений композиционному мотору оставляют их информационные заголовки, для определения места назначения используется координационный контекст. Если контроллер доставляет только основное содержание сообщений, мотор должен искать другие способы соотнесения сообщений адресатам. Одно из решений состоит в явном включении в композиционную схему корреляционной информации, на основе параметров сообщений определяя логику, по которой сообщения могут быть ассоциированы с композиционными запусками.

По мере становления новых технологий вероятнее всего контроллеры разговоров и композиционные моторы будут интегрироваться друг с другом или будут взаимодействовать средствами стандартных интерфейсов, что поможет освободить разработчиков композиционных служб от решения проблем маршрутизации.

В настоящее время для описания сетевых служб широко применяется язык выполнения бизнес процессов для сетевых служб *BPEL* (*Business Process Execution Language for Web Services*, *BPEL4WS*). Этот язык может поддерживать спецификации и композиционных схем, и координационных протоколов. Композиционные схемы BPEL – это полноценные спецификации *выполняемых* процессов, определяющие логику реализации (композитных) служб. В центре координационных протоколов BPEL находятся службы, они специфицируют абстрактные процессы и определяют последовательность обменов сообщениями, поддерживаемых службой (в терминах сообщений, которые служба посылает и получает). Язык BPEL можно использовать для описания внутреннего и внешнего поведения службы. Спецификации BPEL основаны на документах XML, определяющих, роли участников

взаимодействия, типы портов, оркестровку и корреляционную информацию.

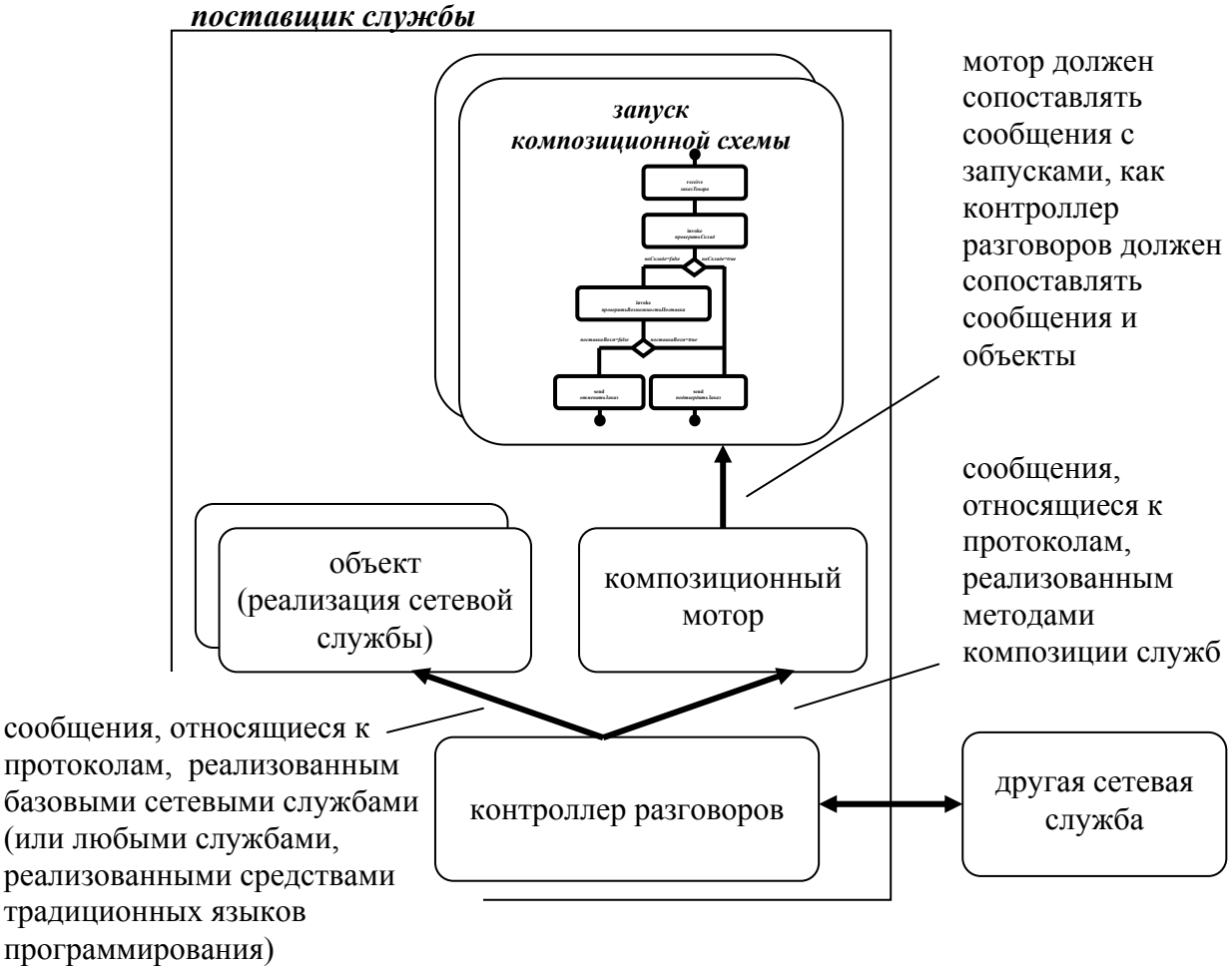


Рис. 5.12. Композиционный мотор сталкивается с проблемой маршрутизации разговоров, сходной с проблемами контроллера разговоров.

Компонентная модель языка BPEL имеет тонкую структуру, состоящую из активностей, которые могут базовыми или структурными, причем базовые активности соответствуют вызовам операций WSDL. Оркестровая модель BPEL сочетает в себе диаграммы и иерархии активностей. Язык BPEL имеет средства поддержки маршрутизации, полезные в тех случаях, когда системная инфраструктура не обеспечивает прозрачной маршрутизации. Средствами языка разработчики могут определять, как на основе данных из сообщений можно соотносить сообщения с конкретными запусками композиционных моторов.

В мае 2003 года предложения по BPEL, представленные компаниями IBM, BEA и Microsoft, были ими пересмотрены и получили поддержку многих поставщиков прикладных систем (SAP, Siebel systems). В настоящее время продолжается работа над рабочим проектом версии 2.0 языка BPEL.

Основная литература

1. Л. Е. Карпов. "Архитектура распределенных систем программного обеспечения", М., МАКС Пресс, 2007. *Шифр в библиотеке МГУ: 5ВГ66, К-265*.
2. Andrew S. Tanenbaum, Maarten van Steen. "Distributed Systems. Principles and paradigms". Prentice Hall, Inc., 2002 (Э. Таненбаум, М. ван Стеен. "Распределенные системы. Принципы и парадигмы". СПб.: Питер, 2003)
3. Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. "Web Services. Concepts, Architectures and Applications". Springer-Verlag, 2004.
4. *http://www-128.ibm.com/developerworks/webservices/standards/*

Дополнительная литература

5. John Barkley. "Comparing Remote Procedure Calls", Oct 1993 (*http://hissa.nist.gov/rbac/5277/titlerpc.html*).
6. Philip A. Bernstein. "Middleware - A model for Distributed System Services". Communications of the ACM, v. 39, No 2, February, 1996. (Ф. Бернштейн. "Middleware: модель сервисов распределенной системы". Открытые системы, Системы управления базами данных, № 2, 1997, *http://www.osp.ru/dbms/1997/02/41.htm*).
7. Robert Orfali, Dan Harkey, Jeri Edwards. "Instant CORBA". Wiley Computer Publishing, John Wiley & Sons, Inc., 1997 (Р. Орфали, Д. Харки, Д. Эдвардс, "Основы CORBA", М., МАЛИП, 1999).
8. Natanya Pitts. "XML In Record Time™", Sybex Inc., 1999 (Натания Питс. "XML за рекордное время", М.: "Мир", 2000).
9. М. Мамаев. "Телекоммуникационные технологии (Сети TCP/IP)". Владивостокский госуниверситет экономики и сервиса. Владивосток, 2001. Доступ в Интернете по адресу *http://athena.vvsu.ru/net/book/index.html*.
10. А. А. Цимбал, М. Л. Аншина. "Технологии создания распределенных систем. Для профессионалов". СПб.: Питер, 2003.
11. Eric Newcomer. "Understanding Web Services: XML, WSDL, SOAP and UDDI", Addison-Wesley, 2002 (Эрик Ньюкомер. "Веб-сервисы. Для профессионалов", СПб.: Питер, 2003).
12. W. Richard Stevens. "UNIX Network Programming. Networking APIs", Prentice Hall PTR, 2nd edition, 1998 (У. Стивенс "Разработка сетевых приложений", СПб.: Питер, 2004).

Вспомогательная литература

13. *http://www.corba.org*
14. *http://www-128.ibm.com/developerworks/webservices/library/specification/ws-tx/*
15. *http://www-128.ibm.com/developerworks/library/specification/ws-bpel/*
16. *http://www.sei.cmu.edu/str/descriptions*
17. Л. А. Калиниченко, М. Р. Когаловский, "Стандарты OMG: Язык определения интерфейсов IDL в архитектуре CORBA", Системы Управления Базами Данных, № 2, стр. 115-129, 1996 (*http://www.tts.tomsk.su/personal/~sas/DBMS/96_2/source/115.htm*).
18. "OSF DCE 1.2.2 Application Development Guide – Core Components", The Open Group, 1997.
19. B. Viveney. "DCE and Object Programming". In W. Rosenberry (ed.) "DCE Today", pp. 251 – 264. Upper Saddle River, NJ, Prentice Hall Inc., 1998.
20. А. Касаткин. "Средства middleware и их классификация". PCWeek, № 19 (193), 1999.
21. А. А. Цимбал. "Технология CORBA для профессионалов". СПб.: Питер, 2001.
22. Oracle Message Broker Administration Guide. Release 2.0.1.0. Part Number A65435-01 (for SPARC Solaris & Windows NT).
23. Jon Siegel. "Quick CORBA™ 3". Wiley Computer Publishing, John Wiley & Sons, Inc., 2001 (Джон Сигел, "CORBA 3", М., МАЛИП, 2002).
24. И. Ш. Хабибуллин. "Создание распределенных приложений на Java 2". СПб.: БХВ-Петербург, 2002.