Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 12. Компонентные технологии и разработка распределенного ПО

Аннотация

Рассматриваются основные понятия компонентных технологий разработки ПО и понятие компонента. Рассказывается об общих принципах разработки распределенного ПО и об организации взаимодействия его компонентов в рамках удаленного вызова процедур и транзакций.

Ключевые слова

Программный компонент, интерфейс, программный контракт, компонентная модель, компонентная среда, базовые службы компонентной среды, распределенное ПО, прозрачность, открытость, масштабируемость, безопасность, синхронное и асинхронное взаимодействие, удаленный вызов процедур, транзакция.

Текст лекции

Основные понятия компонентных технологий

Понятие *программного компонента* (*software component*) является одним из ключевых в современной инженерии ПО. Этим термином обозначают несколько различных вещей, часто не уточняя подразумеваемого в каждом конкретном случае смысла.

- Если речь идет об архитектуре ПО (или ведет ее архитектор ПО), под компонентом имеется в виду то же, что часто называется *программным модулем*. Это достаточно произвольный и абстрактный элемент структуры системы, определенным образом выделенный среди окружения, решающий некоторые подзадачи в рамках общих задач системы и взаимодействующий с окружением через определенный интерфейс. В этом курсе для этого понятия употребляется термин *архитектурный компонент* или *компонент архитектуры*.
- На диаграммах компонентов в языке UML часто изображаются компоненты, являющиеся единицами сборки и конфигурационного управления, — файлы с кодом на каком-то языке, бинарные файлы, какие-либо документы, входящие в состав системы. Иногда там же появляются компоненты, представляющие собой единицы развертывания системы, — это компоненты уже в третьем, следующем смысле.
- Компоненты развертывания являются блоками, из которых строится компонентное программное обеспечение. Эти же компоненты имеются в виду, когда говорят о компонентных технологиях, компонентной или компонентно-ориентированной (component based) разработке ПО, компонентах JavaBeans, EJB, CORBA, ActiveX, VBA, COM, DCOM, .Net, Web-службы (web services), а также о компонентном подходе, упоминаемом в названии данного курса. Согласно [1], такой ***компонент*** представляет собой структурную единицу программной системы, обладающую четко определенным интерфейсом, который полностью описывает ее зависимости от окружения. Такой компонент может быть независимо поставлен или не поставлен, добавлен в состав некоторой системы или удален из нее, в том числе, может включаться в состав систем других поставщиков.

Различие между этими понятиями, хотя и достаточно тонкое, все же может привести к серьезному непониманию текста или собеседника в некоторых ситуациях.

В определении архитектурного компонента или модуля делается акцент на выделение структурных элементов системы в целом и декомпозицию решаемых ею задач на более мелкие подзадачи. Представление такого компонента в виде единиц хранения информации (файлов, баз

данных и пр.) не имеет никакого значения. Его интерфейс хотя и определен, может быть уточнен и расширен в процессе разработки.

Понятие компонента сборки и конфигурационного управления связано со структурными элементами системы, с которыми приходится иметь дело инструментам, осуществляющим сборку и конфигурационное управление, а также людям, работающим с этими инструментами. Для этих видов компонентов интерфейсы взаимодействия с другими такими же элементами системы могут быть не определены.

В данной лекции и в большинстве следующих мы будем иметь дело с компонентами в третьем смысле. Это понятие имеет несколько аспектов.

- Компонент в этом смысле — *выделенная структурная единица с четко определенным интерфейсом.* Он имеет более строгие требования к четкости определения интерфейса, чем архитектурный компонент. Абсолютно все его зависимости от окружения должны быть описаны в рамках этого интерфейса. Один компонент может также иметь несколько интерфейсов, играя несколько разных ролей в системе.
  При описании интерфейса компонента важна не только сигнатура операций, которые можно выполнять с его помощью. Становится важным и то, какие другие компоненты он может задействовать при работе, а также каким ограничениям должны удовлетворять входные данные операций и какие свойства выполняются для результатов их работы.

  Эти ограничения являются так называемым ***интерфейсным контрактом*** или ***программным контрактом*** компонента. Интерфейс компонента включает набор операций, которые можно вызвать у любого компонента, реализующего данный интерфейс, и набор операций, которые этот компонент может вызвать в ответ у других компонентов. Интерфейсный контракт для каждой операции самого компонента (или используемой им) определяет ***предусловие*** и ***постусловие*** ее вызова. Предусловие операции должно быть выполнено при ее вызове, иначе корректность результатов не гарантируется. Если эта операция вызывается у компонента, то обязанность позаботиться о выполнении предусловия лежит на клиенте, вызывающем операцию. Если же эта операция вызывается компонентом у другого компонента, он сам обязуется выполнить это предусловие. С постусловием все наоборот — постусловие вызванной у компонента операции должно быть выполнено после ее вызова, и это — обязанность компонента. Постусловие операции определяет, какие ее результаты считаются корректными. В отношении вызываемых компонентом операций выполнение их постусловий должно гарантироваться теми компонентами, у которых они вызываются, а вызывающий компонент может на них опираться в своей работе.

  Пример. Рассмотрим сайт Интернет-магазина. В рамках этого приложения может работать компонент, в чьи обязанности входит вывод списка товаров заданной категории. Одна из его операций принимает на вход название категории, а выдает HTML-страничку в заданном формате, содержащую список всех имеющихся на складе товаров этой категории. Предусловие может состоять в том, что заданная строка действительно является названием категории. Постусловие требует, чтобы результат операции был правильно построенной HTML-страницей, чтобы ее основное содержимое было таблицей со списком товаров именно указанной категории, название каждого из которых представляло бы собой ссылку, по которой можно попасть на его описание, а в остальном — чтобы эта страница была построена в соответствии с принятым проектом сайта.
  Более аккуратно построенный компонент не требовал бы ничего в качестве предусловия (т.е. оно было бы выполнено при любом значении параметра), а в случае некорректного названия категории в качестве результата выдавал бы HTML-страницу с сообщением о неправильном названии категории товаров.
  При реализации интерфейса предусловия операций могут ослабляться, а постусловия — только усиливаться. Это значит, что, реализуя данную операцию, некоторый компонент может реализовать ее для более широкого множества входных данных, чем это требуется

предусловием, а также может выполнить в результате более строгие ограничения, чем это требуется постусловием. Однако внешним компонентам нельзя опираться на это, пока они работают с исходным интерфейсом, — реализация может поменяться. Точно так же, если интерфейс компонента требует наличия в системе других компонентов с определенным набором операций, это не означает, что данная реализация этого интерфейса действительно вызывает эти операции.

- Компонент в этом смысле — *единица развертывания.* Он может быть присоединен к остальной системе, когда она уже некоторое время работает, и должен после этого выполнять все свои функции, если в исходной системе уже были все компоненты, от которых он зависит. Он может быть удален из нее. Естественно, после этого могут перестать работать те компоненты, которые зависят от него. Он может быть встроен в программные продукты третьих партий и распространяться вместе с ними. В то же время никакая его часть не обладает этими свойствами.
  В идеале такой компонент может быть добавлен или полностью замещен другой реализацией тех же интерфейсов прямо в ходе работы системы, без ее остановки. Хотя и не все разновидности компонентов обладают этим свойством, его наличие признается крайне желательным.
  Все это означает, что такой компонент должен быть работоспособен в любой среде, где имеются необходимые для его работы другие компоненты. Это требует наличия определенной инфраструктуры, которая позволяет компонентам находить друг друга и взаимодействовать по определенным правилам.
- Набор правил определения интерфейсов компонентов и их реализаций, а также правил, по которым компоненты работают в системе и взаимодействуют друг с другом, принято объединять под именем ***компонентной модели (component model)*** [2]. В компонентную модель входят правила, регламентирующие ***жизненный цикл компонента***, т.е. то, через какие состояния он проходит при своем существовании в рамках некоторой системы (незагружен, загружен и пассивен, активен, находится в кэше и пр.) и как выполняются переходы между этими состояниями.
  Существуют несколько компонентных моделей. Правильно взаимодействовать друг с другом могут только компоненты, построенные в рамках одной модели, поскольку компонентная модель определяет «язык», на котором компоненты могут общаться друг с другом.

  Помимо компонентной модели, для работы компонентов необходим некоторый набор *базовых служб (basic services)*. Например, компоненты должны уметь находить друг друга в среде, которая, возможно, распределена на несколько машин. Компоненты должны уметь передавать друг другу данные, опять же, может быть, при помощи сетевых взаимодействий, но реализации отдельных компонентов сами по себе не должны зависеть от вида используемой связи и от расположения их партнеров по взаимодействию. Набор таких базовых, необходимых для функционирования большинства компонентов служб, вместе с поддерживаемой с их помощью компонентной моделью называется ***компонентной средой*** (или ***компонентным каркасом***, ***component framework***). Примеры известных компонентных сред — различные реализации J2EE, .NET, CORBA. Сами по себе J2EE, .NET и CORBA являются спецификациями компонентных моделей и набора базовых служб, которые должны поддерживаться их реализациями.
  Компоненты, которые работают в компонентных средах, по-разному реализующих одну и ту же компонентную модель и одни и те же спецификации базовых служб, должны быть в состоянии свободно взаимодействовать. На практике этого, к сожалению, не всегда удается достичь, но любое препятствие к такому взаимодействию рассматривается как серьезная, подлежащая скорейшему разрешению проблема.
  Соотношение между компонентами, их интерфейсами, компонентной моделью и компонентной средой можно изобразить так, как это сделано на Рис. 65.

- Компоненты отличаются от классов объектно-ориентированных языков.
  - Класс определяет не только набор реализуемых интерфейсов, но и саму их реализацию, поскольку он содержит код определяемых операций. Контракт компонента, чаще всего, не фиксирует реализацию его интерфейсов.
  - Класс написан на определенном языке программирования. Компонент же не привязан к определенному языку, если, конечно, его компонентная модель этого не требует, — компонентная модель является для компонентов тем же, чем для классов является язык программирования.
  - Обычно компонент является более крупной структурной единицей, чем класс. Реализация компонента часто состоит из нескольких тесно связанных друг с другом классов.

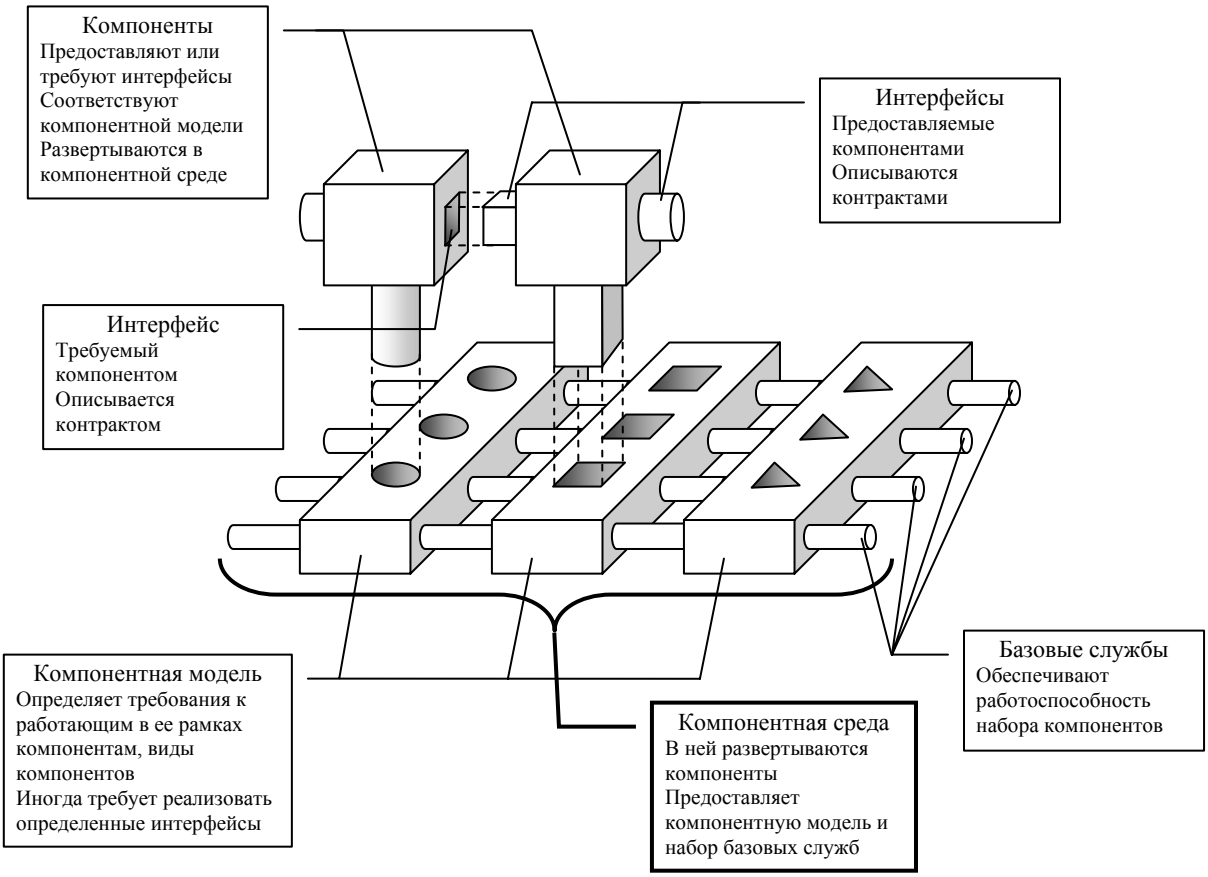

Рисунок 65. Основные элементы компонентного программного обеспечения.

- Понятие компонента является более узким, чем понятие *программного модуля*. Основное содержание понятия модуля — наличие четко описанного интерфейса между ним и окружением. Понятие компонента добавляет атомарность развертывания, а также возможность поставки или удаления компонента отдельно от всей остальной системы.
- Возможность включения и исключения компонента из системы делает его отдельным элементом продаваемого ПО. Компоненты, хотя и не являются законченными продуктами, могут разрабатываться и продаваться отдельно, если они следуют правилам определенной компонентной модели и реализуют достаточно важные для покупателей функции, которые те хотели бы иметь в рамках своей программной системы.
  Надо отметить, что, хотя рынок ПО существует достаточно давно, а компонентные технологии разрабатываются уже около 20 лет, рынок компонентов развивается довольно медленно. Поставщиками компонентов становятся лишь отдельные компании, тесно связанные с разработчиками конкретных компонентных сред, а не широкое сообщество индивидуальных и корпоративных разработчиков, как это представляли себе создатели

компонентных технологий при их зарождении.
По-видимому, один из основных факторов, мешающих развитию этого рынка, — это «гонка технологий» между поставщиками основных компонентных сред. Ее следствием является отсутствие стабильности в их развитии. Новые версии появляются слишком часто, и достаточно часто при выходе новых версий изменяются элементы компонентной модели. Так что при разработке компонентов для следующей версии приходится следовать уже несколько другим правилам, а старые компоненты с трудом могут быть переиспользованы.

Общие принципы построения распределенных систем

В дальнейшем мы будем рассматривать компонентные технологии в связи с разработкой распределенных программных систем. Прежде чем углубляться в их изучение, полезно разобраться в общих принципах построения таких систем, без привязки к компонентному подходу. Тогда многие из решений, применяемых в рамках таких технологий, становятся гораздо более понятными.

Построение распределенных систем высокого качества является одной из наиболее сложных задач разработки ПО. Технологии типа J2EE и .NET создаются как раз для того, чтобы сделать разработку широко встречающихся видов распределенных систем — так называемых бизнес-приложений, поддерживающих решение бизнес-задач некоторой организации, — достаточно простой и доступной практически любому программисту. Основная задача, которую пытаются решить с помощью распределенных систем — обеспечение максимально простого доступа к возможно большему количеству ресурсов как можно большему числу пользователей. Наиболее важными свойствами такой системы являются *прозрачность, открытость, масштабируемость* и *безопасность*.

- ***Прозрачность (transparency)***.
  Прозрачностью называется способность системы скрыть от пользователя физическое распределение ресурсов, а также аспекты их перераспределения и перемещения между различными машинами в ходе работы, репликацию (т.е. дублирование) ресурсов, трудности, возникающие при одновременной работе нескольких пользователей с одним ресурсом, ошибки при доступе к ресурсам и в работе самих ресурсов.
  Технология разработки распределенного ПО тоже может обладать прозрачностью настолько, насколько она позволяет разработчику забыть о том, что создаваемая система распределена, и насколько легко в ходе разработки можно отделить аспекты построения системы, связанные с ее распределенностью, от решения задач предметной области или бизнеса, в рамках которых системе предстоит работать.
  Степень прозрачности может быть различной, поскольку скрывать все эффекты, возникающие при работе распределенной системы, неразумно. Кроме того, прозрачность системы и ее производительность обычно находятся в обратной зависимости — например, при попытке преодолеть отказы в соединении с сервером большинство Web-браузеров пытается установить это соединение несколько раз, а для пользователя это выглядит как сильно замедленная реакция системы на его действия.
- ***Открытость (openness)***.
  Открытость системы определяется как полнота и ясность описания интерфейсов работы с ней и служб, которые она предоставляет через эти интерфейсы. Такое описание должно включать в себя все, что необходимо знать для того, чтобы пользоваться этими службами, независимо от реализации данной системы и платформы, на которой она развернута. Один из основных элементов описания службы — ее контракт.
  Открытость системы важна как для обеспечения ее переносимости, так и для облегчения использования системы и возможности построения других систем на ее основе. Распределенные системы обычно строятся с использованием служб, предоставляемых другими системами, и в то же время сами часто являются составными элементами или поставщиками служб для других систем.

Именно поэтому использование компонентных технологий при разработке практически полезного распределенного ПО неизбежно.

- ***Масштабируемость (scalability)***.
  Масштабируемость системы — это зависимость изменения ее характеристик от количества ее пользователей и подключенных ресурсов, а также от степени географической распределенности системы. В число значимых характеристик при этом попадают функциональность, производительность, стоимость, трудозатраты на разработку, на внесение изменений, на сопровождение, на администрирование, удобство работы с системой. Для некоторых из них наилучшая возможная масштабируемость обеспечивается линейной зависимостью, для других хорошая масштабируемость означает, что показатель не меняется вообще при изменении масштабов системы или изменяется незначительно. Система хорошо масштабируема по производительности, если параметры задач, решаемых ею за одно и то же время, можно увеличивать достаточно быстро (лучше — линейно или еще быстрее, но это возможно не для всех задач) при возрастании количества имеющихся ресурсов, в частности, отдельных машин. Однако, очень плохо, если внесение изменений в систему становится все более трудоемким при ее росте, даже если этот рост линейный, — желательно, чтобы трудоемкость внесения одного изменения почти не возрастала. Для функциональности же, опять, чем быстрее растет число доступных функций при росте числа вовлеченных в систему элементов, тем лучше.
  Большую роль играет *административная масштабируемость* системы — зависимость удобства работы с ней от числа административно независимых организаций, вовлеченных в ее обслуживание.
  При реализации очень больших систем (поддерживающих работу тысяч и более пользователей, включающих сотни и более машин) хорошая масштабируемость может быть достигнута только с помощью децентрализации основных служб системы и управляющих ею алгоритмов. Вариантами такого подхода являются следующие.
  - Децентрализация обработки запросов за счет использования для этого нескольких машин.
  - Децентрализация данных за счет использования нескольких хранилищ данных или нескольких копий одного хранилища.
  - Децентрализация алгоритмов работы за счет использования для алгоритмов:
    - не требующих полной информации о состоянии системы;
    - способных продолжать работу при сбое одного или нескольких ресурсов системы;
    - не предполагающих единого хода времени на всех машинах, входящих в систему.
  - Использование, где это возможно, *асинхронной связи* — передачи сообщений без приостановки работы до прихода ответа.
  - Использование комбинированных систем организации взаимодействия, основанных на следующих схемах.
    - Иерархическая организация систем, хорошо масштабирующая задачи поиска информации и ресурсов.
    - *Репликация* — построение копий данных и их распределение по системе для балансировки нагрузки на разные ее элементы. Частным случаем репликации является *кэширование*, при котором результаты наиболее часто используемых запросов запоминаются и хранятся как можно ближе к клиенту, чтобы переиспользовать их при повторении запросов.
    - Взаимодействие точка-точка (peer-to-peer, P2P) обеспечивает независимость взаимодействующих машин от других машин в системе.
- ***Безопасность (safety)***.
  Так как распределенные системы вовлекают в свою работу множество пользователей,

машин и географически разделенных элементов, вопросы их безопасности получают гораздо большее значение, чем при работе обычных приложений, сосредоточенных на одной физической машине. Это связано как с невозможностью надежно контролировать доступ к различным элементам такой системы, так и с ее доступностью для гораздо более широкого и разнообразного по своему поведению сообщества пользователей.
Понятие безопасности включает следующие характеристики.

- *Сохранность и целостность данных.*
  При обеспечении групповой работы многих пользователей с одними и теми же данными нужно обеспечивать их сохранность (т.е. предотвращать исчезновение данных, введенных одним из пользователей) и в тоже время целостность, т.е. непротиворечивость, выполнение всех присущих данным ограничений.
  Это непростая задача, которая не имеет решения, удовлетворяющего все стороны во всех ситуациях, — при одновременном изменении одного и того же элемента данных разными пользователями итоговый результат должен быть непротиворечив и поэтому часто может совпадать только с вводом одного из них. Как будет обработана такая ситуация и возможно ли ее возникновение вообще, зависит от дополнительных требований к системе, от принятых протоколов работы, от того, какие риски — потерять данные одного из пользователей или значительно усложнить работу пользователей с системой — будут сочтены более важными.
- *Защищенность данных и коммуникаций.*
  При работе с коммерческими системами, содержащими большие объемы персональной и бизнес-информации, а также с системами обслуживания пользователей государственных ведомств очень важна защищенность как информации, постоянно хранящейся в системе, так и информации одного сеанса работы. Для распределенных систем обеспечить защищенность гораздо сложнее, поскольку нельзя физически изолировать все элементы системы и разрешить доступ к ней только проверенным и обладающим необходимыми знаниями и умениями людям.
- *Отказоустойчивость и способность к восстановлению после ошибок.*
  Одним из достоинств распределенных систем является возможность построения более надежно работающей системы из не вполне надежных компонентов. Однако для того, чтобы это достоинство стало реальным, необходимо тщательное проектирование систем с тем, чтобы избежать зависимости работоспособности системы в целом от ее отдельных элементов. Иначе достоинство превращается в недостаток, поскольку в распределенной системе элементов больше и выше вероятность того, что хотя бы один элемент выйдет из строя и хотя бы один ресурс окажется недоступным.
  Еще важнее для распределенных систем уметь восстанавливаться после сбоев. Уровни этого восстановления могут быть различными. Обычно данные одного короткого сеанса работы считается возможным не восстанавливать, поскольку такие данные часто малозначимы или легко восстанавливаются (иначе стоит серьезно рассмотреть необходимость восстановления сеансов). Но так называемые постоянно хранимые (persistent) данные чаще всего требуется восстанавливать в их последнем непротиворечивом состоянии.

Перед разработчиками систем, удовлетворяющих перечисленным свойствам, встает огромное количество проблем. Решать их все сразу просто невозможно в силу ограниченности человеческих способностей. Чтобы хоть как-то структурировать эти проблемы, их разделяют по следующим аспектам [3].

- **Связь.**
  Организация связи и передачи данных между элементами системы.
  В связи с этим аспектом возникают следующие задачи.
  - Какие протоколы использовать для передачи данных.

- Как реализовать обращения к процедурам и методам объектов одних процессов из других.
- Какой способ передачи данных выбрать — ***синхронный*** или ***асинхронный***. В первом случае сторона, инициировавшая передачу, приостанавливает свою работу до прихода ответа другой стороны на переданное сообщение. Во втором случае первая сторона имеет возможность продолжить работу, пока данные передаются и обрабатываются другой стороной.
- Нужно ли, и если нужно, то как, организовать хранение (асинхронных) сообщений в то время, когда и отправитель и получатель сообщения могут быть неактивны.
- Как организовать передачу непрерывных потоков данных, представляющих собой аудио-, видеоданные или смешанные потоки данных. Этот вопрос имеет большое значение, поскольку заметные человеку прерывания в передаче таких данных приводят к значительному падению качества предоставляемых услуг.

- **Именование.**
  Поддержка идентификации и поиска отдельных ресурсов внутри системы.
  - По каким правилам присваивать имена и идентификаторы различным ресурсам.
  - Как организовать поиск ресурсов в системе по идентификаторам и атрибутам, описывающим какие-нибудь свойства ресурсов.
  - Как размещать и находить мобильные ресурсы, изменяющие свое физическое положение в ходе работы.
  - Как организовывать и поддерживать в рабочем состоянии сложные ссылочные структуры, необходимые для описания имеющихся в распределенной системе ресурсов. Как, например, находить и удалять ресурсы, ставшие никому не доступными.
- **Процессы.**
  Организация работ в рамках процессов и потоков.
  - Как разделить работы в системе по отдельным процессам и машинам.
  - Нужно ли определять различные ролей процессов в системе, например, клиентские и серверные, и как организовывать их работу.
  - Как организовать работу исполняемых ***агентов*** — процессов, способных перемещаться между машинами и выполнять свои задачи в любой подходящей среде.
- **Синхронизация.**
  Синхронизация параллельно выполняемых потоков работ.
  - Как синхронизовать действия отдельных процессов и потоков, работающих в системе, для получения нужных результатов.
  - Как организовать работу многих процессов на разных машинах в том случае, если в системе нельзя непротиворечиво определить глобальное время.
  - Как организовать выполнение транзакций — таких наборов действий, которые надо либо все выполнить, либо не выполнить ни одного из них.
- **Целостность.**
  Поддержка целостности данных и непротиворечивости вносимых изменений.
  - Каким образом можно обеспечивать целостность данных.
  - Какие *модели непротиворечивости* нужно поддерживать. Модель непротиворечивости определяет, на основе каких требований формируются результаты выполняемых одновременно изменений и что доступно клиентам, выполнявшим эти изменения.
  - Какие протоколы обеспечения непротиворечивости, создания и записи транзакций, создания и согласования реплик и кэшей использовать для выполнения требований этих моделей.

- **Отказоустойчивость.**
  Организация отказоустойчивой работы.
  - Как организовать отказоустойчивую работу одного процесса.
  - Как обеспечить надежную связь между элементами системы.
  - Какие протоколы использовать для реализации надежной двусторонней связи или надежных групповых рассылок.
  - Какие протоколы использовать для записи промежуточных состояний и восстановления данных и работы системы после сбоев.
- **Защита.**
  Организация защищенности данных и коммуникаций.
  - Как организовать защиту системы в целом.
    При этом большее значение, чем технические аспекты, имеют организационные и психологические факторы — проблемы определения процедур проведения работ, обеспечивающих нужный уровень защищенности, и проблемы соблюдения людьми этих процедур.
  - Как организовать защиту данных от несанкционированного доступа.
  - Как обеспечить защиту каналов связи от двух видов атак — воспрепятствовать несанкционированному доступу к передаваемой информации и не дать подменить информацию в канале.
  - Какие протоколы аутентификации пользователей, подтверждения идентичности и авторства использовать.

Из перечисленных тем отдельного рассмотрения заслуживают вопросы организации передачи сообщений и транзакций, тем более что все рассматриваемые далее технологии используют эти механизмы. Более того, практически любая распределенная система сейчас строится на основе ***программного обеспечения промежуточного уровня*** (***middleware***, это программное обеспечение, которое предназначено для облегчения интеграции ПО, размещенного на нескольких машинах, в единую распределенную систему и поддержки работы такой системы), содержащего ту или иную их реализацию.

Синхронное и асинхронное взаимодействие

При описании взаимодействия между элементами программных систем инициатор взаимодействия, т.е. компонент, посылающий запрос на обработку, обычно называется ***клиентом***, а отвечающий компонент, тот, что обрабатывает запрос — ***сервером***. «Клиент» и «сервер» в этом контексте обозначают роли в рамках данного взаимодействия. В большинстве случаев один и тот же компонент может выступать в разных ролях — то клиента, то сервера — в различных взаимодействиях. Лишь в небольшом классе систем роли клиента и сервера закрепляются за компонентами на все время их существования.

Синхронным (synchronous) называется такое взаимодействие между компонентами, при котором клиент, отослав запрос, блокируется и может продолжать работу только после получения ответа от сервера. По этой причине такой вид взаимодействия называют иногда ***блокирующим (blocking)***.

Обычное обращение к функции или методу объекта с помощью передачи управления по стеку вызовов является примером синхронного взаимодействия.

Синхронное взаимодействие достаточно просто организовать, и оно гораздо проще для понимания. Человеческое сознание обладает единственным «потоком управления», представленным в виде фокуса внимания, и поэтому человеку проще понимать процессы, которые разворачиваются последовательно, поскольку не нужно постоянно переключать внимание на происходящие одновременно различные события. Код программы клиентского компонента, описывающей синхронное взаимодействие, устроен проще — его часть, отвечающая за обработку

ответа сервера, находится непосредственно после части, в которой формируется запрос. В силу своей простоты синхронные взаимодействия в большинстве систем используются гораздо чаще асинхронных.

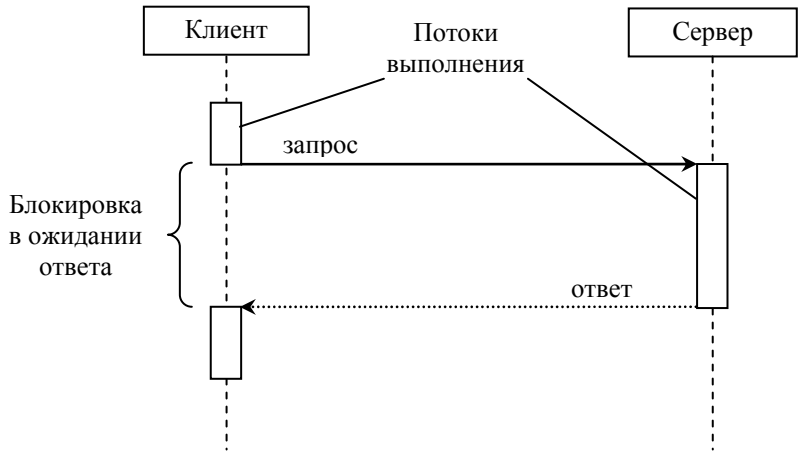

Рисунок 66. Синхронное взаимодействие.

Вместе с тем синхронное взаимодействие ведет к значительным затратам времени на ожидание ответа. Это время часто можно использовать более полезным образом: ожидая ответа на один запрос, клиент мог бы заняться другой работой, выполнить другие запросы, которые не зависят от еще не пришедшего результата. Поскольку все распределенные системы состоят из достаточно большого числа уровней, через которые проходят практически все взаимодействия, суммарное падение производительности, связанное с синхронностью взаимодействий, оказывается очень большим.

Наиболее распространенным и исторически первым достаточно универсальным способом реализации синхронного взаимодействия в распределенных системах является ***удаленный вызов процедур*** (***Remote Procedure Call, RPC***, вообще-то, по смыслу правильнее было бы сказать «дистанционный вызов процедур», но по историческим причинам закрепилась имеющаяся терминология). Его модификация для объектно-ориентированной среды называется ***удаленным вызовом методов (Remote Method Invocation, RMI)***. Удаленный вызов процедур определяет как способ организации взаимодействия между компонентами, так и методику разработки этих компонентов.

На первом шаге разработки определяется интерфейс процедур, которые будут использоваться для удаленного вызова. Это делается при помощи *языка определения интерфейсов (Interface Definition Language, IDL)*, в качестве которого может выступать специализированный язык или обычный язык программирования, с ограничениями, определяющимися возможностью передачи вызовов на удаленную машину.

Определение процедуры для удаленных вызовов компилируется компилятором IDL в описание этой процедуры на языках программирования, на которых будут разрабатываться клиент и сервер (например, заголовочные файлы на C/C++), и два дополнительных компонента — ***клиентскую*** и ***серверную заглушки (client stub*** и ***server stub)***.

Клиентская заглушка представляет собой компонент, размещаемый на той же машине, где находится компонент-клиент. Удаленный вызов процедуры клиентом реализуется как обычный, локальный вызов определенной функции в клиентской заглушке. При обработке этого вызова клиентская заглушка выполняет следующие действия.

1. Определяется физическое местонахождение в системе сервера, для которого предназначен данный вызов. Это шаг называется ***привязкой (binding)*** к серверу. Его результатом является адрес машины, на которую нужно передать вызов.

2. Вызов процедуры и ее аргументы упаковываются в сообщение в некотором формате, понятном серверной заглушке (см. далее). Этот шаг называется ***маршалингом (marshaling)***.
3. Полученное сообщение преобразуется в поток байтов (это *сериализация, serialization*) и отсылается с помощью какого-либо протокола, транспортного или более высокого уровня, на машину, на которой помещен серверный компонент.
4. После получения от сервера ответа, он распаковывается из сетевого сообщения и возвращается клиенту в качестве результата работы процедуры.

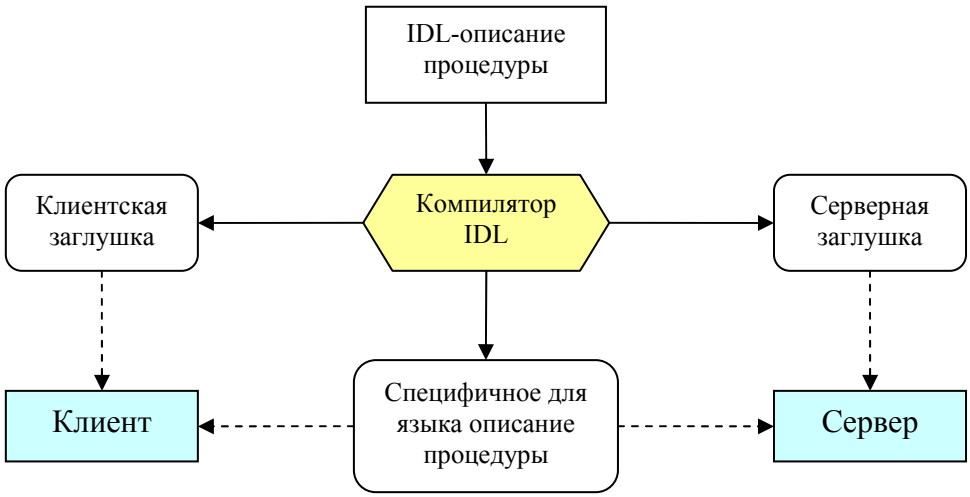

Рисунок 67. Схема разработки компонентов, взаимодействующих с помощью RPC.

В результате для клиента удаленный вызов процедуры выглядит как обращение к обычной функции.

Серверная заглушка располагается на той же машине, где находится компонент-сервер. Она выполняет операции, обратные к действиям клиентской заглушки — принимает сообщение, содержащее аргументы вызова, распаковывает эти аргументы при помощи *десериализации (deserialization)* и *демаршалинга (unmarshaling)*, вызывает локально соответствующую функцию серверного компонента, получает ее результат, упаковывает его и посылает по сети на клиентскую машину.

Таким образом обеспечивается отсутствие видимых серверу различий между удаленным вызовом некоторой его функции и ее же локальным вызовом.

Определив интерфейс процедур, вызываемых удаленно, мы можем перейти к разработке сервера, реализующего эти процедуры, и клиента, использующего их для решения своих задач.

При удаленном вызове процедуры клиентом его обращение оформляется так же, как вызов локальной функции и обрабатывается клиентской заглушкой. Клиентская заглушка определяет адрес машины, на которой находится сервер, упаковывает данные вызова в сообщение и отправляет его на серверную машину. На серверной машине серверная заглушка, получив сообщение, распаковывает его, извлекает аргументы вызова, обращается к серверу с таким вызовом локально, дожидается от него результата, упаковывает результат в сообщение и отправляет его обратно на клиентскую машину. Получив ответное сообщение, клиентская заглушка распаковывает его и передает полученный ответ клиенту.

Эта же техника может быть использована и для реализации взаимодействия компонентов, работающих в рамках различных процессов на одной машине.

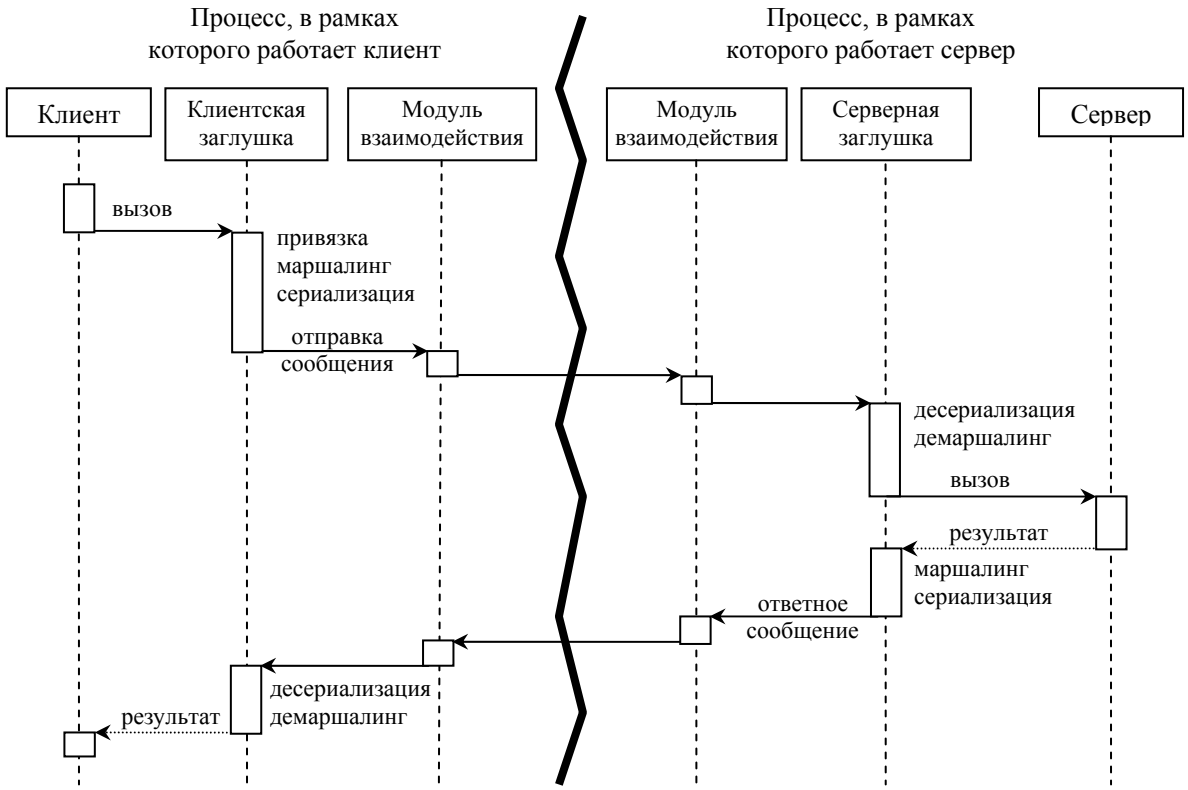

Рисунок 68. Схема реализации удаленного вызова процедуры.

При организации ***удаленного вызова методов*** в объектно-ориентированной среде применяются такие же механизмы. Отличия в его реализации связаны со следующими аспектами.

- Один объект-сервер может предоставлять несколько методов для удаленного обращения к ним.
  Для такого объекта генерируются клиентские заглушки, имеющие в своем интерфейсе все эти методы. Кроме того, информация о том, какой именно метод вызывается, должна упаковываться вместе с аргументами вызова и использоваться серверной заглушкой для обращения именно к этому методу.
  Серверная заглушка в контексте RMI иногда называется ***скелетоном (skeleton)*** или ***каркасом***.
- В качестве аргументов удаленного вызова могут выступать объекты.
  Заметим, что передача указателей в аргументах удаленного вызова процедур практически всегда запрещена — указатели привязаны к памяти данного процесса и не могут быть переданы в другой процесс.
  При передаче объектов возможны два подхода, и оба они используются на практике.
  - Идентичность передаваемого объекта может не иметь значения, например, если сервер использует его только как хранилище данных и получит тот же результат при работе с правильно построенной его копией. В этом случае определяются методы для сериализации и десериализации данных объекта, которые позволяют сохранить их в виде потока байтов и восстановить объект с теми же данными на другой стороне.
  - Идентичность передаваемого объекта может быть важна, например, если сервер вызывает в нем методы, работа которых зависит от того, что это за объект. При этом используется особого рода ссылка на этот объект, позволяющая обратиться к нему из другого процесса или с другой машины, т.е. тоже с помощью удаленного вызова методов.

В рамках ***асинхронного (asynchronous)*** или ***неблокирующего (non blocking)*** взаимодействия клиент после отправки запроса серверу может продолжать работу, даже если ответ на запрос еще не пришел.

Примером асинхронного взаимодействия является электронная почта. Другой пример — распространение сообщений о новостях различных видов в соответствии с имеющимся на текущий момент реестром подписчиков, где каждый подписчик определяет темы, которые его интересуют.

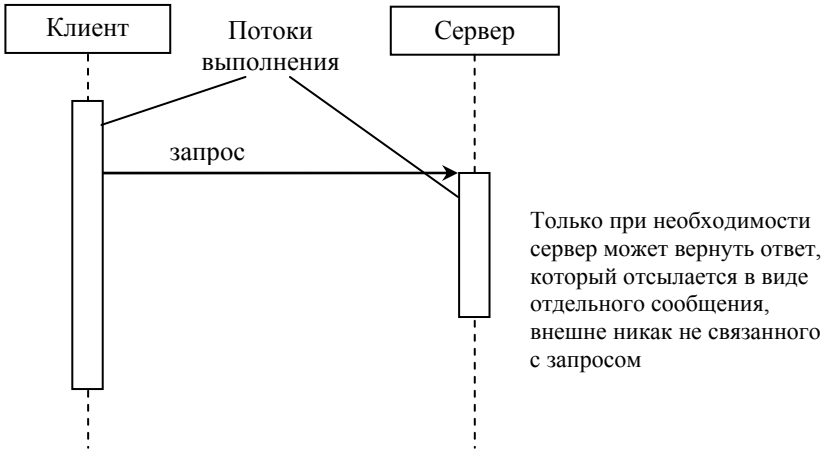

Рисунок 69. Асинхронное взаимодействие.

Асинхронное взаимодействие позволяет получить более высокую производительность системы за счет использования времени между отправкой запроса и получением ответа на него для выполнения других задач. Другое важное преимущество асинхронного взаимодействия — меньшая зависимость клиента от сервера, возможность продолжать работу, даже если машина, на которой находится сервер, стала недоступной. Это свойство используется для организации надежной связи между компонентами, работающей, даже если и клиент, и сервер не все время находятся в рабочем состоянии.

В то же время асинхронные взаимодействия более сложно использовать. Поскольку при таком взаимодействии нужно писать специфический код для получения и обработки результатов запросов, системы, основанные на асинхронных взаимодействиях между своими компонентами, значительно труднее разрабатывать и сопровождать.

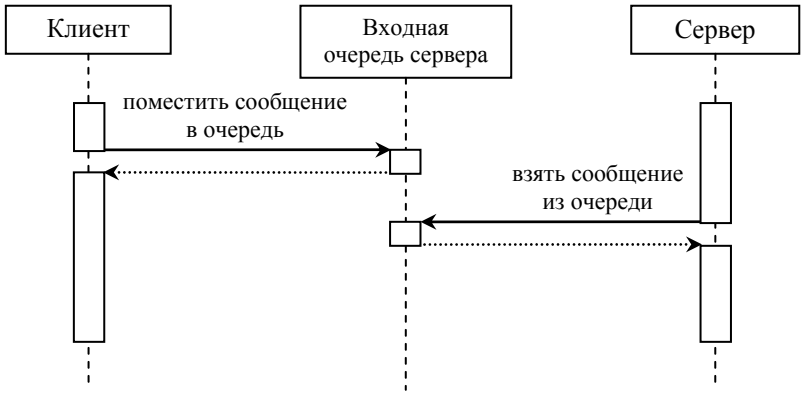

Рисунок 70. Реализация асинхронного взаимодействия при помощи очередей сообщений.

Чаще всего асинхронное взаимодействие реализуется при помощи очередей сообщений. При отправке сообщения клиент помещает его во входную очередь сервера, а сам продолжает работу. После того, как сервер обработает все предшествующие сообщения в очереди, он выбирает это сообщение для обработки, удаляя его из очереди. После обработки, если необходим ответ, сервер

создает сообщение, содержащее результаты обработки, и кладет его во входную очередь клиента или в свою выходную.

Очереди сообщений могут быть сконфигурированы самыми разными способами. У компонента может иметься одна входная очередь, а может — и несколько, для сообщений от разных источников или имеющих разный смысл. Кроме того, компонент может иметь выходную очередь, или несколько, вместо того, чтобы класть сообщения во входные очереди других компонентов. Очереди сообщений могут храниться независимо как от компонентов, которые кладут туда сообщения, так и от тех, которые забирают их оттуда. Сообщения в очередях могут иметь приоритеты, а сама очередь — реализовывать различные политики поддержания или изменения приоритетов сообщений в ходе работы.

Транзакции

Понятие транзакции пришло в инженерию ПО из бизнеса и используется чаще всего (но все же не всегда) для реализации функциональности, связанной с обеспечением различного рода сделок.

Примером, поясняющим необходимость использования транзакций, является перевод денег с одного банковского счета на другой. При переводе соответствующая сумма должна быть снята с первого счета и добавиться к уже имеющимся деньгам на втором. Если между первой и второй операцией произойдет сбой, например, пропадет связь между банками, деньги исчезнут с первого счета и не появятся на втором, что явно не устроит их владельца. Перестановка операций местами не помогает — при сбое между ними ровно такая же сумма возникнет из ничего на втором счете. В этом случае недоволен будет банк, поскольку он должен будет выплатить эти деньги владельцу счета, хотя сам их не получал.

Выход из этой ситуации один — сделать так, чтобы либо обе эти операции выполнялись, либо ни одна из них не выполнялась. Такое свойство обеспечивается их объединением в одну транзакцию.

Транзакции представляют собой группы действий, обладающие следующим набором свойств.

- *Атомарность (atomicity)*. Для окружения транзакция неделима — она либо выполняется целиком, либо ни одно из ее действий транзакции не выполняется. Другие процессы не имеют доступа к промежуточным результатам транзакции.
- *Непротиворечивость (consistency)*. Транзакция не нарушает инвариантов и ограничений целостности данных системы.
- *Изолированность (isolation)*. Одновременно происходящие транзакции не влияют друг на друга. Это означает, что несколько транзакций, выполнявшихся параллельно, производят такой суммарный эффект, как будто они выполнялись в некоторой последовательности. Сама эта последовательность определяется внутренними механизмами реализации транзакций. Это свойство также называют *сериализуемостью* транзакций, поскольку любой сценарий их выполнения эквивалентен некоторой их последовательности или серии.
- *Долговечность (durability)*. После завершения транзакции сделанные ею изменения становятся постоянными и доступными для выполняемых в дальнейшем операций. Если транзакция завершилась, никакие сбои не могут отменить результаты ее работы.

По первым буквам английских терминов для этих свойств, их часто называют ACID.

Свойствами ACID во всей полноте обладают так называемые *плоские транзакции (flat transactions)*, самый распространенный вариант транзакций. Иногда требуется гораздо более сложное поведение, в рамках которого нужно уметь выполнять или отменять только часть операций в составе транзакции; бывают случаи, когда процессам, не участвующим в транзакции, нужно уметь получить ее промежуточные результаты. Сокрытие промежуточных результатов часто накладывает слишком сильные ограничения на работу системы, если транзакция продолжается заметное время (а иногда их выполнение требует нескольких месяцев!). Для решения таких задач применяются механизмы, допускающие вложенность транзакций друг в

друга, длинные транзакции, позволяющие получать доступ к своим промежуточным результатам, и пр.

Одним из широко распространенных видов программного обеспечения промежуточного уровня являются *мониторы транзакций (transaction monitors)*, обеспечивающие выполнение удаленных вызовов процедур с поддержкой транзакций. Такие транзакции часто называют ***распределенными***, поскольку участвующие в них процессы могут работать на разных машинах.

Рисунок 71. Схема реализации поддержки распределенных транзакций.

Для организации таких транзакций необходим *координатор*, который получает информацию обо всех участвующих в транзакции действиях и обеспечивает ее атомарность и изолированность от других процессов. Обычно транзакции реализуются при помощи примитивов, позволяющих *начать транзакцию*, *завершить ее успешно (commit)*, с сохранением всех сделанных изменений, и *откатить транзакцию (rollback)*, отменив все выполненные в ее рамках действия.

Примитив «начать транзакцию» сообщает координатору о необходимости создать новую транзакцию, зарегистрировать начавший ее объект как участника и передать ему идентификатор транзакции. При передаче управления (в том числе с помощью удаленного вызова метода) участник транзакции передает вместе с обычными данными ее идентификатор. Компонент, операция которого была вызвана в рамках транзакции, сообщает координатору идентификатор транзакции с тем, чтобы координатор зарегистрировал и его как участника этой же транзакции.

Если один из участников не может выполнить свою операцию, выполняется откат транзакции. При этом координатор рассылает всем зарегистрированным участникам сообщения о необходимости отменить выполненные ими ранее действия.

Если вызывается примитив «завершить транзакцию», координатор выполняет некоторый протокол подтверждения, чтобы убедиться, что все участники выполнили свои действия успешно и можно открыть результаты транзакции для внешнего мира. Наиболее широко используется ***протокол двухфазного подтверждения (Two-phase Commit Protocol, 2PC)*** [3,4], который состоит в следующем.

1. Координатор посылает каждому компоненту-участнику транзакции запрос о подтверждении успешности его действий.

2. Если данный компонент выполнил свою часть операций успешно, он возвращает координатору подтверждение.
   Иначе — он посылает сообщение об ошибке.
3. Координатор собирает подтверждения всех участников и, если все зарегистрированные участники транзакции присылают подтверждения успешности, рассылает им сообщение о подтверждении транзакции в целом. Если хотя бы один участник прислал сообщение об ошибке или не ответил в рамках заданного времени, координатор рассылает сообщение о необходимости отменить транзакцию.
4. Каждый участник, получив сообщение о подтверждении транзакции в целом, сохраняет локальные изменения, сделанные в рамках транзакции.
   Если же он получить сообщение об отмене транзакции, он отменяет локальные изменения.

Аналог протокола двухфазного подтверждения используется, например, в компонентной модели JavaBeans для уведомления об изменениях свойств компонента, которые некоторые из оповещаемых о них компонентов-подписчиков могут отменить [5,6]. При этом до внесения изменений о них надо оповестить с помощью метода `vetoableChange()` интерфейса `java.beans.VetoableChangeListener`. Если хотя бы один из подписчиков требует отменить изменение с помощью создания исключения типа `java.beans.PropertyVetoException`, его надо откатить, сообщив об этом остальным подписчикам. Если же все согласны, то после внесения изменений о них, как об уже сделанных, оповещают с помощью метода `propertyChange()` интерфейса `java.beans.PropertyChangeListener`.

Литература к Лекции 12

[1] C. Szyperski. Component Software Beyond Object-Oriented Programming. Boston, MA: Addison-Wesley and ACM Press, 1998.

[2] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition/ Technical Report CMU/SEI-2000-TR-008.
Доступен как http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf.

[3] Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003.

[4] G. Alonso, F. Casati, H. Kuno, V. Machiraju. Web Services. Concepts, Architectures and Applications. Springer-Verlag, 2004.

[5] JavaBeans Specification 1.01. Доступна через страницу http://java.sun.com/products/javabeans/docs/spec.html.

[6] Документация по библиотекам J2SE http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[7] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. Pattern-Oriented Software Architecture. Volume 2. Patterns for Concurrent and Networked Objects. Wiley, 2000.