# Лекция 6. Компонентные технологии и разработка распределенного ПО. Продолжение.

**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.

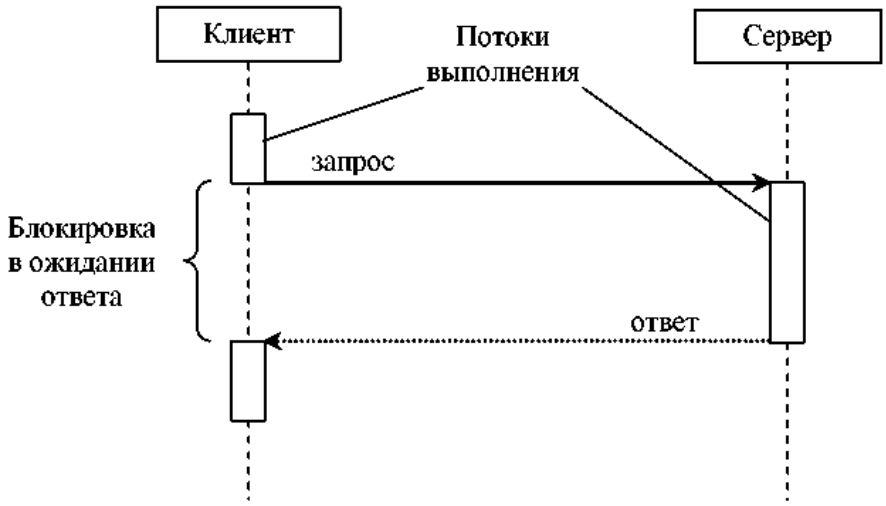


Рисунок 66. Синхронное взаимодействие

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.

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.

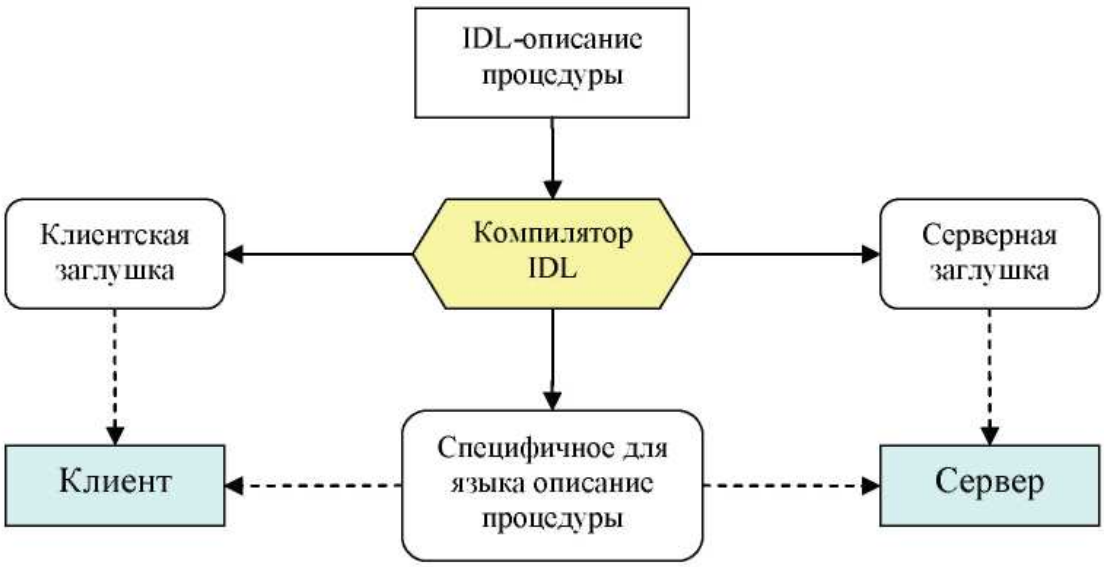


Рисунок 67. Схема разработки компонентов, взаимодействующих с помощью RPC

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).* 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.
3. После получения от сервера ответа, он распаковывается из сетевого сообщения и возвращается клиенту в качестве результата работы процедуры.

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.

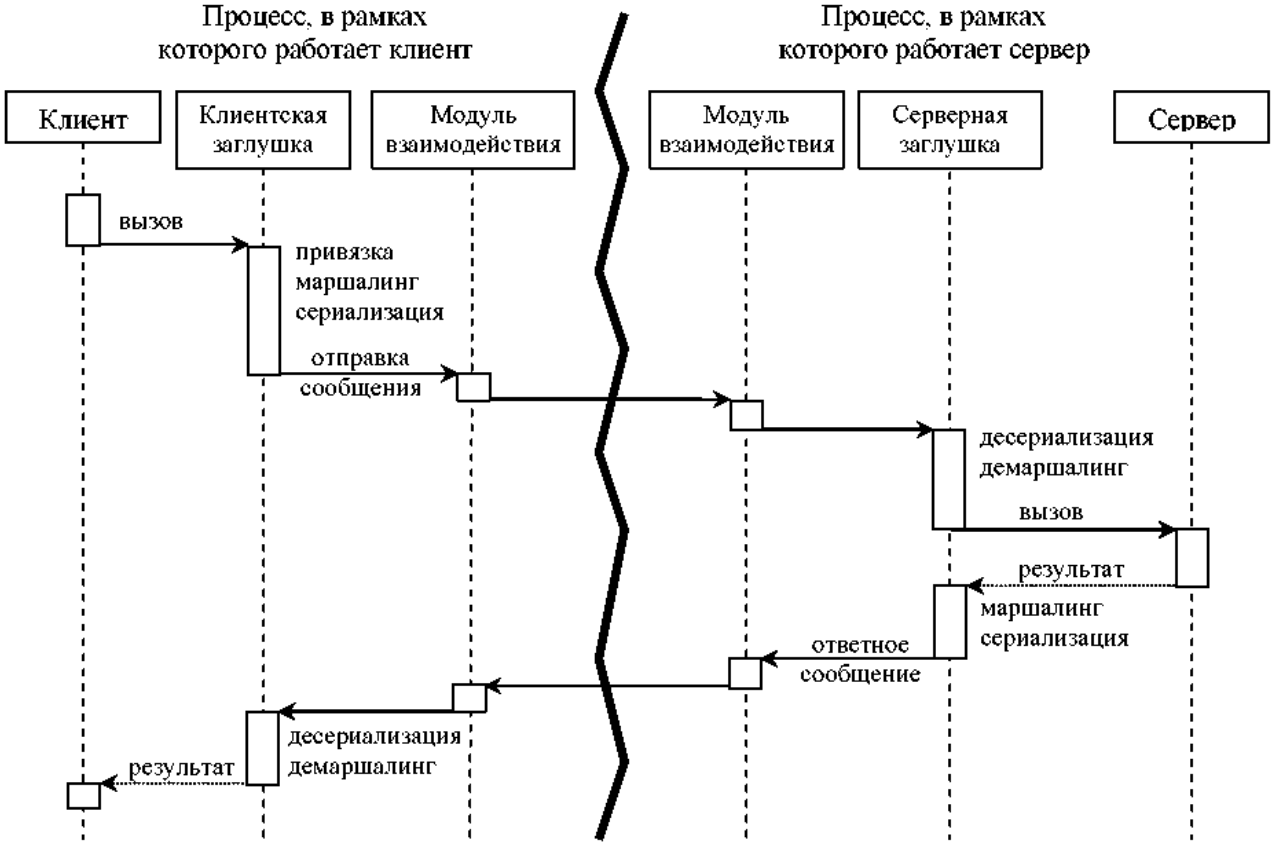


Рисунок 68. Схема реализации удаленного вызова процедуры

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.

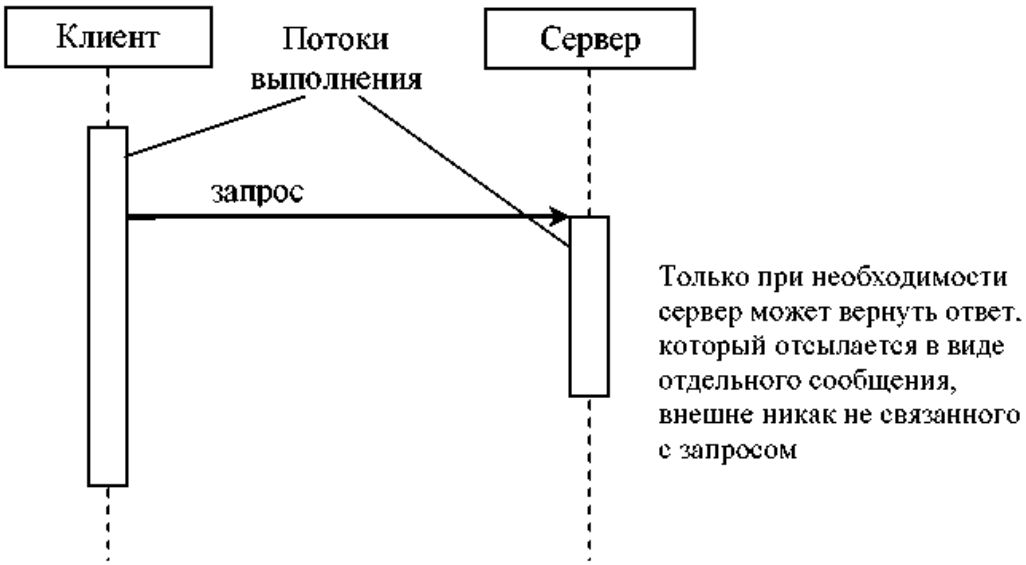


Рисунок 69. Асинхронное взаимодействие

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.

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.

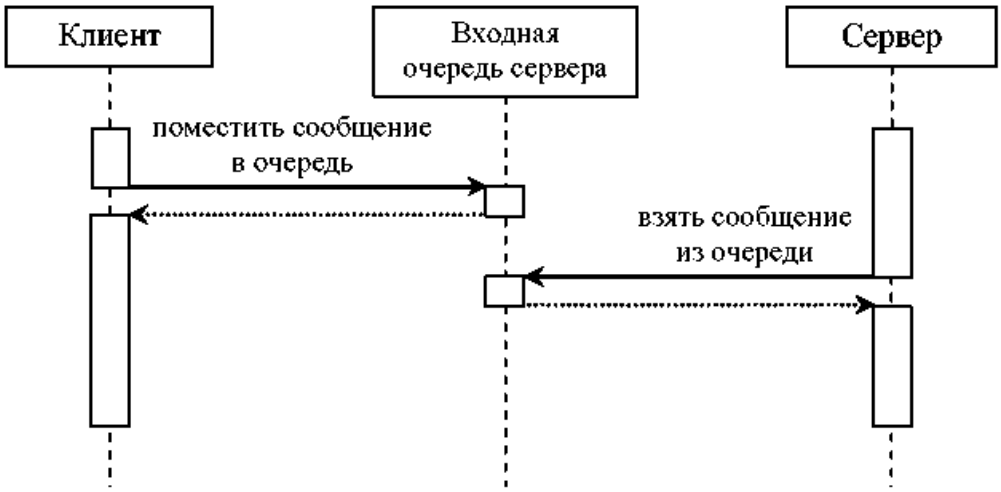


Рисунок 70. Реализация асинхронного взаимодействия при помощи очередей сообщений

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 *сериализуемостъю* 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), 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 получает 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. 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.