ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В настоящее время практически все большие программные системы являются распределенными. *Распределенная система (distributed system)* - система, в которой хранение данных и их обработка сосредоточена не на одной вычислительной машине, а распределена между несколькими компьютерами.

В первой части методических указаний «Технология программирования» [1] были предложены основные приемы моделирования сложных информационных систем с использованием языка визуального моделирования UML и рассмотрены вопросы объектно-ориентированного подхода к анализу и проектированию программного обеспечения.

Во второй части методических указаний [2] были рассмотрены особенности полного инженерного цикла разработки Web-приложений на платформе Microsoft .NET.

Третья часть методических указаний предназначена для изучения особенностей разработки архитектуры распределенных систем.

Практикум предназначен для студентов специальности «Информатика и вычислительная техника» и «Программная инженерия», изучающих информационные технологии в рамках дисциплин «Технология программирования» и «Методология программной инженерии».

ОСОБЕННОСТИ ПРОЕКТИРОВАНИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Распределенные системы нужны для повышения производительности (т.е. вычислительной мощности и доступного объема хранимых данных) и надежности отдельных компьютеров. Разделение ресурсов между группой компьютеров позволяет отдельным пользователям получать совместный доступ к ресурсам, гарантируя, что их должно хватить для выполнения крупной текущей задачи. Распределенные системы позволяют достичь высокой производительности с меньшими затратами, чем при построении высокопроизводительной системы на основе одного компьютера, но они гораздо сложнее в реализации и обслуживании.

Проектирование распределенных систем имеет много общего с проектированием ПО в общем, но все же следует учитывать некоторые специфические особенности.

Существует шесть основных преимуществ распределенных систем:

- системы допускают совместное использование как аппаратных так и программных ресурсов;
- возможно расширение системы путем добавления новых ресурсов;
- в системах несколько процессов могут одновременно выполняться на разных компьютерах в сети. Эти процессы могут взаимодействовать во время их выполнения;
- в системах есть возможность добавления новых свойств и методов;
- системы позволяют дублирование информации и устойчивость к некоторым аппаратным и программным ошибкам. Распределенные системы в случае ошибки могут поддерживать частичную функциональность. Полный сбой в работе системы происходит только при сетевых ошибках;
- пользователям предоставляется полный доступ к ресурсам в системе, в то же время от них скрыта информация о распределении ресурсов.

Распределенные системы обладают и рядом недостатков.

Намного труднее понять и оценить свойства распределенных систем в целом, их сложнее проектировать, тестировать и обслуживать. Также производительность системы зависит от скорости работы сети, а не отдельных процессоров. Перераспределение ресурсов может существенно изменить скорость работы системы.

Обычно доступ к системе можно получить с нескольких разных машин, сообщения в сети могут просматриваться и перехватываться. Поэтому в распределенной системе намного труднее поддерживать безопасность.

Оглавление

Система может состоять из разнотипных компьютеров, на которых могут быть установлены различные версии операционных систем. Ошибки на одной машине могут распространиться непредсказуемым образом на другие машины.

Реакция распределенных систем на некоторые события непредсказуема и зависит от полной загрузки системы, ее организации и сетевой нагрузки. Так как эти параметры могут постоянно изменятся, поэтому время ответа на запрос может существенно отличаться от требуемого времени.

Из этих недостатков можно увидеть, что при проектировании распределенных систем возникает **ряд проблем, которые надо учитывать разработчикам**.

Ресурсы в распределенных системах располагаются на разных компьютерах, поэтому систему имен ресурсов следует продумать так, чтобы пользователи могли без труда открывать необходимые им ресурсы и ссылаться на них. Примером может служить система URL (унифицированный указатель ресурсов), которая определяет имена Web-страниц.

Универсальная работоспособность Internet и эффективная реализация протоколов TCP/IP в Internet для большинства распределенных систем служат примером наиболее эффективного способа организации взаимодействия между компьютерами. Однако в некоторых случаях, когда требуется особая производительность или надежность, возможно использование специализированных средств.

Для дальнейшего рассмотрения вопросов проектирования распределенных систем сформулируем основные их атрибуты и способы улучшения значений этих атрибутов при проектировании систем.

АТРИБУТЫ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Масштабируемость

Масштабируемость системы (scalability) — это зависимость изменения ее характеристик от числа ее пользователей, числа 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 ей 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. В то же время, очень плохо, если внесе-

ние 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 чем быстрее растет число доступных функций при росте числа вовлеченных в систему элементов, чем 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.

На практике применяются следующие подходы [4,10]:

1. Децентрализация обработки запросов за счет использования нескольких машин.
2. Децентрализация данных за счет использования нескольких хранилищ данных или нескольких копий одного хранилища.
3. 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.
4. Использование, где это возможно, *асинхронной связи* — передачи сообщений без приостановки работы до прихода ответа.
5. Использование комбинированных систем организации взаимодействия, основанных на следующих схемах:
   - ***иерархическая организация систем***, хорошо масштабирующей задачи поиска информации и ресурсов;
   - ***репликация*** — построение копий данных и их распределении по системе для балансировки нагрузки на разные ее элементы;
   - ***кэширование*** – организация хранения результатов наиболее часто используемых запросов как можно ближе к клиенту;
   - ***взаимодействие точка-точка (peer-to-peer, P2P***), обеспечивающем независимость взаимодействующих машин от других машин в системе.

Прозрачность

Прозрачностью ***(transparency)*** 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, 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

Открытость (openness) 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. 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.

Безопасность

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 доступом к ней гораздо более широкого и разнообразного по своему поведению сообщества пользователей. **Понятие безопасности** (safety**) включает следующие характеристики:**

1. ***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, не имеющая 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.

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, не

имеющая 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.

Для поддержки целостности данных и непротиворечивости вносимых изменений необходимо определить:

- каким образом можно обеспечивать целостность данных;
- какие *моделей 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 согласования реплик и кэшей использовать для выполнения требований этих моделей.

2. ***Защищенность данных и коммуникаций.***

   При работе с коммерческими системами, с системами, содержащими большие объемы персональной и бизнес-информации, 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.

3. ***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 (если это не так — стоит серьезно рассмотреть необходимость восстановления сеансов). Но так называемые постоянно хранимые (persistent) данные чаще всего требуется восстанавливать до последнего непротиворечивого их состояния.

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 [4,11].

1. **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. Заметные человеку прерывания в передаче таких данных приводят к значительному падению качества предоставляемых услуг.
2. **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.
3. **Процессы.** Организация работ в рамках процессов и потоков.
    - 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.
4. **Синхронизация.** Синхронизация параллельно выполняемых потоков работ требует ответов на следующие вопросы:
    - 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?

5. **Целостность.** Для поддержки целостности данных и непротиворечивости вносимых изменений следует выбрать и обосновать:
   - Способ обеспечения целостности данных.
   - ***Модели непротиворечивости***. Модель непротиворечивости определяет, на основе каких требований формируются результаты выполняемых одновременно изменений и что доступно клиентам, выполнявшим эти изменения.
   - Протоколы обеспечения непротиворечивости, создания и согласования реплик и КЭШей, которые следует использовать для выполнения требований этих моделей.
6. **Отказоустойчивость.** Организация отказоустойчивой работы системы требует определения необходимых процедур и профилей стандартов для решения следующих задач:
   - Организовать отказоустойчивую работу одного процесса.
   - Обеспечить надежную связь между элементами системы.
   - Выбрать и обосновать протоколы, которые следует использовать для реализации надежной двусторонней связи или надежных групповых рассылок.
   - Выбрать и обосновать протоколы, которые следует использовать для записи промежуточных состояний и восстановления данных и работы системы после сбоев.

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 (если это не так — стоит серьезно рассмотреть необходимость восстановления сеансов). Но так называемые постоянно хранимые (persistent) данные чаще всего требуется восстанавливать до последнего непротиворечивого их состояния. Организация отказоустойчивой работы требует учета следующих вопросов:

- 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.

7. **Защита.** Для организации защищенности данных и коммуникаций следует рассмотреть следующие технические, организационные и психологические вопросы:
   - Для организации защиты системы в целом следует определить процедуры проведения работ, обеспечивающих нужный уровень защищенности, и правила соблюдения людьми этих процедур.
   - Организовать защиту данных от несанкционированного доступа.
   - Обеспечить защиту каналов связи с двух сторон — воспрепятствовать несанкционированному доступу к передаваемой информации и не дать подменить информацию в канале.
   - Выбрать протоколы аутентификации пользователей, подтверждения идентичности и авторства.

При работе с коммерческими системами, с системами, содержащими большие объемы персональной и бизнес-информации, 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. Организация защищенности данных и коммуникаций предполагает решение следующих задач:

- 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 с двух сторон — 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. Практически любая распределенная система сейчас строится на основе ***программного обеспечения промежуточного уровня* (middleware).**

Это программное обеспечение, предназначенное для облегчения интеграции ПО, размещенного на нескольких машинах, в единую распределенную систему и поддержки работы такой системы.

ОБЩИЕ ПРИНЦИПЫ ПОСТРОЕНИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ

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.

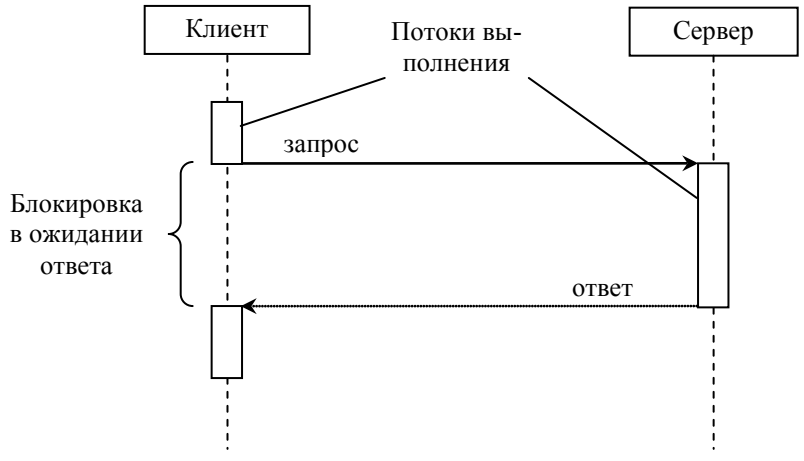


Рисунок 1. Синхронное взаимодействие.

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.

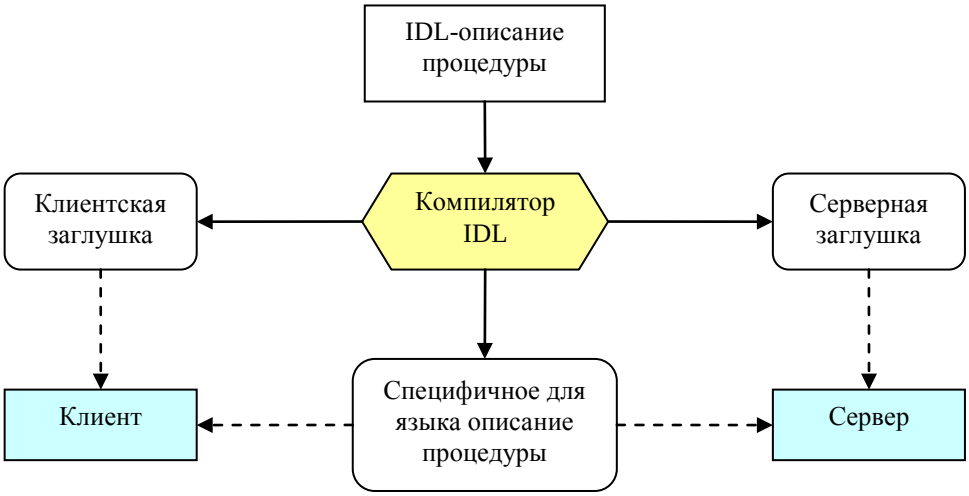


Рисунок 2. Схема разработки компонентов, взаимодействующих с помощью RPC.

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.

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.

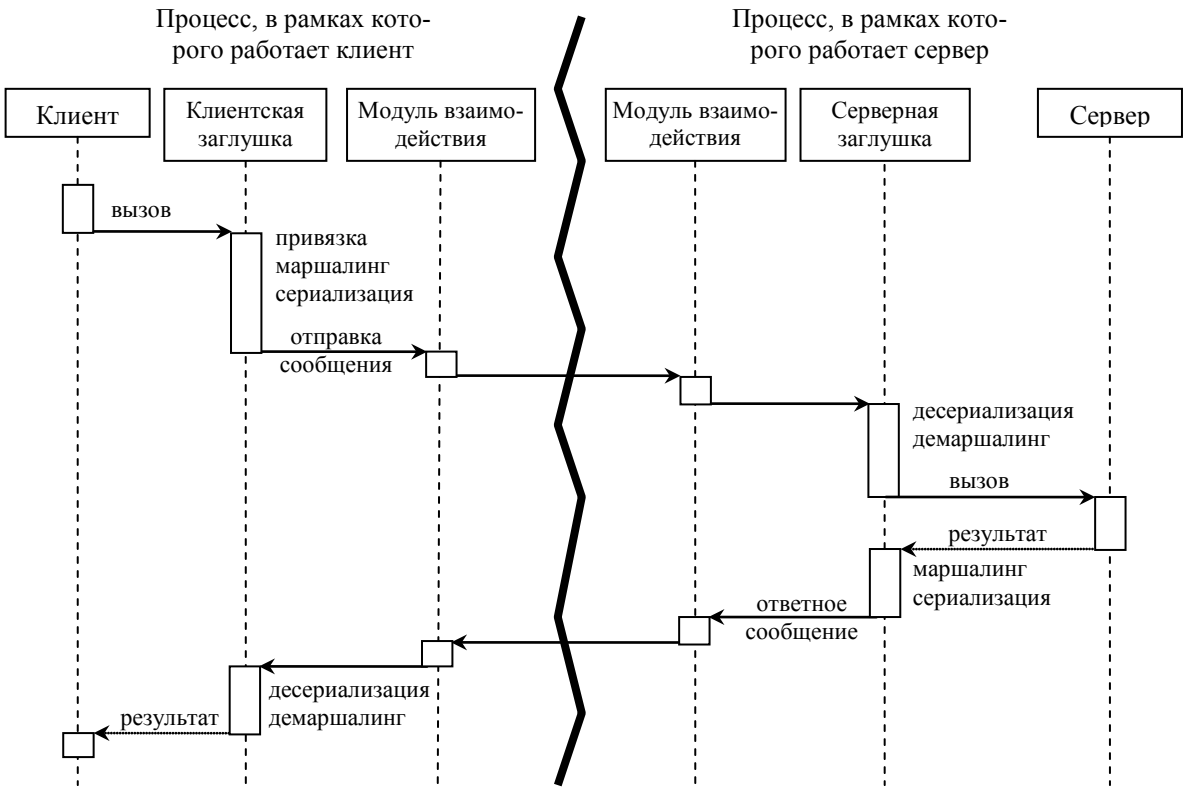


Рисунок 3. Схема реализации удаленного вызова процедуры.

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 его правильно построенной 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, в котором каждый подписчик определяет темы, которые его интересуют.

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, даже если и клиент, и сервер не все время находятся в рабочем состоянии.

В то же время асинхронное взаимодействие гораздо более сложно организовать. 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.

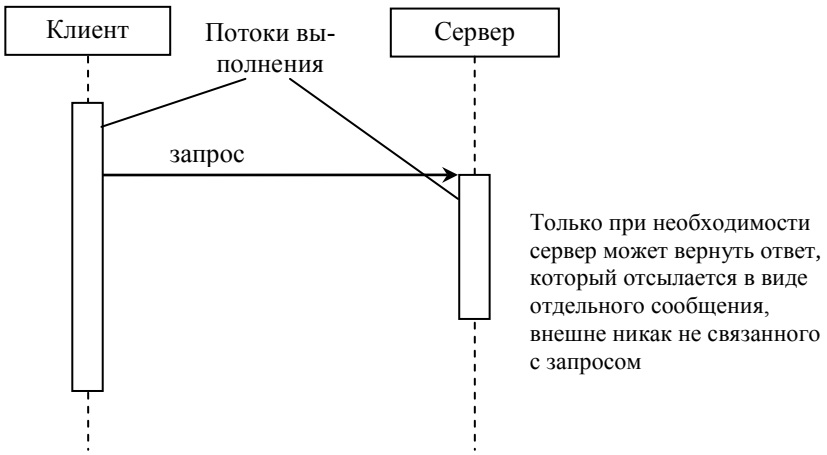


Рисунок 4. Асинхронное взаимодействие.

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.

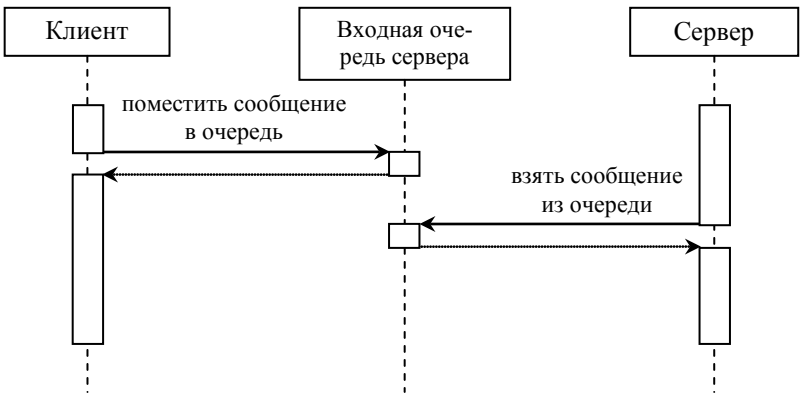


Рисунок 5. Реализация асинхронного взаимодействия при помощи очередей сообщений.

Очереди сообщений могут быть сконфигурированы самыми разными способами. У компонента может иметься 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 различно 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 хозяина. 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 объединением их 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 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; 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 для промежуточного уровня являются *мониторы транзакций (transaction monitors)*, обеспечивающие выполнение удаленных вызовов процедур с поддержкой транзакций. Такие транзакции часто называют ***распределенными.***

Для организации 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.

Рисунок 6. Схема реализации поддержки распределенных транзакций.

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 оповещения об изменениях свойств компонента, которые некоторые из компонентов-подписчиков, оповещаемых об этих изменениях, 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, оповестив об этом остальных подписчиков. Если же все согласны, то после внесения изменений о них, как об уже сделанных, оповещают с помощью метода propertyChange() интерфейса java.beans.PropertyChangeListener.

Обеспечение целостности при конкурентном доступе

Одним из основных понятий является понятие целостности данных, которое включает в себя обеспечение непротиворечивости, выверенности и корректности данных. Классические механизмы обеспечения целостности по данным реализованы в реляционных БД и включают в себя пункты:

- каждый объект в БД (строка таблицы) должен быть уникально идентифицирован;
- значения атрибутов объектов должны принадлежать соответствующей области возможных значений;
- ссылки должны соответствовать корректным объектам.

Кроме того, на уровне логики приложения могут быть наложены дополнительные ограничения на данные. Для обеспечения целостности данных используется такие понятия как локальные и распределенные транзакции. Ниже схематически представлен механизм двухфазной фиксации, используемый для управления распределенными транзакциями.

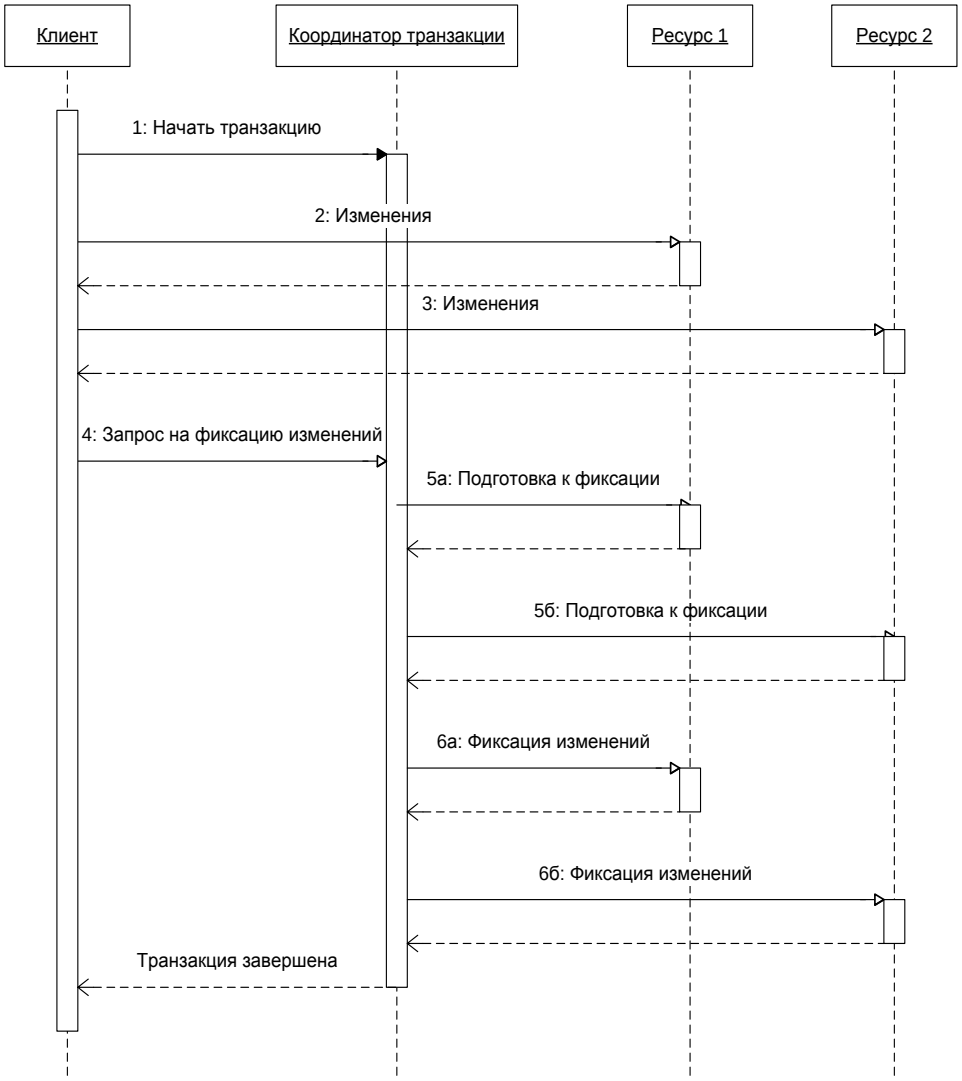


Рисунок 7. Механизм двухфазной фиксации.

Механизм двухфазной фиксации включает в себя следующих участников.

- Клиент – это бизнес-сервис, бизнес-процесс или любая другая программа, которой требуется взаимодействие с распределенными ресурсами.
- Координатор транзакции – это управляющая программа, которая координирует выполнение транзакции (её начало и завершение транзакции процедурой фиксации или отката) между клиентом и ресурсами. Основным стандартом, определяющим работу координатора транзакции, является стандарт X/Open standard for Distributed Transaction Processing (X/Open DTP). Наряду с другими протоколами и программными интерфейсами этот стандарт определяет так называемый XA-интерфейс, который координатор транзакции использует для взаимодействия с ресурсом, например, для выполнения команд “prepare” или “commit”. Координаторы транзакций являются частью продукта класса монитор транзакций, среди которых наиболее известными являются CICS, IMS, Encina (IBM), and Tuxedo (BEA), JTS (Java Transaction Service), MTS (Microsoft Transaction Server).
- Ресурсы – это программные модули, к которым клиентом осуществляется доступ на изменение данных. Все основные технологические платформы (СУБД, системы управ-

ления очередями и т.д.), как правило, реализовывают интерфейс XA для своих продуктов, что позволяет их использовать в рамках распределенной транзакции.

На рисунке иллюстрируются следующие основные шаги по выполнению распределенной транзакции включающей в себя два ресурса.

Клиент отсылает запрос на начало транзакции и регистрирует ресурсы, которые будут в ней участвовать.

1. Клиент выполняет изменения в ресурсе 1.
2. Клиент выполняет изменения в ресурсе 2.
3. Клиент отсылает запрос на фиксацию изменений.
4. Координатор формирует запрос ресурсам на переход в фазу подготовки к фиксации изменений. Ресурсы возвращают в качестве результата свое решение о том можно ли производить фиксацию изменений или нет.
5. На основе «голосования» в зависимости от того все ли ресурсы готовы выполнить фиксацию или есть хотя бы один из ресурсов, требующий выполнение отката изменений, то принимается решение и формируется команда всем ресурсам на выполнение фиксации или отката соответственно. Все ресурсы возвращают подтверждения успешного выполнения команды.

В результате выполнения по описанной схеме может произойти ситуация, когда целостность системы подтвердить невозможно, например, была дана команда на фиксацию изменений всем ресурсам, а какой-то ресурс не подтвердил её выполнение. В этом случае, автоматическими средствами данную ситуацию разрешить невозможно, что приводит к возникновению ошибки выполнения транзакции, которая может быть разрешена только с участием администраторов. Механизм двухфазной фиксации – надежная схема обеспечения целостности ресурсов при распределенных транзакциях. Все блокировки ресурсов и настройка необходимых уровней изоляции выполняется, как правило, на конфигурационном уровне и, таким образом, прозрачна для разработчика. Однако, из-за высоких накладных расходов по хранилищу и по производительности этот механизм нельзя использовать при длительных транзакциях. Таким образом, рекомендовать использование двухфазной фиксации можно только при разработке бизнес-сервисов, при реализации очень быстрых и автоматизированных распределенных взаимодействий.

Идемпотентность операций

При разработке бизнес-сервисов, включающих в себя взаимодействие с удаленными ресурсами, возникает задача обработки ошибок. Отличие от случая разработки локальных приложений состоит в том, что часто при возникновении ошибки клиент не может определить - ошибка на удаленном ресурсе возникла до или после обработки этим ресурсом запроса клиента. Например, пусть клиент вызывает операцию «Создать документ» бизнес-сервиса «Управление документами», который в свою очередь выполняет запрос на создание документа системе LanDocs. Если при этом бизнес-сервис неожиданно завершается, не успев отослать ответ клиенту (или возникает обрыв связи), то клиент не знает выполнено ли создание документа в системе LanDocs или нет.

Данный пример проиллюстрирован на рисунках ниже.

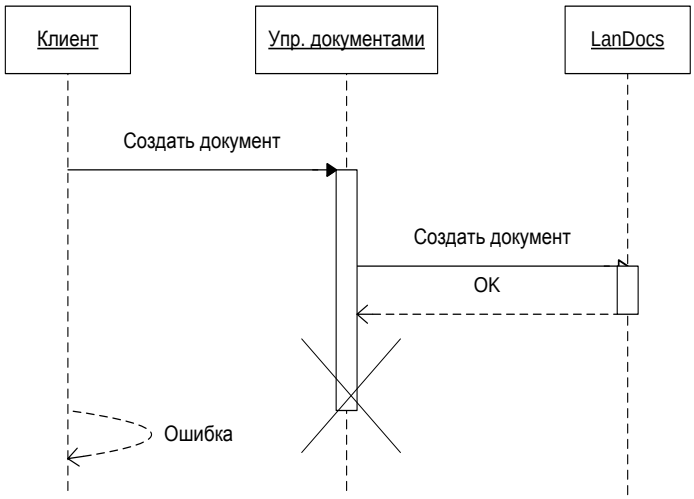


Рисунок 8. Операция успешно завершена, но результат операции клиентом не получен

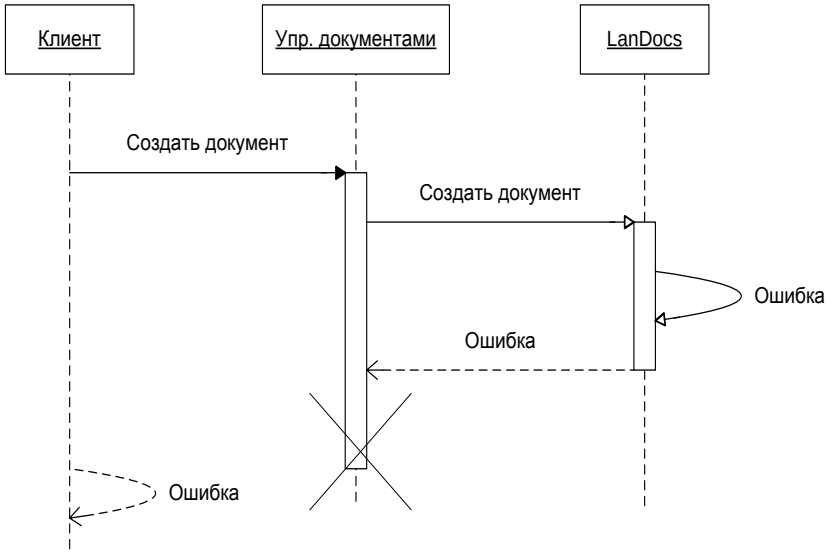


Рисунок 9. Операция завершена с ошибкой, но результат операции клиентом не получен

Таким образом, при получении ошибки нельзя однозначно определить можно ли повторно попробовать выполнить операцию или необходимо выполнить компенсирующие действия по откату последствий ошибочной операции. Такого рода проблемы возникают не со всеми типами операций: необходимо различать операции с семантикой «установить значение» и операции с семантикой «создать, добавить, инкрементировать значение». Как правило, операции с семантикой «установить значение» можно без последствий перезапустить, т.к. в самом плохом случае мы просто перезапишем ранее установленное значение. При перезапуске операций с семантикой «создать, добавить, инкрементировать значение» могут возникнуть проблемы, например, в примере с созданием документа если перезапустить операцию «Создать документ», то в итоге могут быть созданы лишние документы.

Операция называется идемпотентной, если при её выполнении сколько угодно раз (при использовании одного и того же набора параметров) результат и состояние используемых ею ресурсов те же, как если бы она была бы выполнена только один раз.

Оглавление

Таким образом, выше было показано, что все операции с семантикой «установить значение» являются идемпотентными. Остальные операции могут быть, а могут и не быть идемпотентными. Некоторые операции, не будучи формально идемпотентными, могут к ним сводиться достаточно простыми действиями. Например, операция создания патча изменений в системе Norma формально не является идемпотентной, т.к. при повторении операции будут созданы два разных патча с одинаковым содержимым. Однако, т.к. система Norma поддерживает отслеживание и дедуплицирование справочников, то данная операция, возможно, может считаться похожей по своим последствиям на идемпотентную операцию.

Неидемпотентные операции могут быть доработаны таким образом, чтобы они стали идемпотентными. Решение состоит в следующем:

- контракт бизнес-сервиса должен требовать, чтобы операция включала в качестве параметра (т.е. в схеме входящих сообщений) идентификатор работы (английский термин – Unit of Work, UOW ID);
- контракт не может предполагать уникальность UOW ID (т.к.это нельзя гарантировать с достаточной степенью уверенности);
- идентификатор работы UOW ID будет представлять собой работу, которая должна быть выполнена только однажды;
- бизнес-сервис будет использовать UOW ID для определения того была ли работа уже выполнена или еще нет;

Существует три варианта обработки запросов с UOW ID, по которым соответствующая им работа уже была выполнена.

1. Возвращать клиенту закэшированный результат работы.

Этот вариант предполагает кэшированием сервисом всех выполненных запросов до отсылки результатов клиентам. При использовании этого варианта могут возникнуть дополнительные вопросы:

- что если текущее значение отличается от закэшированного?
- что если результатом была ошибка?
- что если клиент повторно использует UOW ID для разных запросов?

2. Выполнить операцию еще раз. Этот вариант, скорее всего, применим для операций чтения, и может быть не применим для других операций.
3. Сгенерировать ошибку.

При использовании этого подхода может возникнуть ситуация, когда клиент не получил ответа от сервиса и просто еще раз вызвал ту же самую операцию с тем же самым UOW ID. Как в этом случае он получит ответ от сервиса?

Обеспечение целостности при конкурентном доступе

Обеспечение целостности бизнес-процессов, которые затрагивают много функциональных систем, является одной из самых сложных задач при разработке и развитии инте-

грационного решения. Поэтому при реализации бизнес-процессов используют понятие бизнес-транзакции, которое характеризуются следующими свойствами:

- используют несколько распределенных ресурсов (например, баз данных, функциональных систем и т.д.);
- взаимодействие с ресурсами осуществляется в удаленном режиме по менее надежным каналам связи по сравнению с доступом к локальным ресурсам;
- функционируют длительное время;
- доступ к ресурсам в рамках одного бизнес-процесса может осуществляться через длительные промежутки времени (что может быть обусловлено как неготовностью каналов связи, так и особенностью бизнес-логики процесса).

Ниже перечислены основные варианты обеспечения целостности бизнес-транзакций:

- механизм оптимистической блокировки;
- механизм пессимистической блокировки.

Далее представлены механизмы оптимистической и пессимистической блокировки, адаптированные для применения в длительных транзакциях. Эти механизмы не так просты для использования, т.к. требуют внесения логики по координации транзакции в реализацию приложения.

Ниже схематично представлен механизм использования оптимистической блокировки ресурса.

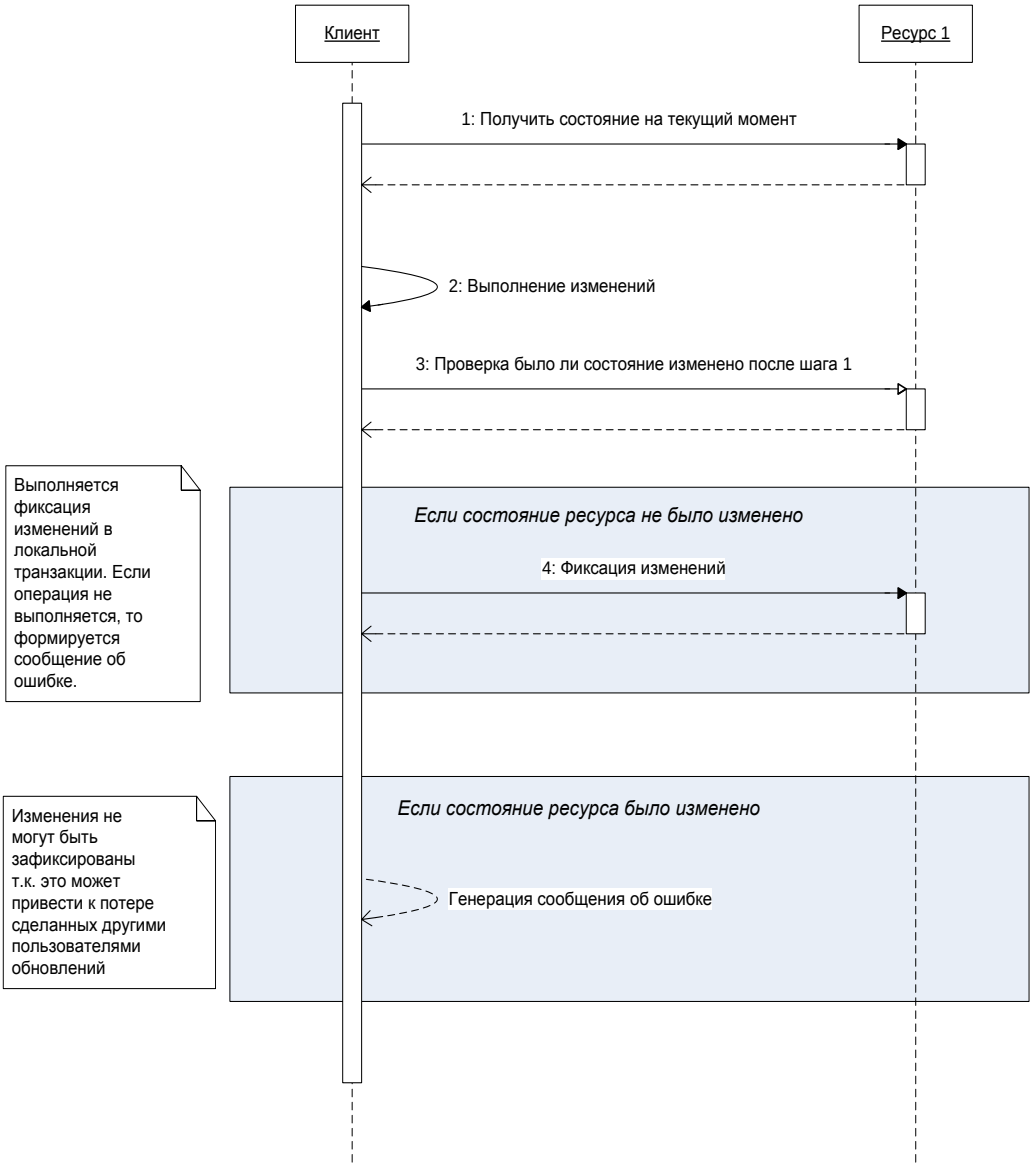


Рисунок 10. Механизм оптимистической блокировки

На рисунке иллюстрируются следующие основные шаги по выполнению транзакции на основе оптимистической блокировки, включающей в себя один ресурс.

1. Клиент читает состояние необходимых ему для работы в рамках транзакции информационных объектов ресурса 1.
2. Клиент выполняет необходимые изменения.
3. Клиент выполняет проверку, было ли с шага 1 изменено состояние (например, другим клиентским приложением) затронутых в рамках транзакции информационных объектов ресурса 1.
4. Если состояние всех информационных объектов не было изменено, то выполняется фиксация изменений информационных объектов ресурса 1. Фиксация изменений производится с помощью локальной транзакции и, если при выполнении этой операции возникла ошибка, то вся транзакция завершается с ошибкой.
5. В противном случае, генерируется ошибка о невозможности завершить транзакцию.

Для реализации шага 4 может в зависимости от ситуации использоваться:

- подход, основанных на сравнении состояний;
- подход, основанный на сравнении версий объектов;
- подход, основанный на сравнении временных меток модификации объектов.

Механизм оптимистической блокировки применяется в том случае, если вероятность конфликта из-за конкурентного доступа к одним и тем же информационным объектам разными транзакциями не велика. В этом случае, выгода от использования этого подхода заключается в том, что он применим для использования в длительных бизнес-транзакциях, которые, например, могут включать в себя взаимодействия с пользователем длящиеся потенциально долгое время.

На рис.1 схематично представлен механизм использования пессимистичной блокировки ресурса.

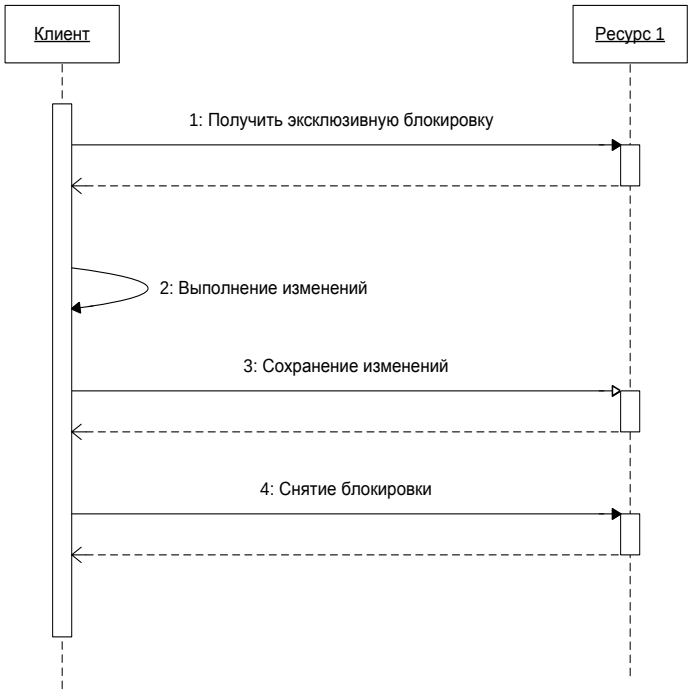


Рисунок 11. Механизм пессимистической блокировки

На рисунке иллюстрируются следующие основные шаги по выполнению транзакции на основе пессимистической блокировки, включающей в себя один ресурс.

1. Клиент запрашивает эксклюзивную блокировку по необходимым ему для работы в рамках транзакции информационным объектам ресурса 1. Если блокировка получена быть не может, то транзакция завершается с ошибкой.
2. Клиент выполняет необходимые изменения.
3. Клиент сохраняет выполненные изменения.
4. Клиент снимает блокировку.

Для реализации шага 1 в информационный объект может быть внесен специальный атрибут LOCKED BY, в котором указывается идентификатор пользователя или транзакции, которая получил блокировку над этим объектом.

Механизм пессимистической блокировки применяется в длительных транзакциях в том случае, если вероятность конфликта из-за конкурентного доступа к одним и тем же информационным объектам разными транзакциями велика. В этом случае, использование оптимистической блокировки не эффективно из-за частных откатов транзакций. Недостаток этого механизма в том, что, как правило, он требует обеспечения дополнительной инфраструктуры по управлению блокировками – высокоуровневой самостоятельной системы, которая позволяла бы назначать, переназначать, снимать блокировки, решать конфликты, которые возникают, если блокировка не снята транзакцией в течение длительного времени и т.д.

При реализации распределенной бизнес-транзакции механизмы оптимистической и пессимистической блокировки могут использоваться совместно (рис. 12).

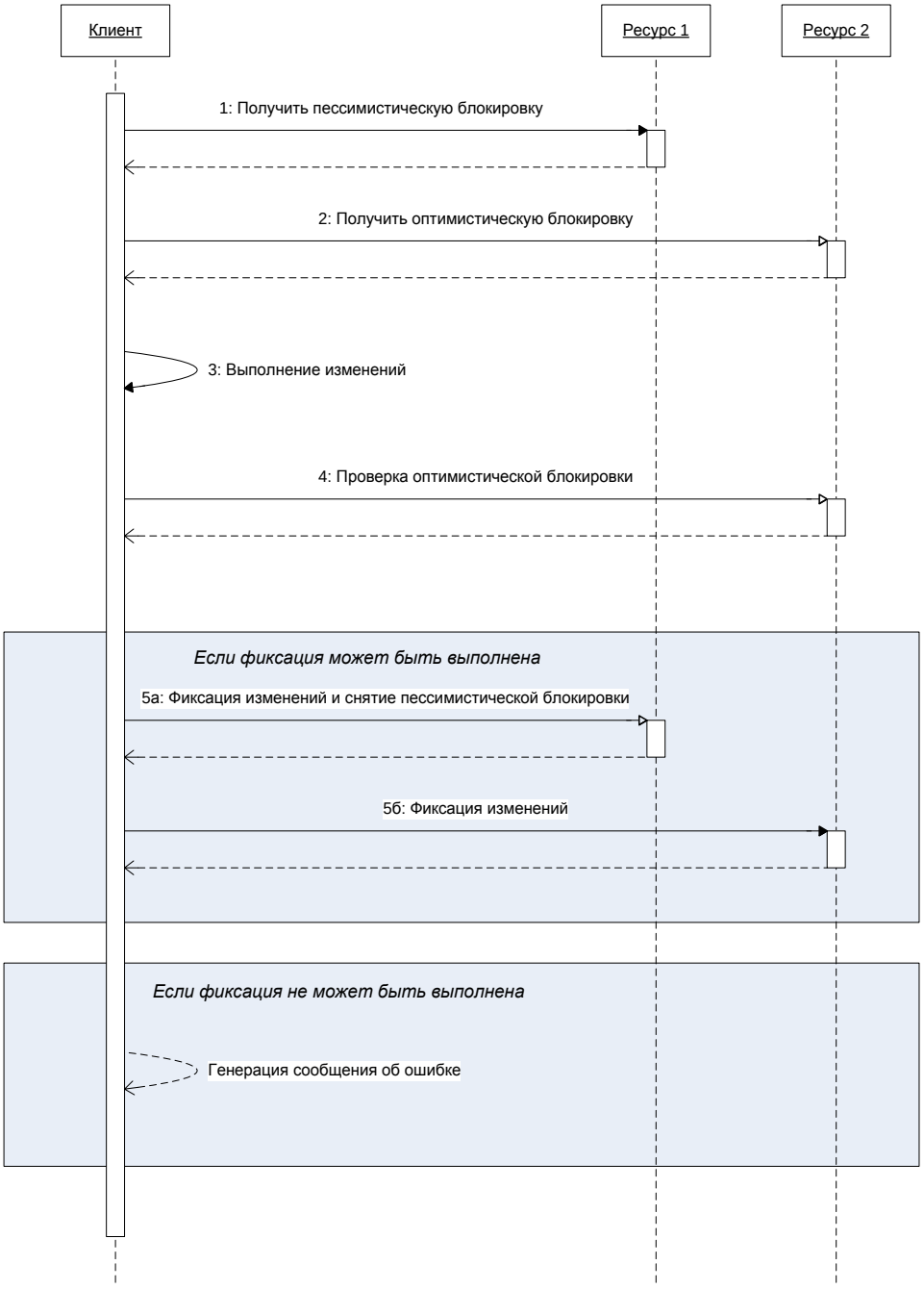


Рисунок 12. Совместное использование оптимистических и пессимистических блокировок в распределенной транзакции

Паттерны проектирования

"Каждый паттерн описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким образом, что этим ключом можно пользоваться при решении самых разнообразных задач".

Christopher Alexander [9]

Паттерн проектирования представляет собой именованное описание проблемы и ее решения, кроме того, содержит рекомендации по применению в различных ситуациях, описание достоинств и недостатков.

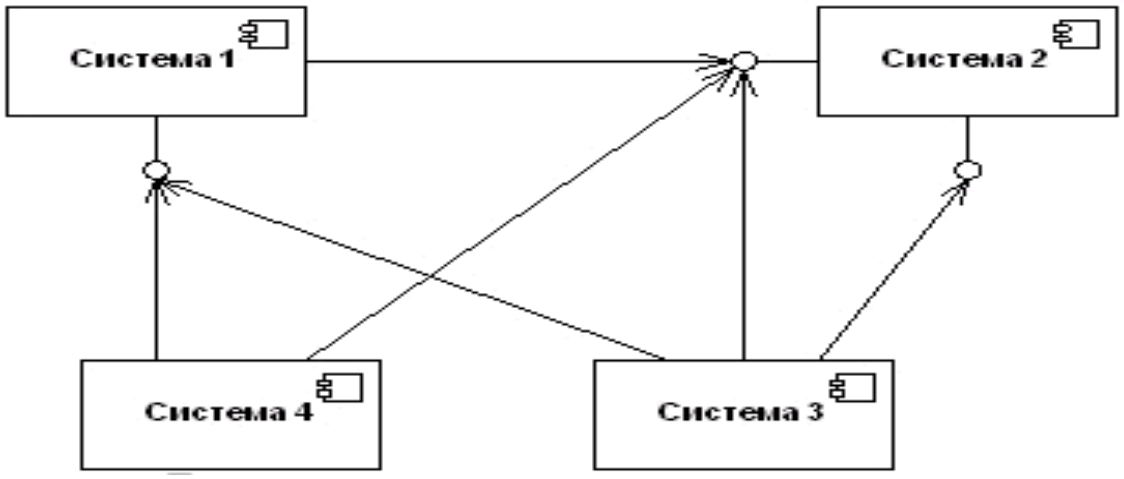
Паттерны интеграции корпоративных информационных систем

Паттерны интеграции информационных систем представляют собой верхний уровень классификации паттернов проектирования. Аналогично паттернам более низких уровней классификации, среди паттернов интеграции выделена группа структурных паттернов. Структурные паттерны описывают основные компоненты единой интегрированной метасистемы. В свою очередь, для описания взаимодействия отдельных корпоративных систем, включенных в интегрированную метасистему, организована группа паттернов, объединенных в соответствии с тем или иным методом интеграции,. Далее, интеграция корпоративных информационных систем подразумевает тем или иным способом организованный обмен данными между системами. Следует отметить, что в отличие от паттернов проектирования классов/обьектов и архитектурных системных паттернов, отнесение отдельного паттерна интеграции к тому или иному виду является менее условным.

Структурные паттерны интеграции

Взаимодействие "точка - точка"

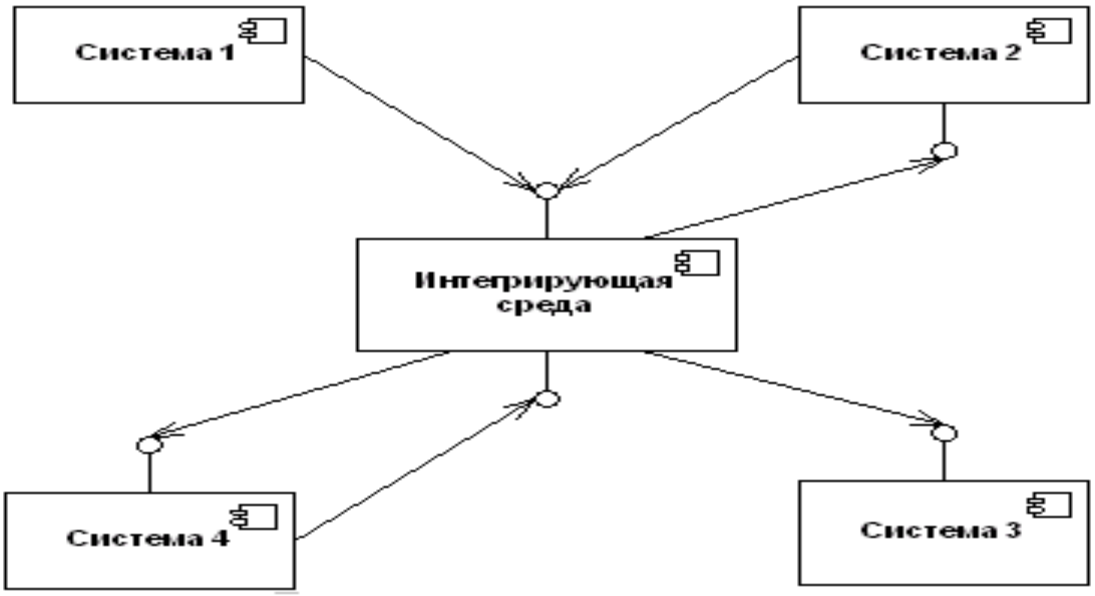
Описание. *У* одной из систем есть интерфейс для доступа к ней активной системы. Данный паттерн применяется, в основном, при стихийной интеграции систем.



Недостаток. Данный метод взаимодействия соответствует требованиям активной системы, но непригоден для использования другой системой в качестве активной.

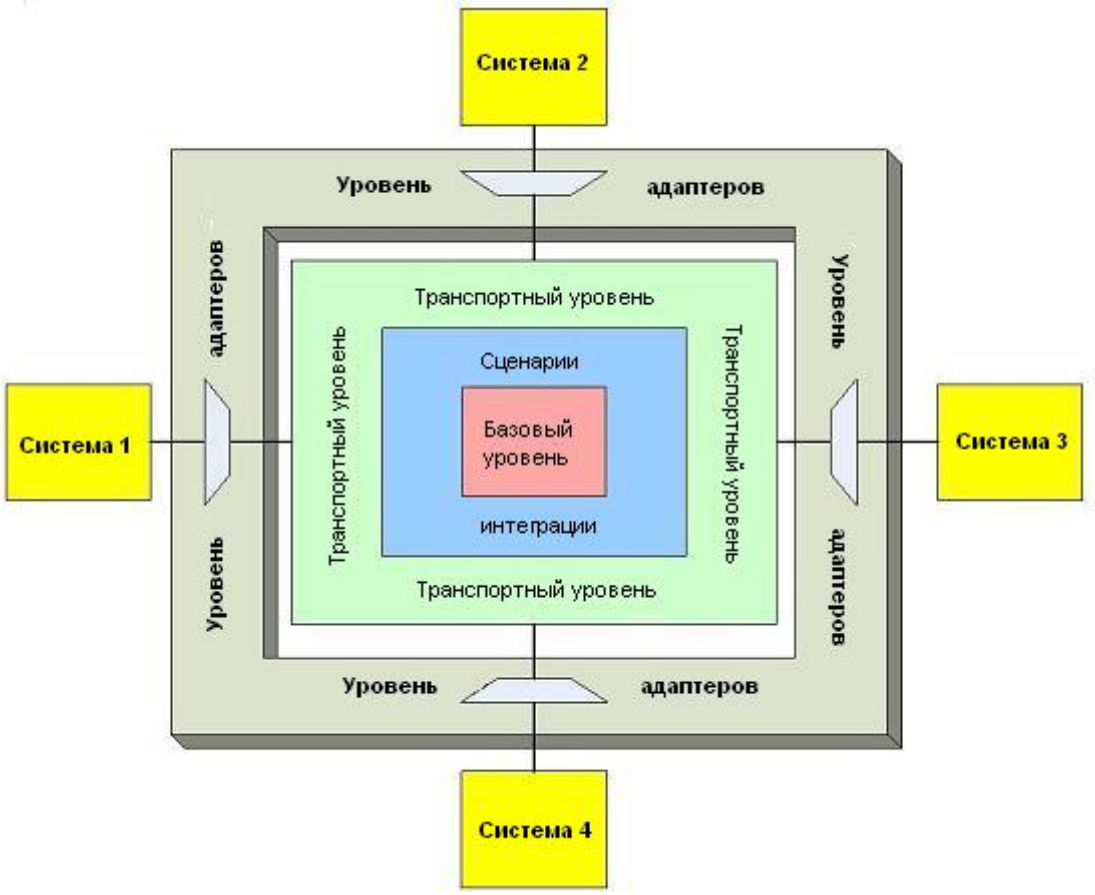
Взаимодействие "звезда" (интегрирующая среда)

Данный способ взаимодействия характеризуется наличием центрального компонента (интегрирующей среды), управляющего взаимодействием подсистем в рамках информационной системы в целом.



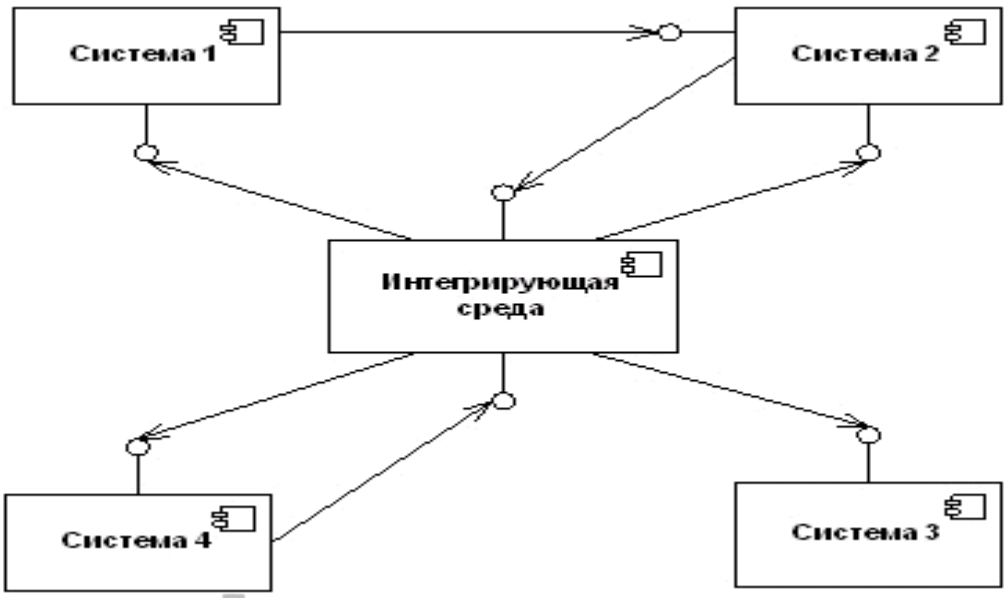
Интегрирующая среда имеет универсальный интерфейс для доступа активных систем. Интегрирующая среда может использовать интерфейсы пассивных систем. Интегрирующая система включает в себя реализацию основных уровней интегрирующей среды: - базовый

уровень интегрирующей среды (представляет собой ядро интегрирующей среды. Содержит платформу для исполнения сценариев транзакции, базовый функционал по взаимодействию приложений, службы протоколирования и мониторинга состояния интегрирующей среды); -уровень сценариев интеграции (графическая схема обмена сообщениями между системами, алгоритмы преобразования и маршрутизации этих сообщений); -транспортный уровень интегрирующей среды (физическая доставка сообщений между компонентами); -уровень адаптеров компонентов (взаимодействие с системой посредством ее API, генерация сообщений, передача сообщений базовому уровню посредством транспортного).



Смешанный способ взаимодействия

В данном способе совмещены оба предыдущих подхода к взаимодействию систем. При этом интерфейсы частично могут использоваться непосредственно напрямую в обход интегрирующей среды. Указанный способ сочетает в себе преимущества централизации управления процессами взаимодействия систем, унификации интерфейсов, а также возможность использовать прямые интерфейсы между системами. Необходимость использования прямых интерфейсов в обход интегрирующей среды может диктоваться, например, специфическими требованиями безопасности.



Паттерны по методу интеграции

Интеграция систем по данным (data-centric)

Данный поход был исторически первым в решении проблемы интеграции приложений. Этот подход характерен для традиционных систем "клиент-сервер". При интеграции приложений по данным считается, что основным системообразующим фактором при построении информационной системы является интегрированная база данных коллективного доступа. Концепция интеграции в этом подходе состоит в том, что приложения объединяются в систему вокруг интегрированных данных под управлением СУБД. Интегрирующей средой является промышленная СУБД (как правило, реляционная) со стандартным интерфейсом доступа к данным (обычно это доступ на SQL). Все функции прикладной обработки размещаются в клиентских программах.
Недостаток. Необходимость передачи больших объемов данных.

Функционально-центрический (function-centric) подход

При функционально-центрическом подходе основным системообразующим фактором являются сервисы - общеупотребительные прикладные и системные функции коллективного доступа, реализованные в виде серверных программ со стандартным API. В виде сервисов реализуются такие функции, как различного вида прикладная обработка, контроль информационной безопасности, служба единого времени, централизованный файловый доступ и т.п. Все сервисы являются интегрированными в том же смысле, что и интегрированные данные в базе данных коллективного доступа, т.е. реализуемые сервисами функции достоверны, непротиворечивы и общедоступны. Концепция интеграции в данном подходе состоит в том, что приложения объединяются в систему вокруг интегриро-

ванных сервисов со стандартизованным интерфейсом. Интегрирующей средой является сервер приложений или монитор транзакций со стандартным API. При использовании функционально-центрического подхода приложение декомпозируется на три уровня (взаимодействие с пользователем, прикладная обработка, доступ к данным). Общая архитектура системы является трехзвенной: клиентское приложение - функциональные сервисы - сервер базы данных.

Объектно-центрический (object-centric)

Объектно-центрический подход, основанный на стандартах объектного взаимодействия CORBA, COM/DCOM, .NET и пр. и является композицией типов объединения систем по данным и обьектно - центрического. Концепция интеграции в состоит в том, что системы объединяются вокруг общедоступных распределенных объектов со стандартными интерфейсами. Характерными особенностями данного подхода являются: " унифицированный язык спецификации интерфейсов объектов (например IDL); " отделение реализации компонентов от спецификации их интерфейсов; " общий механизм поддержки взаимодействия объектов (брокер объектных запросов, играющий роль "общей шины", поддерживающей взаимодействие объектов). Интегрирующей средой является брокер объектных запросов с интерфейсом в стандарте CORBA или DCOM. Общая архитектура системы формируется на основе распределенных объектов и является n-звенной.

Интеграция на основе единой понятийной модели предметной области

Задача. Требуется интеграция в рамках единой системы разнородных интегрирующих средств (concept-centric). Данная проблема весьма актуальна для любой информационной системы большого масштаба, в которой применяются различные покупные системы со своими серверами приложений и другими видами программного обеспечения промежуточного слоя.

Решение. Средством решения проблемы интеграции второго уровня является разработка ОЯВ компонентов, основанного на единой понятийной модели, описывающей объекты предметной области, их взаимосвязи и поведение. Как правило, *объектный язык взаимодействия (ОЯВ)* является языком сообщений высокого уровня и имеет достаточно простой синтаксис и естественно-языковую лексику на основе бизнес-объектов. Единая понятийная модель представляет собой базу метаданных, хранящую описания интерфейсных бизнес-объектов каждого из компонентов и отношения (связи) между этими объектами. Между интегрируемыми компонентами и их описаниями в базе метаданных должно поддерживаться постоянное соответствие. Хранящиеся в базе метаданных описания и сам язык взаимодействия строятся как независимые от конкретного интегрирующего программного обеспечения. Преобразование сообщений на ОЯВ в вызовы функций той или иной интегрирующей среды обеспечивается дополнительной интегрирующей оболочкой с единым интерфейсом, который предназначен только для обмена сообщениями на ОЯВ. Единицей информационного обмена в рассматриваемом подходе являются сообщения, поэтому целесообразно строить такое программное обеспечение на основе программных продуктов класса МОМ.

Паттерны интеграции по типу обмена данными

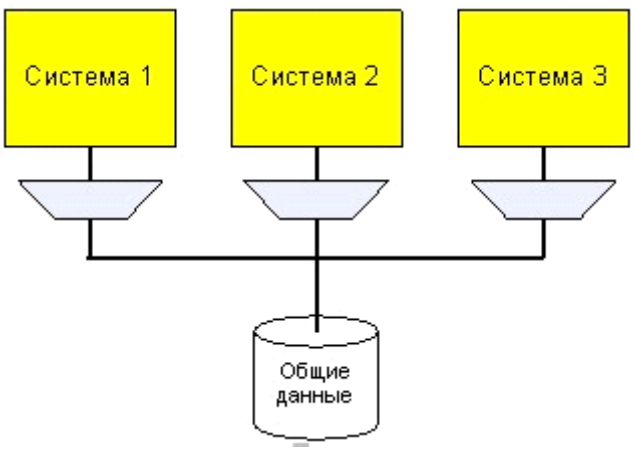
Файловый обмен

Данный тип интеграции основывается на концепции "Точка - Точка, системы экспортируют общие данные в формате пригодном для импорта в другие системы. В последнее время в качестве единого формата файлов обмена все чаще выбирают XML, как наиболее распространенный и поддерживаемый в мире, большинство систем позволяют производить экспорт-импорт данных в формате XML, на рынке программного обеспечения существует большое количество программ, позволяющих в удобной форме создавать так называемые "преобразователи" XML данных на основе технологии XSLT.

Недостаток. Необходим сотрудник, который ответственен за регулярность проведения операций экспорта-импорта, корректности этих операций, а также за соблюдение формата обмена и, возможно за процесс преобразования форматов, т.к. несоответствие форматов экспорта и импорта является частой ситуацией.

Общая база данных

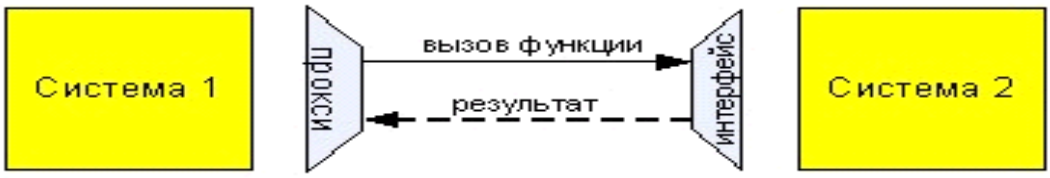
Является реализацией подхода *data-centric*. Данный тип интеграции позволяет получить полностью интегрированную систему приложений, работающую с едиными данными в любой момент времени. Изменения, произведенные в одном из приложений, автоматически отражаются в другом. За корректность данных отвечает многопользовательская СУБД.



Затруднительно интегрировать существующие системы, удобно использовать для вновь создаваемых систем.

Удаленный вызов процедур

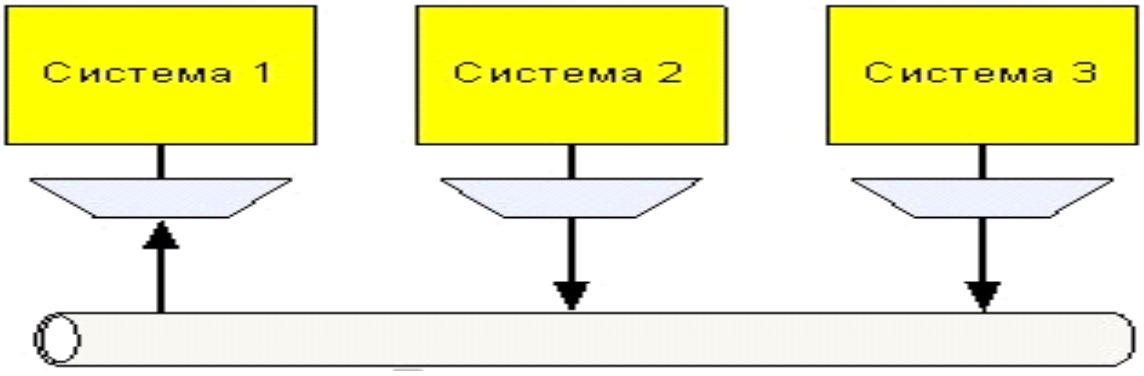
Данный тип интеграции является реализацией *объектно-центрического подхода*. При таком подходе приложения интегрированы на уровне функций. Изменение данных в другой системе происходит также посредством вызова функций.



Недостаток. Каждая из систем самостоятельно заботится о поддержке данных в корректном состоянии, что является довольно сложно задачей

Обмен сообщениями

Данный тип интеграции приложений основан на асинхронном обмене сообщениям посредством шины данных и предназначен для интеграции независимых приложений без или с минимальными доработками существующих систем. Он является реализацией подхода *concept-centric*. При этом за логику интеграции отвечает интеграционная шина в отличие от других типов интеграции, где за логику интеграции отвечала одна из интегрируемых систем. Такой подход позволяет легко интегрировать новые системы, а также изменять логику интеграции, легко приводя ее в соответствие с бизнес логикой процесса.



ТЕХНОЛОГИИ ПОСТРОЕНИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ

DCOM

Distributed Component Object Model (DCOM) - программная архитектура, разработанная компанией Microsoft для распределения приложений между несколькими компьютерами в сети [4]. Программный компонент на одной из машин может использовать DCOM для передачи сообщения (его называют удаленным вызовом процедуры) к компоненту на другой машине. DCOM автоматически устанавливает соединение, передает сообщение и возвращает ответ удаленного компонента.

Для того чтобы различные фрагменты сложного приложения могли работать вместе через Internet, необходимо обеспечить между ними надежные и защищенные соединения, а также создать специальную систему, которая направляет программный трафик.

Для решения этой задачи компания Microsoft создала распределенную компонентную объектную модель Distributed Component Object Model (DCOM), которая встраивается в операционные системы Windows NT 4.0 и Windows 98 и выше.

Преимуществом DCOM является, по мнению Карен Буше, аналитика The Standish Group, значительная простота использования. Если программисты пишут свои Windows-приложения с помощью ActiveX (предлагаемого Microsoft способа организации программных компонентов), то операционная система будет автоматически устанавливать необходимые соединения и перенаправлять трафик между компонентами, независимо от того, размещаются ли компоненты на той же машине или нет.

Способность DCOM связывать компоненты позволила Microsoft наделить Windows рядом важных дополнительных возможностей, в частности, реализовать сервер Microsoft Transaction Server, отвечающий за выполнения транзакций баз данных через Internet. Новая же версия COM+ еще больше упростит программирование распределенных приложений, в частности, благодаря таким компонентам, как базы данных, размещаемые в оперативной памяти.

Однако у DCOM есть и ряд недостатков. "На самом деле это решение до сих пор ориентировано исключительно на системы Microsoft", - считает Буше. Изначально DCOM создавалась под Windows. Хорошо известно, что Microsoft заключила соглашение с компанией Software AG, предмет которого - перенос DCOM на другие платформы. Впрочем, по мнению Буше, значение этой работы достаточно ограниченно, поскольку Microsoft уже успела внести ряд существенных изменений в Windows-версию DCOM.

В числе недостатков и то, что архитектура предусматривает использование для поиска компонентов в сети разработанной Microsoft сетевой службы каталогов Active Directory. Но эта служба каталогов появилась только в версии Windows 2000. В более ранних версиях DCOM должна использовать локальные списки компонентов, что совершенно неприемлемо для приложений большего масштаба, нежели рабочая группа, поскольку информация об изменении местонахождения компонента должна вручную заноситься в каждый работающий в сети компьютер.

Можно перечислить следующие достоинства и недостатки DCOM:
Достоинства

1. Независимость от языка
2. Динамический/статический вызов
3. Динамическое нахождение объектов
4. Масштабируемость
5. Открытый стандарт (контроль со стороны TOG)
6. Множественность Windows-программистов

Недостатки

1. Сложность реализации
2. Зависимость от платформы
3. Нет именования через URL
4. Нет проверки безопасности на уровне выполнении ActiveX компонент
5. Отсутствие альтернативных разработчиков.

DCOM является лишь частным решением проблемы распределенных объектных систем. Он хорошо подходит для Microsoft-ориентированных сред. Как только в системе возникает необходимость работать с архитектурой, отличной от Windows, DCOM перестает быть оптимальным решением проблемы. Конечно, вскоре это положение может измениться, так как Microsoft стремится перенести DCOM и на другие платформы. Например, фирмой Software AG уже выпущена версия DCOM для Solaris UNIX и планируется выпуск версий и для других версий UNIX. Но все-таки, на сегодняшний день, DCOM хорош лишь в качестве решения для систем, ориентированных исключительно на продукты Microsoft. Большие нарекания вызывает также отсутствие безопасности при исполнении ActiveX компонент, что может привести к неприятным последствиям.

CORBA

В конце 1980-х и начале 1990-х годов многие ведущие фирмы-разработчики были заняты поиском технологий, которые принесли бы ощутимую пользу на все более изменчивом рынке компьютерных разработок [4]. В качестве такой технологии была определена область распределенных компьютерных систем. Необходимо было разработать единообразную архитектуру, которая позволяла бы осуществлять повторное использование и интеграцию кода, что было особенно важно для разработчиков. Цена за повторное использование кода и интеграцию кода была высока, но ни кто из разработчиков в одиночку не мог воплотить в реальность мечту о широко используемом, языково-независимом стандарте, включающем в себя поддержку сложных многосвязных приложений. Поэтому в мае 1989 была сформирована OMG (Object Managment Group). Как уже отмечалось, сегодня OMG насчитывает более 700 членов (в OMG входят практически все крупнейшие производители ПО, за исключением Microsoft).

Задачей консорциума OMG является определение набора спецификаций, позволяющих строить интероперабельные информационные системы. Спецификация OMG -- The Common Object Request Broker Architecture (CORBA) является индустриальным стан-

дартом, описывающим высокоуровневые средства поддерживания взаимодействия объектов в распределенных гетерогенных средах.

CORBA специфицирует инфраструктуру взаимодействия компонент (объектов) на представительском уровне и уровне приложений модели OSI. Она позволяет рассматривать все приложения в распределенной системе как объекты. Причем объекты могут одновременно играть роль и клиента, и сервера: роль клиента, если объект является инициатором вызова метода у другого объекта; роль сервера, если другой объект вызывает на нем какой-нибудь метод. Объекты-серверы обычно называют "реализацией объектов". Практика показывает, что большинство объектов одновременно исполняют роль и клиентов, и серверов, попеременно вызывая методы на других объектах и отвечая на вызове извне. Используя CORBA, тем самым, имеется возможность строить гораздо более гибкие системы, чем системы клиент-сервер, основанные на двухуровневой и трехуровневой архитектуре.

Dynamic Invocation Interface ***(DII):*** позволяет клиенту находить сервера и вызывать их методы во время работы системы.
IDL Stubs: определяет, каким образом клиент производит вызов сервера.
***ORB** Interface:* общие как для клиента, так и для сервера сервисы.
IDL Skeleton: обеспечивает статические интерфейсы для объектов определенного типа.
Dynamic Skeleton Interface: общие интерфейсы для объектов, независимо от их типа, которые не были определены в IDL .
Skeleton *Object Adapter*: осуществляет коммуникационное взаимодействие между объектом и ORB.
Вот небольшой список достоинств использования технологии CORBA.

1. Платформенная независимость.
2. Языковая независимость.
3. Динамические вызовы.
4. Динамическое обнаружение объектов.

EJB

Enterprise JavaBeans (также часто употребляется в виде аббревиатуры EJB) — спецификация технологии написания и поддержки серверных компонентов, содержащих бизнес-логику. EJB является частью Java EE 9 [12, 13]. Эта технология обычно применяется, когда бизнес-логика требует как минимум один из следующих сервисов, а часто все из них:

- поддержка сохранности данных (persistence); данные должны быть в сохранности даже после остановки программы, чаще всего достигается с помощью использования базы данных;
- поддержка распределённых транзакций;
- поддержка конкурентного изменения данных и многопоточность;
- поддержка событий;
- поддержка именования и каталогов (JNDI);
- безопасность и ограничение доступа к данным;

- поддержка автоматизированной установки на сервер приложений;
- удалённый доступ.

Каждая EJB компонента является набором Java классов со строго регламентированными правилами именования методов (верно для EJB 2.0, в EJB 3.0 за счет использования аннотаций выбор имен — свободный). Бывают компоненты трех основных типов:

- объектные (Entity Bean), перенесены в спецификацию Java Persistence API;
- сессионные (Session Beans), которые бывают:
  - без состояния (stateless);
  - с поддержкой текущего состояния сессии (stateful);
  - один объект на все приложение (singleton), начиная с версии 3.1;
- управляемые сообщениями (Message Driven Beans) — их логика является реакцией на события в системе.

Основная идея, лежавшая в разработке технологии Enterprise JavaBeans - создать такую инфраструктуру для компонент, чтобы они могли бы легко “вставляться” (plug in) и удаляться из серверов, тем самым увеличивая или снижая функциональность сервера.

Технология Enterprise JavaBeans похожа на технологию JavaBeans в том смысле, что она использует ту же самую идею (а именно, создание новой компоненты из уже существующих, готовых и настраиваемых компонент, аналогиично RAD-системам), но во всем остальном Enterprise JavaBeans - совершенно иная технология.

Принципиальные отличия от JavaBeans:

- Технология Enterprise JavaBeans описывается не спецификацией JavaBeans Component Specification, а совсем другим документом - Enterprise JavaBeans Specification.
- Если JavaBeans имеют дело лишь с *клиентскими* компонентами (как правило, это GUI-компоненты, или компоненты, с ними связанные), то EJB описывает каким образом *внутри EJB-системы* взаимодействуют между собой клиенты и серверы, как EJB-системы взаимодействуют с другими системами и какова роль различных компонент этой системы.

ПРИЛОЖЕНИЯ

Приложение 1. СЛОВАРЬ ТЕРМИНОВ

Общие термины

Rational Rose - инструмент для визуального моделирования объектно-ориентированных информационных систем компании Rational Software. Продукт использует UML.

API (Application Programming Interface) - интерфейс программирования прикладных систем.

UML (Unified Modeling Language) - унифицированный язык моделирования обьектно - ориентированных программных систем.

Анализ - исследование обьектов и процессов предметной области, кроме того, данный термин может означать исследование требований к системе, имеющегося функционала системы и пр.

Архитектура системы - организация и структура основных элементов системы, имеющая принципиальное значение для функционирования системы в целом.

Диаграмма - графическое представление набора элементов разрабатываемой системы или предметной области, в вершинах данного представления находятся сущности, а дуги представляют собой отношения этих сущностей. Диаграммы строятся в рамках определенной модели. Например, в рамках UML строятся следующие диаграммы: - прецедентов (описывает функциональное назначение системы), - концептуальных классов (описывает предметную область), - состояний (описывает поведения зависимых от состояний обьектов системы), - деятельности (используется для алгоритмического описания поведения системы) - программных классов, - взаимодействия (описывает взаимодействие обьектов системы).

Проектирование - выработка концептуальных решений, обеспечивающих выполнение основных требований и разработка системной спецификации.

Модель - представление разрабатываемой системы или предметной области в рамках определенного стандарта, например, модель данных системы, выполненная с использованием стандарта IDEF1X.

Модуль - компонент системы (подсистемы), который предоставляет один или несколько сервисов. Модуль может использовать сервисы, поддерживаемые другими модулями. Модуль не может рассматриваться как независимая система.

Подсистема - часть системы, которая выделяется при проектировании архитектуры. Операции выполняемые подсистемой не зависят от сервисов, предоставляемых другими подсистемами, и, кроме того, подсистемы имеют интерфейсы, посредством которых взаимодействуют с другими подсистемами. Подсистемы могут состоять из модулей или представлять собой группу классов.

Предметная область - область знаний или деятельности, характеризующаяся специальной терминологией, используемой экспертами предметной области, и набором бизнес - правил.

Проектирование - выработка концептуальных решений, обеспечивающих выполнение основных требований и разработка системной спецификации.

Принцип разделения обязанностей - разделение различных аспектов функционирования системы, то есть, разделение системы на элементы, соответствующие разным аспектам функционирования и задачам. Например, программные объекты уровня предметной области должны отвечать только за реализацию логики приложения, а взаимодействие с внешними службами должны обеспечивать отдельные группы обьектов.

Система - совокупность взаимодействующих компонентов, работающих совместно для достижения определенных целей.

Событие - происшествие в системе, значимое для обеспечения требуемого функционала. Событие может быть внешним по отношению к системе и внутренним, то есть инициируемым самой системой.

Требования к системе - условия или возможности, которые система должна выполнять или предоставлять, и, кроме того, соглашение между заказчиком системы и ее разработчиком об этих условиях или возможностях.

Паттерн проектирования представляет собой именованное описание проблемы и ее решения, кроме того, содержит рекомендации по применению в различных ситуациях, описание достоинств и недостатков.

Термины паттернов проектирования объектов

Зацепление (cohesion) - мера "сфокусированности" обязанностей класса. Класс с низкой степенью зацепления выполняет много разнородных функций и несвязанных между собой обязанностей. Создавать такие классы нежелательно.

Инкапсуляция - механизм, используемый для сокрытия данных, внутренней структуры и деталей объекта, все взаимодействие обьектов выполняется через интерфейс операций.

Инстанцирование - создание экземпляра класса.

Интерфейс объекта (системы) - совокупность сигнатур всех определенных для объекта (системы) операций.

Класс специфицирует внутренние данные объекта и его представление, а также операции, которые объект может выполнять. Класс в языке UML (программный или концептуальный) -это описание набора элементов, имеющих одинаковые атрибуты, операции и отношения. *Абстрактный класс* делегирует свою реализацию подклассам, его единственным назначением является спецификация интерфейса.

Наследование - это отношение, которое определяет одну сущность в терминах другой. В качестве примера можно привесит создание специализированных подклассов (классов - потомков) на основе более общих суперклассов (классов-предков). При этом атрибуты и операции суперкласса автоматически присваиваются подклассу, и, кроме того, подкласс

может иметь новые операции и атрибуты. Это позволяет избавиться от необходимости создавать подкласс "с нуля".

Объект - экземпляр класса. В процессе создания объекта выделяется память для переменных - атрибутов класса (внутренних данных класс) и с этими данными ассоциируются операции. Объект существует во время выполнения программы, хранит данные и операции для работы с этими данными, при этом данные объекта могут быть изменены только с помощью операций.

Полиморфизм - отношение, при котором различные сущности (связанные отношением наследования) по - разному реагируют на одно и то же сообщение, например, различная реакция подклассов одного класса на одно и то же сообщение.

Принцип разделения обязанностей - разделение различных аспектов функционирования системы, то есть, разделение системы на элементы, соответствующие разным аспектам функционирования и задачам. Например, программные объекты уровня предметной области должны отвечать только за реализацию логики приложения, а взаимодействие с внешними службами должны обеспечивать отдельные группы обьектов.

Связанность (coupling) - зависимость между классами, вызываемая взаимодействием между ними при выполнении определенной задачи. Класс с высокой степенью связанности зависит от множества других классов, что нежелательно.

Сигнатура операции - имя операции, передаваемые параметры и возвращаемые значения.

Системное событие - событие, генерируемое внешним исполнителем.

Термины архитектурных системных паттернов

Реляционная база данных - база данных, построенная на реляционной модели. Информация в реляционной базе данных хранится в виде связанных таблиц, состоящих из столбцов и строк.

Сеанс - долговременный процесс взаимодействия клиента и сервера, обычно начинается с подключения клиента к системе, включает отправку запросов, выполнение одной или нескольких бизнес - транзакций и пр.

СУБД - система управления базами данных, комплекс программных и семантических средств, реализующий поддержку создания баз данных, централизованного управления и организации доступа к ним различных пользователей в условиях принятой технологии обработки данных.

Транзакция - ограниченную последовательность действий с базой данных с явно определенными начальной и завершающими операциями. Следует выделить следующие свойства транзакций: атомарность (в рамках транзакции выполняются все действия, либо не выполняется ни одно), согласованность (системные ресурсы должны пребывать в целостном и непротиворечивом состоянии после проведения транзакции), изолированность (промежуточные результаты транзакции должны быть закрыты для доступа со стороны любой другой транзакции до проведения их фиксации) , устойчивость (результат проведения транзакции не должен быть утрачен). Выделяют *системные транзакции*, то есть группу SQL - команд в совокупности с командами начала и завершения и *бизнес - транзакции*, то

есть совокупность определенных действий, инициируемых пользователем системы. В качестве примера бизнес - транзакции можно привести регистрацию пользователя, выбор счета, ввод требуемой суммы и подтверждение проведенной операции. Выполнение бизнес - транзакции, как правило, охватывает несколько системных транзакций.

Термины паттернов интеграции

Активная система - система, использующая интерфейс другой системы.

Пассивная система - система, предоставляющая интерфейсы для пользования другим системам и не использующая напрямую интерфейсы других систем.

Интегрирующая среда - совокупность программных и организационных составляющих, целью которых является обеспечение взаимодействия систем и образование единой системы. Наличие интегрирующей среды позволяет говорить о целостности единой системы, а не о наборе отдельных приложений.

ОЯВ - общесистемный язык взаимодействия.

EAI (Enterprise Application Integration) - Интеграция корпоративных систем.

IDL (Interface Definition Language) - язык спецификации интерфейсов.

MOM (Message Oriented Middleware) - системное программное обеспечение промежуточного слоя, ориентированное на обмен сообщениями.

XML (eXtensile Markup Language)- расширяемый язык гипертекстовой разметки, используемый в интернете. Язык XML использует структуру тегов и определяет содержание гипертекстового документа. XML позволяет автоматизировать обмен данными, при этом обьем программирования будет незначительным.

XSLT (eXtensible Stylesheet Language for Transformations) - предназначен для преобразования XML документов. С его помощью можно описать правила преобразования, которые позволят преобразовать документ в другую форму (структуру) или формат, например, в текстовый или HTML.

Контрольные вопросы

1. Какие механизмы обеспечения целостности данных существуют?
2. В каком случае применяется пессимистическая блокировка, а в каком случае оптимистическая?
3. Какие существуют подходы к интеграции вновь создаваемых систем?
4. Чем отличается технология Enterprise JavaBeans от технологии JavaBeans?
5. Что такое модель непротиворечивости? Для чего ее применяют?

Приложение 2. Пример структуры развертывания интеграционного решения

Интеграционное решение состоит из следующих компонентов:

- бизнес-процессы – компоненты, которые разрабатываются в среде BEA WebLogic Workshop, развертываются и исполняются в среде интеграционной платформы BEA WebLogic Integration;
- бизнес-сервисы – компоненты, которые разрабатываются с использованием любых технологий (предпочтительно на языке Java с использованием встроенных средств по разработке процессов и трансформаций интеграционной платформы компании BEA) и исполняются в среде интеграционной платформы;
- компоненты доступа - компоненты, которые могут быть разработкой третьих фирм или разрабатываться самостоятельно с использованием любых технологий (предпочтительно на языке Java с использованием встроенных в интеграционную платформу компании BEA средств по разработке Web-сервисов или адаптеров совместимых со спецификацией Java Connector Architecture) и исполняются в среде интеграционной платформы.

Диаграмма развертывания интеграционного решения представлена на рисунке 13.

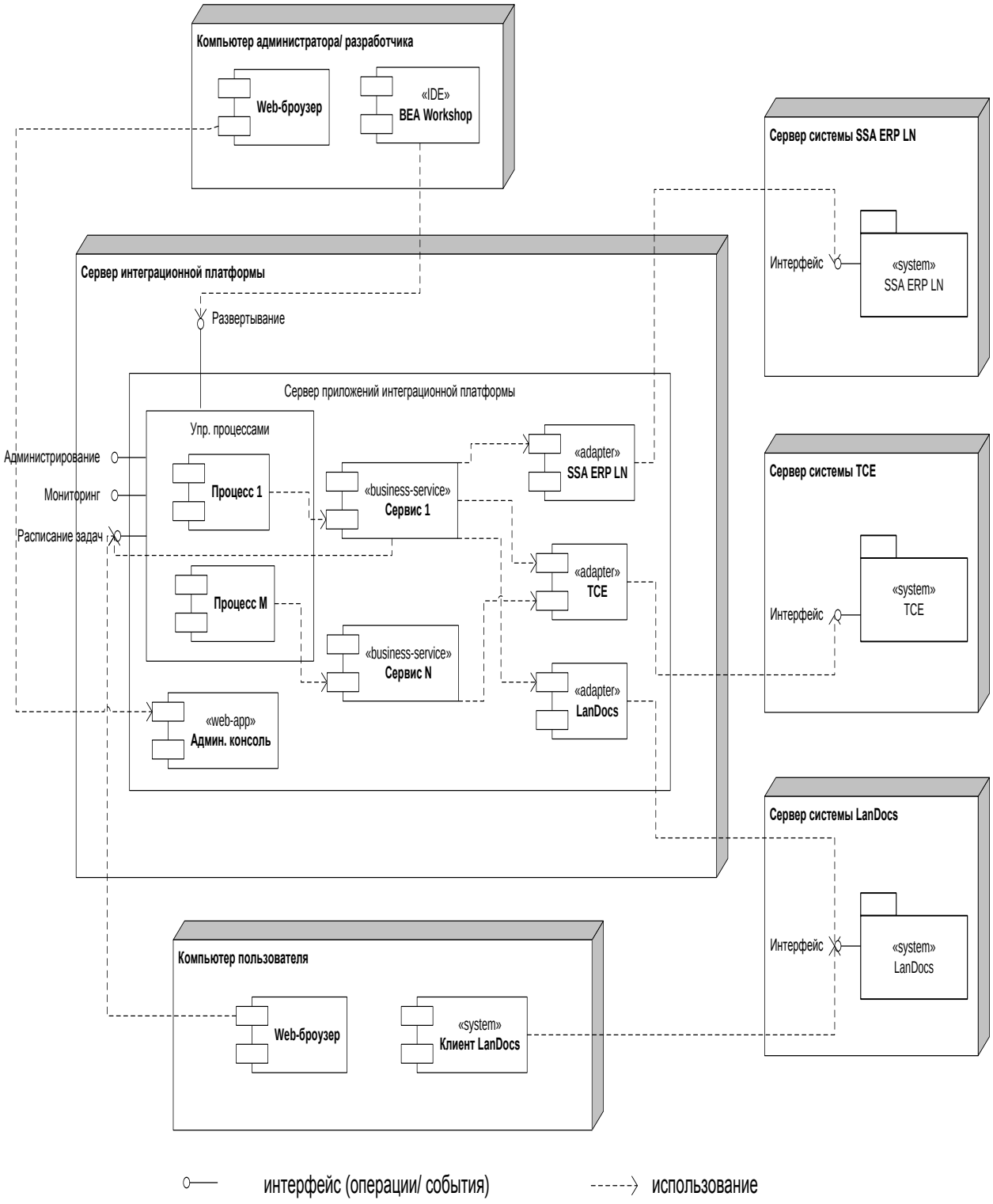


Рис.13. Диаграмма развертывания интеграционного решения

СПИСОК ЛИТЕРАТУРЫ

1. Вишневская Т.И., Романова Т.Н. Технология программирования: Мет. указания к лабораторному практикуму. - Ч. 1. – М: Изд-во МГТУ им. Н.Э. Баумана, 2007, 59с.
2. Вишневская Т.И., Романова Т.Н. Технология программирования: Мет. указания к лабораторному практикуму. - Ч. 2. – М: Изд-во МГТУ им. Н.Э. Баумана, 2010, 46с.
3. Технологии разработки программного обеспечения: Учебник для вузов. / С.А. Орлов. – СПб: Питер, 2004. – 527 с.
4. Харби Дейтел, Пол Дейтел, Дэвид Чофнес. Операционные системы. Распределенные системы, сети, безопасность. – М: ООО ”Издательство Бином”, 2011г. – 704 с.
5. Исаев Г.Н. Проектирование информационных систем. – М.: Издательство “Омега-Л” , 2013. – 424 с.
6. Смит Конни, Уильямс Ллойд. Эффективные решения: практическое руководство по созданию гибкого и масштабируемого программного обеспечения. – М.: Издательский дом «Вильямс», 2003.
7. ГОСТ 19.201-78 (ЕСПД) «Техническое задание. Требования к содержанию и оформлению».
8. Тестирование производительности Web-серверов [Электрон. ресурс] / Д. Намиот, С. Рогов, osp.ru. [1992-2009]. Режим доступа: http://www.osp.ru/os/2002/12/182266/, свободный.
9. K. Alexander et al. Pattern Language. Oxford ,1977.
10. Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб: Питер, 2003.
11. G. Alonso, F. Casati, H. Kuno, V. Machiraju. Web Services. Concepts, Architectures and Applications. Springer-Verlag, 2004.
12. JavaBeans Specification 1.01. [Электрон. ресурс] / Режим доступа : http://java.sun.com/products/javabeans/docs/spec.html.
13. Документация по библиотекам J2EE [Электрон. ресурс] / Режим доступа : http://java.sun.com/j2se/1.5.0/docs/api/index.html.