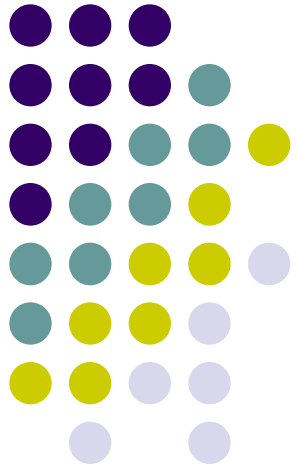
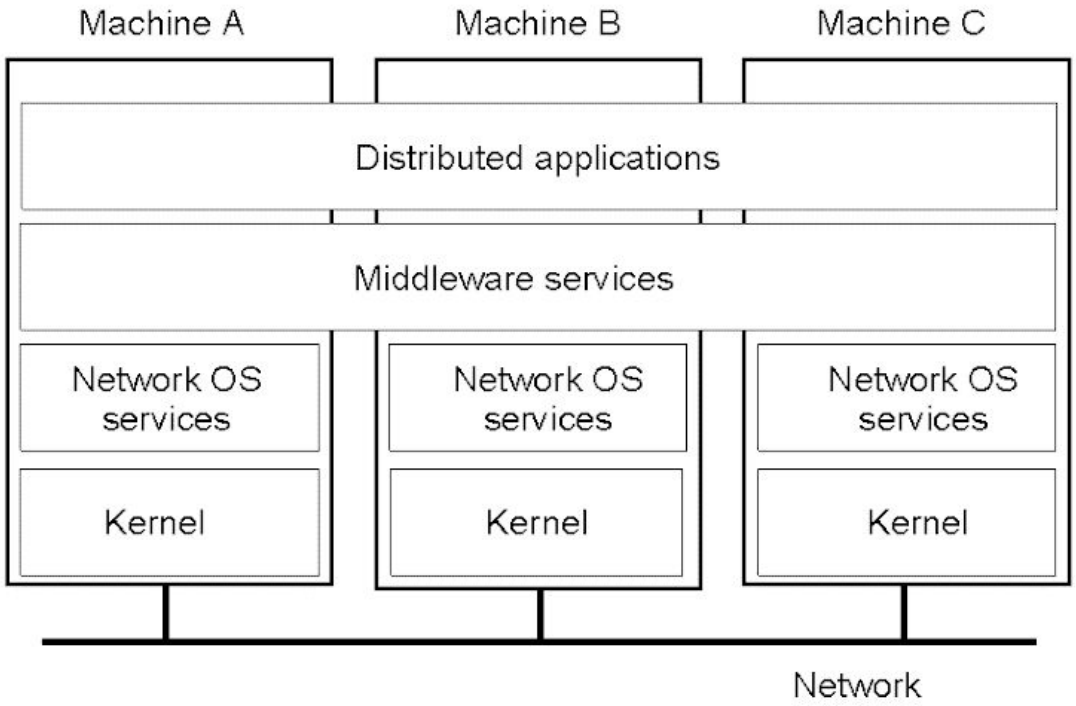
Лекция № 4

Промежуточное программное обеспечение

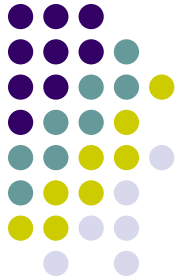


Позиционирование программного обеспечения промежуточного уровня



Общая структура распределенных систем с промежуточным уровнем

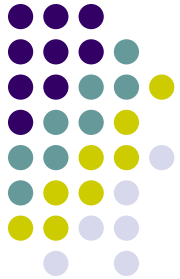
Модели промежуточного уровня



Распределенная файловая система

- на шаг впереди сетевых операционных систем:
  - прозрачность распределения поддерживается ***только*** для стандартных файлов (то есть файлов, предназначенных для хранения данных)
  - процессы, часто должны запускаться исключительно на ***определенных*** машинах
- преимущество:
  - легко масштабируется

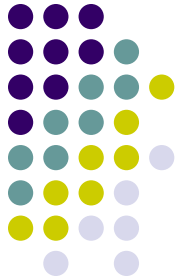
Удаленный вызов процедур (Remote Procedure Calls, RPC)



Процессу разрешается вызывать процедуры, реализация которых находится на удаленной машине (сокрытие сетевого обмена)

- параметры прозрачно передаются на удаленную машину
- процедура выполняется на удаленной машине
- результат возвращается в точку вызова процедуры
- выглядит как локальное исполнение вызванной процедуры
- вызывающий процесс не уведомляется об имевшем место факте сетевого обмена

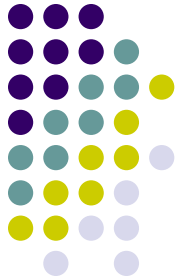
Распределенные объекты (distributed objects)



Основная идея:

- каждый объект реализует интерфейс, который скрывает все внутренние детали объекта от его пользователя
- интерфейс содержит методы, реализуемые объектом
- все, что видит процесс, — это интерфейс

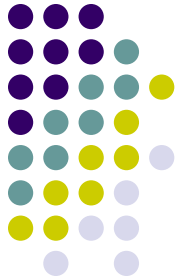
Распределенные объекты (distributed objects)



Реализация распределенных объектов:

- объект размещается на одной из машин
- на множестве других машин открывается доступ к его интерфейсу
- когда процесс вызывает метод, реализация интерфейса на машине с процессом просто преобразует вызов метода в сообщение, пересылаемое объекту
- объект выполняет запрашиваемый метод и отправляет назад результаты
- реализация интерфейса преобразует ответное сообщение в возвращаемое значение, которое передается вызвавшему процессу
- как и в случае с RPC, процесс может оказаться не осведомленным об этом обмене

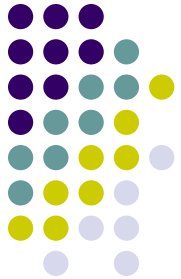
Распределенные документы (distributed documents)



- информация организована в виде документов
- каждый из документов размещен на машине, расположение которой абсолютно прозрачно
- документы содержат ссылки, связывающие текущий документ с другими
- если следовать по ссылке, то документ, с которым связана эта ссылка, будет извлечен из места его хранения и выведен на экран пользователя
- концепция документа не ограничивается исключительно текстовой информацией

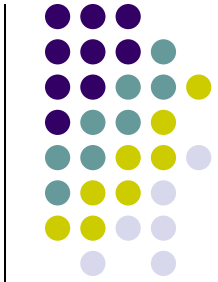
Пример - World Wide Web

Задачи решаемые промежуточным уровнем



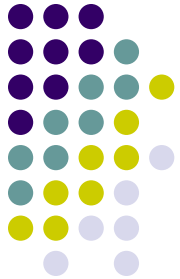
- **Организация связи**. Организация связи и передачи данных между компонентами системы.
- **Идентификация компонентов.** Поддержка идентификации и поиска отдельных ресурсов внутри системы.
- **Синхронизация.** Синхронизация параллельно выполняемых потоков работ.
- **Целостность.** Поддержка целостности данных и непротиворечивости вносимых изменений.
- **Отказоустойчивость.** Организация отказоустойчивой работы.
- **Защита.** Организация защищенности данных и коммуникаций.

Организация связи



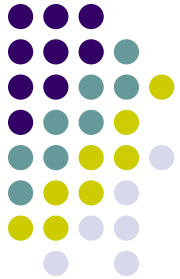
- В распределенных системах происходит обмен данными между компонентами, которые расположены на разных компьютерах и работают совместно для решения общих задач
- Обмен данными происходит при взаимодействии компонентов.
- Все взаимодействия выполняются по протоколу запрос-ответ.
- Взаимодействия могут быть синхронными или асинхронными в зависимости от времени (последовательности) получения ответа.

Синхронное взаимодействие



- **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)**.

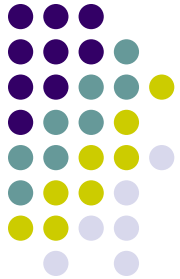
Синхронное взаимодействие



Достоинства:

- достаточно просто организуется
- гораздо проще для понимания
- соответствующий код просто устроен
- часть кода, 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.

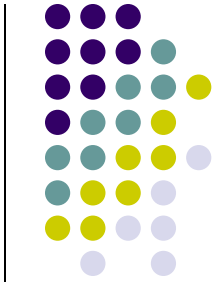
Синхронное взаимодействие



Недостатки:

- ведет к значительным затратам времени на ожидание ответа
- время ожидания ответа на один запрос, можно использовать более продуктивно, выполняя другие запросы, которые не зависят от еще не пришедшего результата
- суммарное падение производительности, связанное с синхронностью взаимодействий, оказывается очень большим из-за достаточно большого числа уровней, через которые проходят практически все взаимодействия в распределенных системах

Организация синхронного взаимодействия



Рассмотрим взаимодействие между элементами программных систем:

инициатор взаимодействия, т.е. компонент, посылающий запрос на обработку, обычно называется **клиентом**,

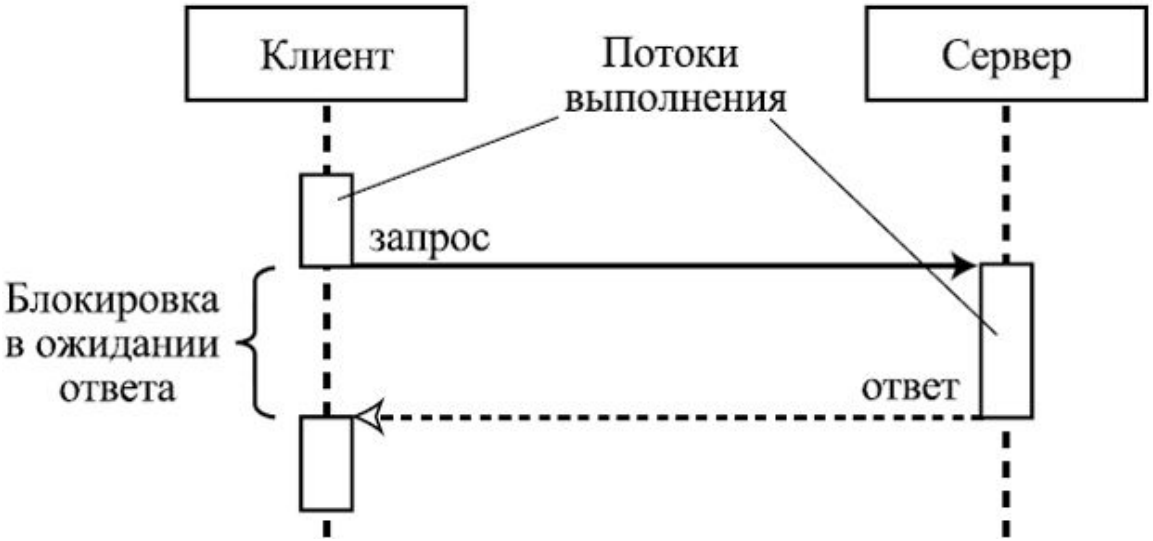
компонент, обрабатывающий запрос — **сервером**

"Клиент" и "сервер" в данном рассмотрении обозначают роли в рамках данного взаимодействия.

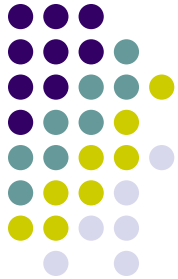
В большинстве случаев один и тот же компонент может выступать в разных ролях — то клиента, то сервера — в различных взаимодействиях.

Лишь в небольшом классе систем роли клиента и сервера закрепляются за компонентами на все время их существования.

Организация синхронного взаимодействия



Клиентская заглушка



Клиентская заглушка:

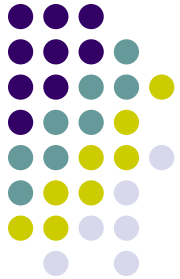
компонент, размещаемый на той же машине, где находится компонент-клиент.

Удаленный вызов метода клиентом реализуется как обычный, локальный вызов определенной функции в клиентской заглушке

При обработке этого вызова клиентская заглушка выполняет следующие действия.

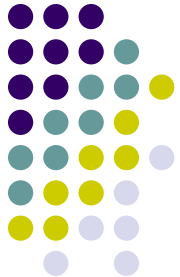
1. Определяется физическое местонахождение в системе объекта-сервера, для которого предназначен данный вызов.
   - Это шаг называется **привязкой (binding)** к серверу.
   - Его результатом является адрес машины, на которую нужно передать вызов.

Клиентская заглушка



1. Вызов метода и его аргументы упаковываются в сообщение в некотором формате, понятном серверной заглушке. Этот шаг называется **маршалингом (marshaling)**.
2. 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.
4. В результате для клиента *удаленный вызов метода* выглядит как обращение к обычной функции.

Серверная заглушка

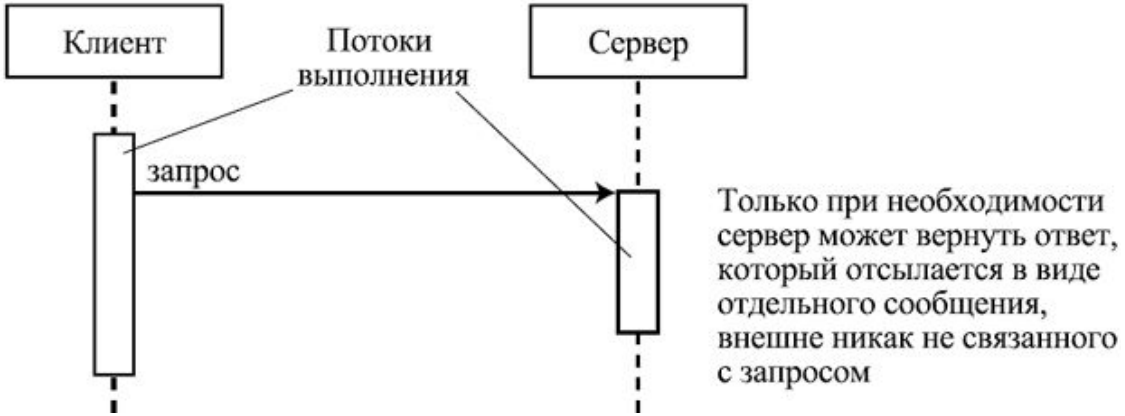


- располагается на той же машине, где находится компонент-сервер.
- выполняет операции, обратные к действиям клиентской заглушки:
  - принимает сообщение, содержащее аргументы вызова
  - распаковывает эти аргументы при помощи **десериализации (deserialization)** и **демаршалинга (unmarshaling)**
  - 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.

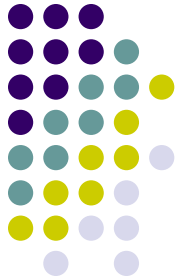
Схема реализации синхронного удаленного вызова процедуры

Организация асинхронного взаимодействия

При асинхронном взаимодействии клиент, отправивший запрос не ожидает ответа, а продолжает выполнять свои задачи



Асинхронное взаимодействие

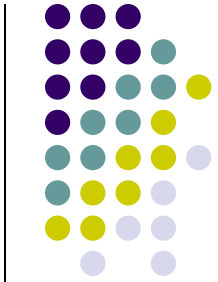


Преимущества:

- позволяет получить более высокую производительность системы за счет использования времени между отправкой запроса и получением ответа на него для выполнения других задач
- 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, которая продолжает работать, даже если и клиент, и сервер не все время находятся в рабочем состоянии.

Асинхронное взаимодействие



Недостатки:

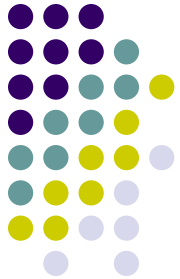
асинхронное взаимодействие более сложно использовать, поскольку:

- нужно писать специфический код для получения и обработки результатов запросов
- системы, основанные на асинхронных взаимодействиях значительно труднее разрабатывать и сопровождать.

Реализация асинхронного взаимодействия 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.

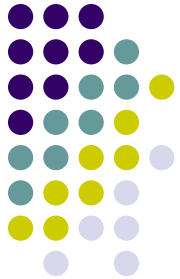
Конфигурация очередей сообщений



Очереди сообщений могут быть сконфигурированы самыми разными способами:

- компонент может иметь одну входную очередь
- компонент может иметь несколько входных очередей для сообщений от разных источников или имеющих разный смысл
- компонент может иметь одну или несколько выходных очередей и может класть сообщения туда а не во входные очереди других компонентов.
- очереди сообщений могут храниться независимо как от тех компонентов, которые кладут туда сообщения, так и от тех, которые забирают их оттуда
- сообщения в очередях могут иметь приоритеты
- очередь может реализовывать различные политики поддержания или изменения приоритетов сообщений в ходе работы

Идентификация компонентов



При взаимодействии распределенных компонентов клиент отправляет запрос серверу.

При этом он должен однозначно идентифицировать сервер, указав его имя и адрес его местонахождения.

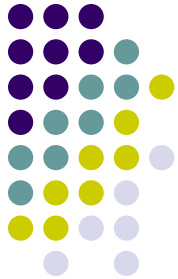
В качестве адреса может выступать:

- IP адрес,
- DNS имя,
- имя очереди,
- электронный адрес почты и т.п.

Возможны два варианта определения адреса:

- статический – клиент обращается по заранее известному адресу указанному еще на этапе его реализации;
- динамический – адрес определяется по имени сервера, через вспомогательные службы.

Определение адреса сервера



Определение статического адреса:

- как правило не вызывает проблем при реализации
- накладывает ограничения на сервер:
  - адрес сервера должен быть известен;
  - сервер не дожжен менять своего местоположения (адрес должен быть постоянен).

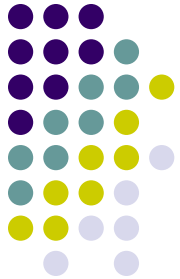
Определение динамического адреса

- более сложено в осуществлении:
- требует реализации дополнительных служб, позволяющих клиентам по имени сервера определять его местоположения

Существуют различные подходы к реализации таких служб, можно выделить следующие распространенные подходы:

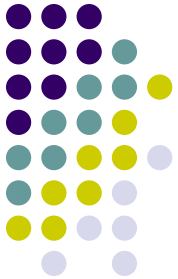
- Централизованная служба наименования;
- Децентрализованная служба наменования.

Синхронизация параллельно выполняемых потоков работ



- **Доступ к ресурсам.** Распределение компонентов по разным компьютерам и возможность их параллельной работы порождает проблему, связанную с общим доступом компонентов к ресурсу. При этом в качестве ресурса могут выступать файлы, данные в БД, сервисы и др.
- **Взаимодействие**. Синхронное взаимодействие (которое требует блокировки работы компонента до получения ответа) нескольких компонентов требует согласования для избегания взаимоблокировки и «зависания» работы.
- **Время**. Распределенные компоненты работающие на разных компьютерах используют локальное системное время, которое часто может быть несинхронизированое. Следствием этого является, то что для разных компонентов понятие «раньше» и «позже» может не совпадать

Мониторы транзакций

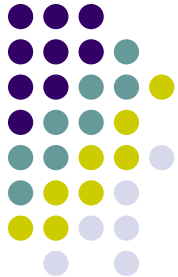


Мониторы *транзакций* **(transaction monitors) -** один из широко распространенных видов программного обеспечения промежуточного уровня:

- обеспечивают выполнение *удаленных вызовов процедур* с поддержкой *транзакций*.

Рассмотрим кратко свойства транзакций.

Транзакции

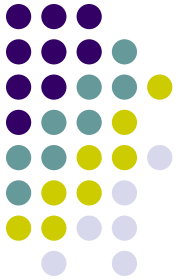


Примером, поясняющим необходимость использования *транзакций*, является перевод денег с одного банковского счета на другой.

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*.

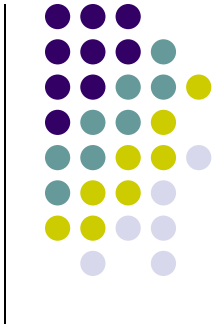
Транзакции:



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.

Транзакции:

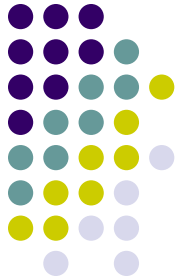


Изолированность (isolation):

- Одновременно происходящие *транзакции* не влияют друг на друга:
  - 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.

Транзакции:



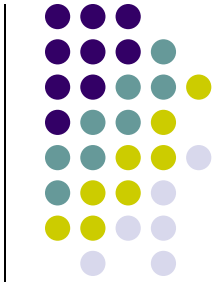
Долговечность (durability):

- 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.

Транзакции:



Плоские тракзакции часто накладывает слишком сильные ограничения на работу системы :

- недоступны промежуточные результаты
- *транзакция* продолжается заметное время (а иногда их выполнение требует нескольких месяцев

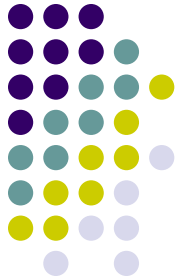
Иногда требуется гораздо более сложное поведение:

уметь выполнять или отменять только часть операций в составе *транзакции*

уметь получить ее промежуточные результаты транзакций процессам, которые в ней не участвуют

Для решения таких задач применяются механизмы, допускающие вложенность *транзакций* друг в друга, длинные *транзакции*, позволяющие получать доступ к своим промежуточным результатам, и пр.

Распределенные транзакции



Транзакция называется распределенной, если участвующие в ней процессы работают на разных машинах.

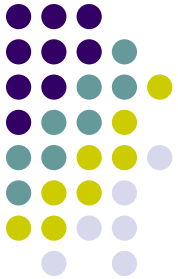
Для организации распределенных *транзакций* необходим **координатор**, который:

- получает информацию обо всех участвующих в *транзакции* действиях
- обеспечивает ее атомарность и изолированность

Обычно *транзакции* реализуются при помощи примитивов, которые:

- позволяют **начать** *транзакцию*,
- **завершить ее успешно (commit)**, с сохранением всех сделанных изменений,
- **откатить** *транзакцию* **(rollback)**, отменив все выполненные в ее рамках действия.

Распределенные транзакции



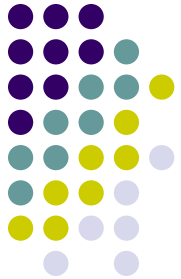
Примитив «начать *транзакцию*»:

- 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*.

Распределенные транзакции

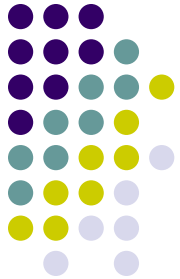


Если один из участников не может выполнить свою операцию, выполняется откат *транзакции*.

При этом координатор рассылает всем зарегистрированным участникам сообщения о необходимости отменить выполненные ими ранее действия.

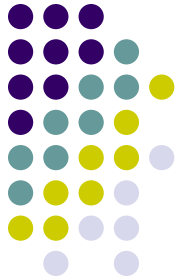
- Если вызывается примитив "завершить *транзакцию*", координатор выполняет некоторый протокол подтверждения, чтобы убедиться, что все участники выполнили свои действия успешно и можно открыть результаты *транзакции* для внешнего мира.

Протокол двухфазного подтверждения (Two-phase Commit Protocol, 2PC)



- протокол, подтверждающий успешность завершения транзакции
- является одним из наиболее распространенных
- имеет следующих алгоритм:
  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.
  3. Иначе — он посылает сообщение об ошибке.

Протокол двухфазного подтверждения (Two-phase Commit Protocol, 2PC)



1. 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*.
2. Каждый участник, получив сообщение о подтверждении *транзакции* в целом, сохраняет локальные изменения, сделанные в рамках *транзакции*.
3. Если же он получил сообщение об отмене *транзакции*, он отменяет локальные изменения.