Распределёнными будем называть системы, которые программные и аппаратные компоненты находятся в компьютерной сети и организуют своё взаимодействие только лишь через обмен сообщениями. При этом не особо существенно, находятся ли эти взаимодействующие устройства в одной комнате или на разных концах света. Основное назначение распределённых систем -- работа с общими ресурсами (под ресурсами можно понимать что угодно, что может быть полезно в распределённой системе).

Из этого определения следует несколько ключевых особенностей такого рода систем.

- **Параллельная работа** является не только нормой, но и необходимостью. Часто производительность всей системы зависит от того, насколько хорошо масштабируется её работа. Координация параллельно выполняемых программ, работающих с общими ресурсами -- задача, которой следует уделять много внимания.
- **Независимые отказы**. Любой компонент системы может отказать, и задача проектировщика системы спланировать работу при любом возможном отказе. В распределённых системах помимо “обычных” программных ошибок добавляются отказы в работе сети. Компьютер может оказаться изолирован от других, но при этом продолжать выполнять свои задачи. Связь может не пропасть совсем, а лишь может снизиться пропускная способность канала, начать теряться часть пакетов или что-то ещё. Любая из компонент системы, а также среда передачи данных может перестать работать независимо от других, и система должна максимально продолжать работу в любых условиях.
- **Отсутствие общих часов**. Прямым следствием взаимодействия программ только лишь через посылку сообщений является отсутствие общего понятия о времени. На разных компьютерах время может быть установлено по-разному и абсолютно точной синхронизации часто бывает добиться невозможно. Поэтому строить взаимодействие можно лишь на относительном порядке событий.

Проектирование подобных систем имеет массу сложностей. Практически все они известны с самого момента появления компьютерных сетей, в 1994 году под названием Fallacies of Distributed Computing был сформулирован список основных заблуждений в предположениях, которые делают разработчики распределённых приложений:

1. Сеть надёжна.
2. Задержка (latency) равна нулю.
3. Пропускная способность бесконечна.
4. Сеть безопасна.
5. Топология сети неизменна.
6. Администрирование сети централизовано.
7. Передача данных “бесплатна”.
8. Сеть однородна.

Каждый из этих пунктов может привести к определённым архитектурным решениям при проектировании системы. Например, если сеть ненадёжна, то инфраструктура приложения должна динамически подстраиваться под изменения в сети. Наличие большой задержки при передаче данных может потребовать работы приложения на основе экстраполяции полученных ранее сообщений. Ограничения или

нестабильность пропускной способности сети заставляет приложение подстраивать свои ожидания от транспортного уровня, возможно сокращая объём передаваемых данных. Наличие более, чем одного центра администрирование сети может потребовать внедрения явных механизмов безопасности. А неоднородность сети заставляет выделять в коде уровни дополнительные абстракции, чтобы приспосабливаться к форматам и способам взаимодействия разных устройств и систем.

Архитектурные элементы распределённых систем

Понять главные строительные блоки распределённых систем помогут следующие четыре вопроса.

- Какие сущности взаимодействуют между собой в распределённой системе?
- Как они взаимодействуют (точнее, в рамках какой парадигмы происходит взаимодействие)?
- Какие (возможно изменяющиеся) роли и ответственности имеют эти сущности в рамках всей архитектуры?
- Как они размещаются на физическую инфраструктуру?

Рассмотрим эти вопросы подробнее.

Виды сущностей

Ответы на первые два вопроса дают проектировщику богатый выбор возможностей. Тут важно рассмотреть две точки зрения.

С системной точки зрения ответ довольно прост, так как распределённые системы состоят из процессов, взаимодействующих в рамках одной из парадигм (будут рассматриваться дальше). В некоторых особых случаях (например, в случае встроенных систем) операционные системы могут не поддерживать абстракцию процесса, поэтому там взаимодействующими сущностями будут сами узлы. Ну и также не стоит забывать про то, что процессы могут содержать в себе ряд потоков исполнения, которые могут ещё взаимодействовать между собой.

С точки зрения программирования, однако, такого уровня детализации бывает недостаточно. На уровне программного кода можно выделить следующие сущности:

- **Объекты**. При применении объектно-ориентированного подхода распределённые системы состоят из набора объектов, представляющих части декомпозированной предметной области. Доступ к объектам осуществляется через их интерфейсы, которые описываются на специальных языках описания интерфейсов (interface definition languages, IDL). Использованию объектов в распределённых системах посвящено много внимания.
- **Компоненты**. Использование связующего ПО, основанного на объектах, может иметь свои недостатки. К ним можно отнести неявные зависимости реализации объектов, чрезмерная низкоуровневость получаемого кода и архитектуры всего решения, необходимость разработчиков решать вопросы безопасности, обработки ошибок, параллельного выполнения и другие типовые проблемы, повторяющиеся из приложения к приложению. Компонентно-ориентированная архитектура помогает бороться с этими сложностями, сохраняя все

преимущества ООП (инкапсуляцию, декомпозицию предметной области, гибкость решений и т.п.). Разница тут в том, что компоненты специфицируют не только свои интерфейсы, но и явные предположения о наличии и интерфейсах других компонент, необходимых им для полноценной работы -- то есть явно специфицируя все контракты и зависимости. Также в компонентно-ориентированном ПО часто автоматизированы процессы сборки и размещения приложений.

- **Веб-сервисы** -- третья важная парадигма разработки распределённых систем. Они продолжают все те же идеи инкапсуляции поведения и доступ к функциональности через интерфейсы. Консорциум W3C определяет веб-сервис следующим образом:

  ```
  a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based message exchanges via Internet-based protocols.
  ```

  Тогда как объекты и компоненты обычно используются для создания приложений в сильно связном окружении, веб-сервисы обычно считаются самостоятельными единицами, которые могут использоваться совместно для достижения каких-то целей, более сложных, чем каждый из них решает в отдельности. Использование веб-сервисов часто выходит за границы компании, позволяя осуществлять интеграцию бизнес-задач различных компаний. При этом как конкретно будут реализовываться эти веб-сервисы, в данном контексте не так важно.

Виды взаимодействия

Межпроцессное взаимодействие подразумевает довольно низкоуровневую поддержку взаимодействия между процессами, включая прямую передачу сообщений посредством API сетевых протоколов (в частности, через сокеты), а также отправку мультикаст-сообщений.

Удалённые вызовы представляет собой наиболее распространённую парадигму взаимодействия в распределённых системах, в рамках которой происходит двусторонний обмен данными посредством вызова удалённой операции, метода или функции. Тут традиционно выделяют три варианта взаимодействия.

- *Протоколы вида “запрос-ответ”* -- весьма популярная парадигма взаимодействия клиент-серверных систем. Подобные протоколы обычно строятся на двустороннем обмене сообщениями от клиента к серверу и обратно, при котором первое сообщение содержит операцию для выполнения на сервере и, возможно, данные для этой операции, а обратно приходит результат выполнения этой операции. Это очень примитивный способ взаимодействия, и имеет смысл использовать его лишь для совсем простых случаев или когда имеет большое значение производительность решения (например, для встроенных устройств). Стоит также учесть, что следующие два

варианта взаимодействия по сути являются надстройками для такого рода протоколов.

- *Удалённые вызовы процедур (remote procedure calls, RPC)*. Концепция RPC, предложенная в 1984 году, была в то время некоторым прорывом в разработке распределённых систем. В рамках этого подхода процедуры выполняются на удалённых устройствах, но при этом выглядит это так, как будто они выполняются в локальном адресном пространстве. Низлежащий уровень middleware прячет все особенности распределённости, включая кодирование аргументов и результата, передачу сообщений и т.п. Этот подход естественно вписывается в клиент-серверную архитектуру, позволяя серверам предоставлять набор операций посредством своего интерфейса, а клиентам вызывать эти операции, как его бы они были доступны локально.
- *Удалённые вызовы методов (remote method invocation, RMI)* -- это по сути RPC, реализованный в рамках объектно-ориентированного подхода. Теперь объектам позволяется вызывать методы удалённых объектов. Подобные middleware идут дальше, позволяя пробрасывать между удалёнными объектами исключения или даже учитывать их при распределённой сборке мусора, предоставляя более тесную интеграцию с используемыми языками и фреймворками.

У всех трёх подходов есть одна общая особенность: взаимодействие представляет собой двунаправленное отношение между отправителем и получателем, подразумевающее явную отправку сообщения или вызов метода/процедуры. Получатели обычно также осведомлены о наличии конкретных отправителей (пространственная связность), и в большинстве случаев оба участника должны существовать в одно и то же время (временн*а*я связность).

Чтобы бороться с этими типами связности, между сущностями организуют **неявное взаимодействие**.

- *Групповое взаимодействие* используется, когда хочется организовать множественную рассылку сообщений “один-ко-многим”. Группы определяются идентификатором, и получатель заявляет о своём желании получать те или иные сообщения, присоединяясь к определённому типу группы. Отправители шлют сообщения группе в целом по идентификатору, тем самым не зная, кем конкретно это сообщение в итоге будет получено.
- *Модель “издатель-подписчик”* используется в случае, когда есть явно выделенные узлы-производители данных, событий или другой информации и большое количество узлов-потребителей, которые в этой информации заинтересованы. Использовать классические схемы коммуникации в этом случае массовых рассылок было бы весьма неэффективно.
- *Очереди сообщений*. Если модель “издатель-подписчик” ориентирована на отношение “один-ко-многим”, то очереди сообщений реализуют каналы связи “точка-точка”. В этом случае производитель данных отправляет сообщение в особую очередь, а получатель вычитывает их оттуда, возможно получая оповещения при возникновении новых сообщений. Как обсуждалось в прошлой лекции, тем самым между отправителем и получателем вводится дополнительный уровень косвенности.

- *Распределённая общая память* предоставляет абстракцию разделения памяти между процессами, которые не имеют общей физической памяти. Программистам тем не менее предоставляет знакомая абстракция чтения и записи общих структур данных, как если бы они находились в локальном адресном пространстве. Вопросами синхронизации и целостности таких данных занимается соответствующее middleware.

Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Роли и обязанности

В процессе взаимодействия в рамках распределённой системы процессы (а на самом деле объекты, компоненты или сервисы) взаимодействуют друг с другом для решения некоторой полезной задачи. Ответственности, которые берут на себя те или иные сущности, сильно определяют получаемую архитектуру и свойства системы в целом. В зависимости от роли каждой сущности рассмотрим две распространённых схемы.

Клиент-сервер -- наиболее часто используемая на практике архитектура распределённых приложений. Клиентские приложения взаимодействуют с потенциально разными процессами серверных приложений с целью выполнения последними некоторых полезных действий или доступа к данным. Система может быть многоуровневой, когда сервер является клиентом для другого сервера.

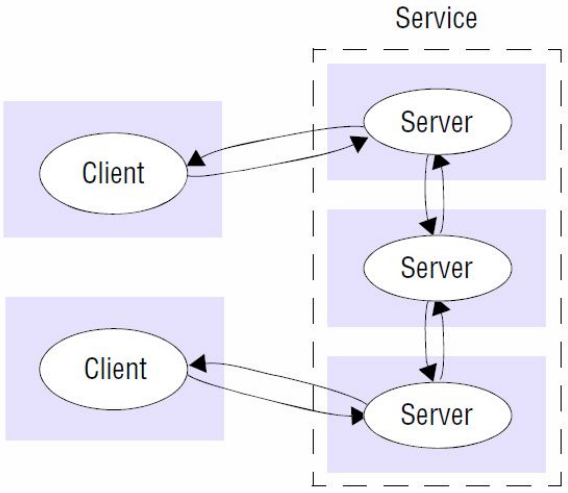
В то время, как клиент-серверная модель представляет прямой и довольно простой в реализации подход, она очень плохо масштабируется. Централизация сервиса и управления этим единым компьютером не дают возможности масштабировать сервис за пределы этого компьютера или как-то снимать с него нагрузку по обработке всех запросов. Архитектура *точка-точка (peer-to-peer, P2P)* подразумевает одинаковые роли для всех составляющих систему сущностей. Узлы на равных взаимодействуют между собой без разграничений на клиентов и серверов. На всех узлах работает одно и то же ПО, предоставляя всем остальным одинаковый интерфейс.

Варианты размещения

Последний вопрос, который стоит зафиксировать при проектировании распределённой системы -- как это объекты и сервисы будут размещаться на физическую инфраструктуру, потенциально состоящую из большого числа компьютеров, объединённых в сеть произвольной сложности. Подобный выбор должен быть основан на шаблонах взаимодействия между узлами и их процессами, надёжности выбранных устройств и их загруженности, качества среды передачи данных между компьютерами и т.п. В каждом конкретном случае эти решения должны приниматься в соответствии с особенностями приложения, тут же мы рассмотрим несколько общих стратегий, которые приводят систему к тому или иному набору качеств.

Разбиение сервисов по нескольким серверам

Сервисы могут размещаться в качестве набора серверных процессов, запущенных на разных компьютерах. Сервера могут поддерживать общее состояние данных, синхронизируя изменения между собой, либо поддерживать каждый собственную копию данных.



Кэширование

Кэш -- хранилище недавно запрашиваемых данных, которое ближе к конкретному клиенту или набору клиентов, чем оригинальный источник данных. Когда происходит получение нового объекта от сервера, этот объект сохраняется в кэш, заменяя предыдущую версию этого объекта, если она там есть. Когда объект требуется клиенту, сначала происходит поиск в кэше, и если там находится актуальная версия, возвращается именно она. Если не находится, происходит полноценный запрос на сервер.

Кэши очень активно используются в реальной практике. Кэширующие прокси-сервера позволяют сильно снизить нагрузку на целевой сервер, параллельно, возможно выполняя и дополнительные обязанности (протоколирование, туннелирование между сетями и т.п.).

Мобильный код

Работа с сервисом означает выполнение кода, который вызывает некоторые операции этого сервиса. Многие сервисы хорошо стандартизованы, так что их использование возможно даже из стандартных приложений. Типичный пример здесь -- веб-браузер, который может быть использован для доступа к множеству разных веб-сервисов. Однако даже тут бывают исключения, когда некоторым сервисам не хватает стандартной функциональности браузеров, и требуется скачивание дополнительного кода, который будет взаимодействовать с сервером по определённому протоколу или реализовывать какую-нибудь нестандартную модель взаимодействия. Например, когда сервер инициирует взаимодействие с клиентом посредством push-уведомлений.

В недавнем прошлом для этого были крайне популярны Java-апплеты, сейчас для этого чаще всего используется JavaScript. Например, это может быть финансовое приложение, которое будет уведомлять своих клиентов об изменениях цен на заданные позиции. Чтобы пользоваться сервисом, каждый клиент скачивает специальный кусок кода, который получает уведомления от серверной компоненты, отображает их пользователю и, возможно, осуществляет операции купли-продажи в соответствии с заданными пользователем и хранящимися на его локальном компьютере правилами.

Преимущество такого подхода в том, что при выполнении на локальном устройстве пропадает задержка, вызванная сетевым взаимодействием, а, значит, повышается отзывчивость и интерактивность кода. Однако, это потенциально угроза безопасности, поэтому браузеры обычно сильно ограничивают ресурсы, к которым имеет доступ подобный код (например, не разрешается прямой доступ к локальным файлам).

Мобильный агент

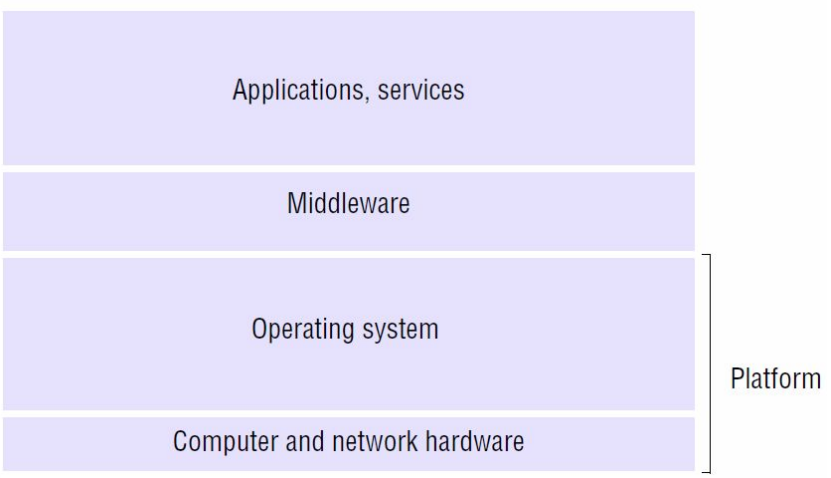
Мобильный агент -- это подход, при котором запускаемая программа и данные для неё передаются по сети и выполняются на удалённом устройстве, выполняя предписанную автором этой программы задачу и возвращая ему в итоге результат. При выполнении эта программа имеет доступ к ресурсам устройства, на котором она работает (например, доступ к локальной базе данных). По сравнению со статическим клиент-серверным подходом в случае необходимости обработки больших объёмов (особенно из разных источников) данных подобный подход может быть гораздо более эффективным. Но это даже ещё большая угроза безопасности, чем мобильный код, поскольку каждое окружение, в котором этот агент запускается, должно само определять набор ресурсов, в которым он будет иметь доступ. К тому же, мобильный агент может оказаться довольно хрупким, если в работе будет сталкиваться с непредвиденными событиями (типа блокировки или отсутствия необходимых данных). Исходя из этого подобный подход используется крайне редко, чаще всего требуемую функциональность можно получить другими способами. Например, реализовать обработку данных через удалённые вызовы.

Архитектурные шаблоны

При разработке распределённых приложений часто применяются следующие архитектурные шаблоны.

Слоистая архитектура (Layered architecture)

Как мы уже обсуждали в прошлых лекциях, подобный подход заключается в разбиении сложной системы на набор слоёв, каждый из которых предоставляет сервисы тем, кто находится выше, и использует сервисы тех, кто находится ниже. С точки зрения распределённых систем это позволяет организовывать вертикальную декомпозицию сервисов на сервисные слои: распределённый сервис может предоставляться одним или несколькими серверными процессами, взаимодействующими между собой и с клиентскими процессами, сохраняя общее представление об используемых ресурсах.



Многоуровневая архитектура (Tiered architecture)

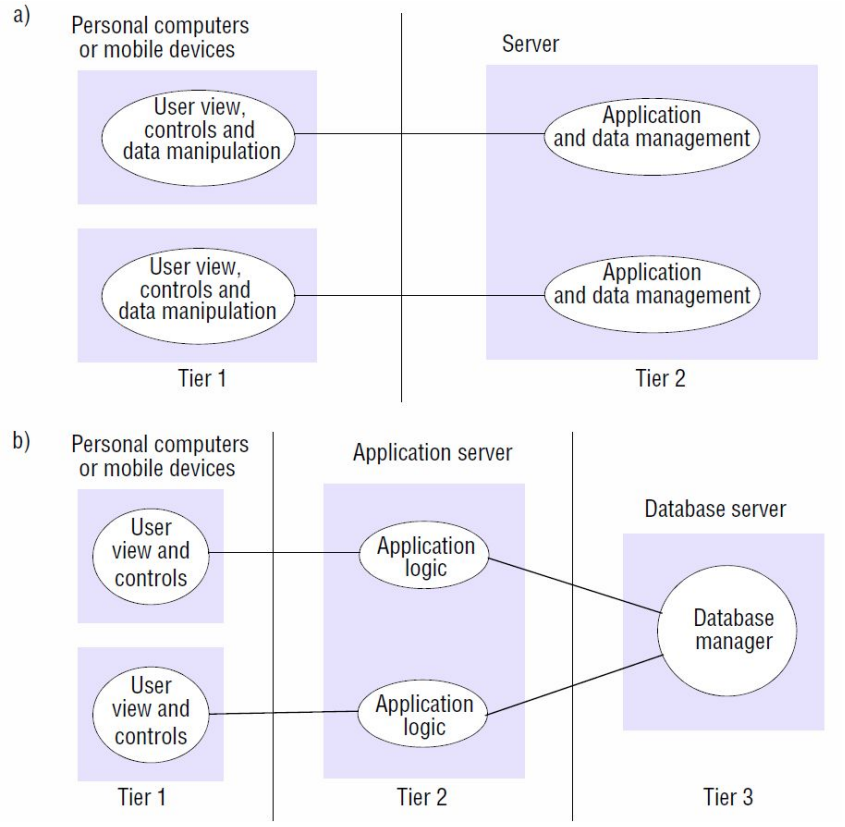
Дальнейшее развитие идеи декомпозиции системы на уровни. Если разбиение на слои позволяет провести вертикальное разбиение сервиса на уровни абстракции, выделение уровней позволяет организовать функциональность каждого слоя и разместить эту функциональность на определённый физический сервер.

На рисунке ниже приведён пример разбиения системы на 2 и 3 уровня соответственно. Функционально эта система должна реализовывать логику отображения (включая взаимодействие с пользователем), бизнес-логику (некую обработку данных в соответствии со своим назначением) и хранение данных.

В двухуровневой схеме часть бизнес-логики будет выполняться на клиенте, часть на сервере. Это позволит получить высокую отзывчивость и скорость реакции пользовательского интерфейса, однако часто бывает трудно адекватно разделить логику между клиентом и сервером.

В трёхуровневом решении каждая часть функциональности представляет собой отдельный уровень и размещается на собственном компьютере. Каждый уровень имеет свою чётко определённую ответственность, однако такая схема увеличивает

время реакции на пользовательские действие, к тому же большим количеством устройств сложнее управлять.



Тонкие клиенты

В современных распределённых системах прослеживается тренд переноса сложной логики с клиентских устройств на серверные сервисы. Эта концепция хорошо применяется и в случае многуровневой архитектуры, и в случае модных нынче облачных вычислений. Термин “тонкий клиент” подразумевает отсутствие какой-либо серьёзной бизнес-логики в клиентском приложении, часто представляющего собой лишь простой оконный интерфейс для доступа к удалённым сервисам.

В результате такого подхода функциональность простых устройств с низкой производительностью и/или малыми объёмами памяти может быть серьёзно расширена за счёт удалённых сервисов, выполняющихся на мощных серверах. Однако, подобный подход весьма плохо работает, когда клиенту необходимо работать с большим количеством данных (например, CAD-система или видео-редактор) ввиду естественных задержек при передаче данных и возможной обработке их удалённым сервисом. Пределом этой идеи являются приложения типа VNC, которые дают возможность удалённо подключаться к серверу и работать с ним как с локальным компьютером. В этом случае все данные остаются на удалённом устройстве, код также выполняется там, а на клиент лишь поступает изображение получаемых экранов.

Межпроцессное сетевое взаимодействие

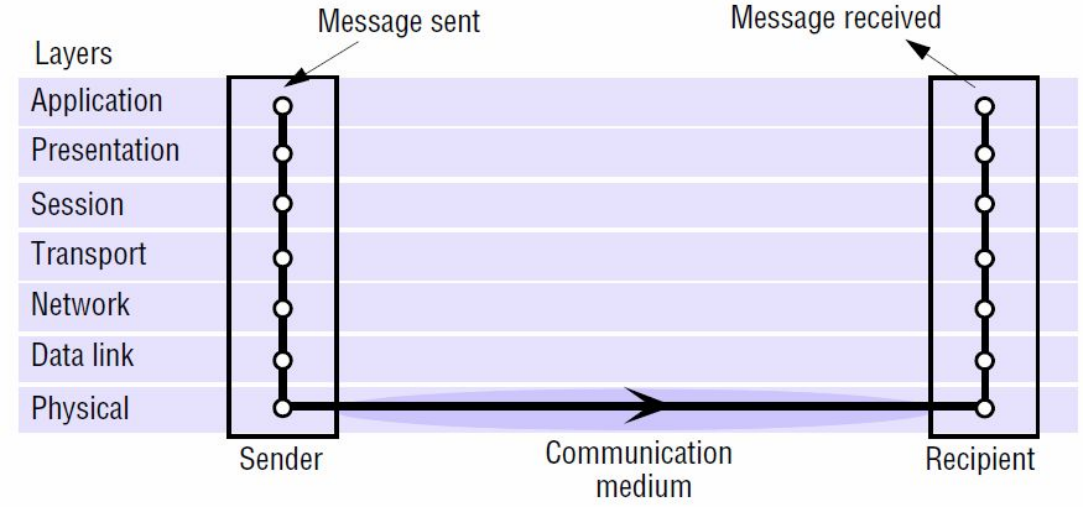
Далее подробно обсудим виды взаимодействия сущностей в распределённых системах, которые упоминались ранее. Самое базовое из них -- межпроцессное взаимодействие, основанное на примитивах сетевого уровня.

Сетевые протоколы

Основа всего сетевого взаимодействия -- пакетная передача данных была заложена ещё в 1960 годах. Данные разбивались на пакеты и рассылались нужным адресатам по единому каналу связи (в противовес коммутируемым телефонным каналам). Пакеты собираются в очереди и отправляются, когда канал связи становится доступным. Взаимодействие асинхронное -- сообщения прибывают на место назначения с задержкой, вызванной временем транспортировки по сети. Пакеты состоят из полезных данных, а также содержат в себе служебную информацию (адрес отправителя и получателя, размер сообщения, контрольную сумму и т.п.).

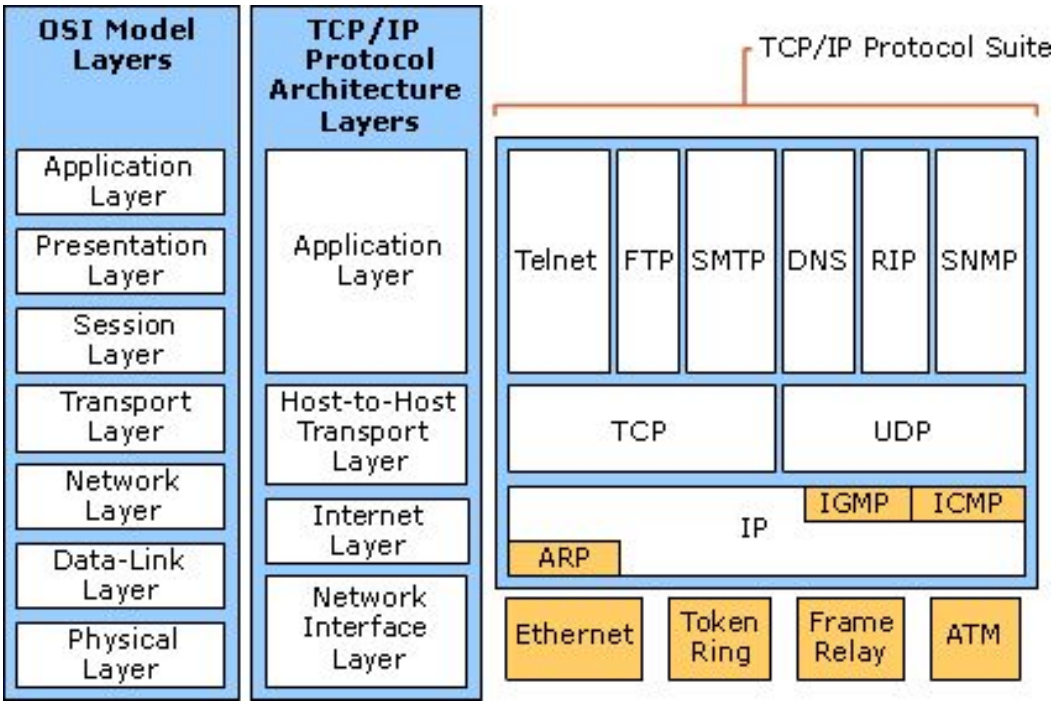
Для организации процесса обмена сообщениями по сети появляется понятие протокола -- набора правил, формально определяющего (1) последовательность обмена сообщениями и (2) формат этих сообщений. Наличие стандартных протоколов позволяет разрабатывать разные компоненты распределённых приложений независимо друг от друга, при помощи разных языков программирования, на разных операционных системах и работающих с разными форматами данных.

Структура ПО для работы с сетью имеет слоистую архитектуру. Каждый слой обычно реализуется отдельным модулем в операционной системе и предоставляет сервис вышележащему уровню, пользуясь сервисом нижележащего. На самом низком уровне -- физическая среда передачи данных (например, медный или оптоволоконный кабель или радио-канал), по которой сигнал передаётся другой стороне взаимодействия. При получении сообщения оно проходит через все те же уровни в обратном порядке, в конечном итоге попадая целевому процессу. Поэтому полный набор слоёв протоколов часто называют стеком протоколов. На рисунке ниже представлена семиуровневая эталонная модель взаимодействия открытых систем.



Как и любая слоистая архитектура, она сильно упрощает обобщение, разработку и использование сетевых приложений, однако вносит существенные накладные расходы. Передача сообщения от одного прикладного приложения другому в архитектуре с N слоями потребует N передач управления между компонентами, реализующими эти уровни (как минимум одно из которых потребует системных вызовов), а также добавления к данным N служебных заголовков. Из-за этого скорость реальной передачи данных может быть ниже, чем потенциально возможная пропускная способность сети.

В реальной жизни модель OSI практически не используется, её заменил ставший стандартом де-факто сетевой стек интернет-протоколов TCP/IP.



В стеке TCP/IP есть два транспортных протокола: TCP (Transport Control Protocol) and UDP (User Datagram Protocol). Оба из них работают на базе сетевого протокола IP (Internet Protocol), который отвечает за адресацию и передачу данных внутри сети.

UDP практически не добавляет ничего к функциональности IP. В его заголовке содержится порт отправителя и получателя (сам адрес хранится в IP-заголовке), длина поля и контрольная сумма (которая опциональна, и кроме неё нет никаких средств обеспечения корректности передаваемых данных). UDP не гарантирует доставку сообщения (IP-пакеты могут теряться при передаче по сети).

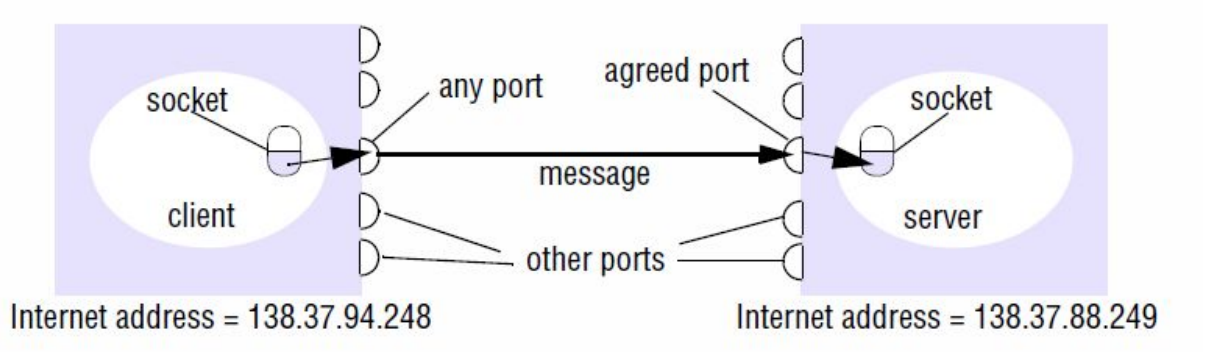
TCP предоставляет гораздо более изысканный транспортный сервис. Он даёт возможность передавать последовательности байт произвольной длины через абстракцию потока. Гарантируется корректность получения пакетов адресатом, причём в том же самом порядке и с теми же самыми данными, которые послал отправитель. TCP ориентирован на предварительные соединения: перед тем, как отправлять какие-либо данные отправитель и получатель проходят процедуру установления двунаправленного канала связи. Соединение -- это соглашение между отправителем и получателем. Промежуточные сервера (маршрутизаторы и т.п.), впрочем, ничего не

знают об этих соединениях, пакеты уровня IP с данными одного TCP соединения могут путешествовать совершенно разными маршрутами. Кратко перечислим основные механизмы, которые TCP привносит поверх IP.

- **Упорядочение пакетов.** Каждому пакету присваивается порядковый номер, в соответствии с которыми на получателе происходит пересортировка полученных пакетов. При потере или задержке пакета все полученные пакеты с большими номерами помещаются в очередь и ждут опоздавшего.
- **Управление потоком данных**. Когда адресат успешно получает сегмент данных, он запоминает его номер. Время от времени он посылает отправителю номер последнего полученного сегмента данных (сообщая, что все предыдущие сегменты им успешно получены) и размер окна, которое задаёт следующий объём данных, которые можно отправить без уведомления. Если общение двустороннее, подобные уведомления могут отправляться внутри обратных пакетов данных, если одностороннее -- то в отдельных пакетах уведомлений.
  Другое применение -- буферизация данных для отправки. Если отправитель генерирует очень часто очень мелкие порции данных, становится крайне неэффективно отправлять их по отдельности. В таком случае используются специальные алгоритмы, которые позволяют объединять небольшие пакеты вместе и отправлять их как один. Для интерактивных приложений это может существенно снизить трафик и повысить эффективность сетевого уровня.
- **Повторная передача данных**. Отправитель сохраняет номера сегментов данных, которые он отправляет. При получении уведомления он удаляет соответствующие сегменты из исходящей очереди, но если уведомления не приходит, инициируется повторная передача.
- **Буферизация**. Для балансировки потока данных на получателе используется буфер входящих сообщений, в котором сообщения могут некоторое время находиться в ожидании обработки. Это позволяет выравнивать скорость обработки данных при задержках сети. Однако, если буфер переполнится, данные будут отбрасываться без записи об их получении, так что отправитель будет пытаться отправить их повторно.
- **Контрольная сумма**, покрывающая заголовок и данные сегмента. Если сегмент не соответствует контрольной сумме, он отбрасывается.

Для программиста API для UDP предоставляет возможность передачи сообщения -- самого простого способа межпроцессного взаимодействия. Можно отправить сообщение и надеяться, что оно будет получено и обработано. API для TCP предоставляет абстракцию двунаправленного потока данных между парой процессов. Данные передаются и принимаются безотносительно разбиения их на пакеты.

Оба транспортных уровня используют абстракцию сокета -- конечной точки соединений между процессами. Для программиста сокет -- это канал между двумя процессами, в который если записать данные, они через некоторое время появятся с другой стороны.



Чтобы получать сообщения, сокет должен быть связан с локальным портом и IP-адресом локального компьютера. При этом IP-адрес позволяет найти компьютер в сети, а порт -- найти конкретное приложение в рамках этого компьютера. Довольно много портов уже распределены под разные нужды, однако в операционной системе всегда есть пул свободных, которые можно выбрать для своих приложений. Приложение может использовать произвольное количество портов, однако использовать один порт совместно для разных приложений запрещается. При этом слать разными процессами сообщения на один и тот же порт вполне допустимо.

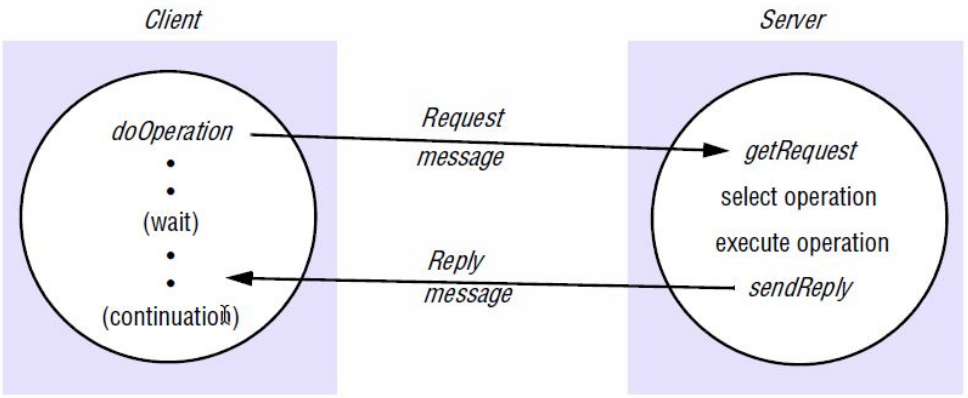
Сокет позволяет осуществлять двустороннее взаимодействие, один и тот же сокет может использовать как для записи, так и для чтения. Каждый сокет ассоциируется при создании с определённым протоколом.

При передаче по сети все используемые в приложениях структуры данных должны быть преобразованы в плоский набор байтов и восстановлены из него обратно по получении. При этом на разных архитектурах используются разные форматы для представления и хранения определённых типов данных. Например, может использоваться разный порядок байтов для представления целых чисел. И это уж не говоря о том, что есть целая куча разных кодировок символов. Процесс преобразования данных из структур программы в плоский набор байтов называется маршалинг, обратное преобразование -- демаршалинг. Важно, что при этом процессе данные по сути не меняются, как не меняются они и при передаче по сети.

Взаимодействие через удалённые вызовы

Протоколы “запрос-ответ”

Данная форма взаимодействия довольно типична для клиент-серверных систем. В большинстве случаев “запрос-ответ” реализуется синхронно, процесс клиента блокируется до тех пор, пока не будет получен ответ от сервера. Подобное взаимодействие также в целом можно считать надёжным, поскольку обязательный ответ от сервера по сути заменяет собой уведомление о получении запроса. Асинхронный вариант реализации тоже вполне допустим, если клиент может подождать и забрать ответ на запрос когда-то после.



На рисунке выше показана типичная схема подобного взаимодействия, состоящая из примитивов doOperation(), getRequest() и sendReply().

Функция doOperation() используется клиентом для выполнения удалённой операции. В неё аргументами передаются адрес удалённого сервера, идентификатор желаемой операции и массив байтов, представляющий аргументы к ней, если необходимо. Возвращаемым значением её будет массив байтов, представляющий ответ. (Предполагаем сейчас, что клиент сам осуществляет маршалинг/демаршалинг данных.) Функция doOperation() отправляет запрос на сервер и делает системный вызов receive() для получения ответа. Функция getRequest() используется серверным процессом, чтобы выполнить запрос. Когда сервер выполнит заданную операцию, он вызовет sendReply(), чтобы передать ответ клиенту. Когда ответ приходит на клиентскую сторону, doOperation() получает управление, извлекает из него результат и возвращает его клиенту. Всё то время, между вызовом doOperation() и возвратом из неё значения клиент остаётся в заблокированном состоянии.

Подобную схему взаимодействия можно реализовать и поверх UDP диаграмм, и поверх TCP-потоков. На самом деле использования UDP датаграмм может иметь даже определённые преимущества в силу специфики задачи:

- уведомления излишни ввиду того, что на каждый запрос приходит ответ;
- установление TCP-соединения требует дополнительных сообщений в добавок к необходимым двум (запросу и ответу);
- управление практически не имеет смысла в случае небольших размеров аргументов и возвращаемых данных.

Но и очевидные недостатки (ненадёжность доставки и отсутствие гарантий порядка сообщений при получении) при реализации протокола поверх UDP тоже никуда не пропадают.

В случае, если серверу не удалось обработать запрос или само сообщение с запросом не было доставлено, внутри doOperations() имеет смысл использовать таймаут на получение ответа. Если по истечении таймаута ответа не получено, запрос нужно посылать повторно.

Если сервер уже отправил ответ на запрос и получает этот же запрос повторно, ему придётся выполнить запрашиваемую операцию повторно, если он не сохранил результат прошлого выполнения. Для хранения результатов можно использовать очередь или любую другую структуру данных, хранящую историю выполненных операций (идентификатор запроса, сообщение, идентификатор клиента, отправленный

ответ). Недостаток такого решения -- дополнительный расход памяти, который может быть существенным, если нужно обрабатывать много разных запросов от разных клиентов. При этом если клиент выполняет только по одной операции за раз, последующий запрос можно считать подтверждением получения ответа на предыдущий и, соответственно, удалять этот результат из временной очереди. Но и тут проблема не решается целиком: когда клиент прекращает свою работу, последний его запрос так и останется в очереди, если не добавлять никаких специальных таймаутов.

При использовании UDP часто возникают сложности, связанные с размером передаваемых полезных данных и фрагментацией пакетов, особенно при построении поверх этих протоколов RPC или RMI-библиотек. Поэтому многие всё же переходят на использование в качестве транспортного протокола TCP, что позволяет не реализовывать мультипакетирование вручную и передавать данные произвольного объёма. В частности, TCP-шные потоки можно удобно подружить с потоками ОО-языков, удобно передавая коллекции объектов любого размера по сети. TCP сам контролирует доставку запросов и ответов, так что все эти механизмы с перепосылкой и историями становятся ненужными. Поэтому TCP часто выбирается для упрощения реализации протокола обмена сообщениями. К тому же если последовательные запросы и ответы отправляются между одной и той же парой клиент-сервер, то накладные расходы на создание соединений и отправку уведомлений уже становятся не так актуальны.

Типовым примером протокола “запрос-ответ” является HTTP. Он реализован поверх TCP и поддерживает ряд запросов (GET, PUT, POST и т.п.), через которые можно воздействовать на ресурсы сервера. В оригинальной версии протокола на каждый запрос открывалось новое соединение между клиентом и сервером, после ответа на сообщение сервер соединение разрывал. Со временем пришло понимание того, что такой механизм не особо эффективен (особенно в случае большого количества разных картинок и т.п. на страницах), поэтому в версии HTTP 1.1 добавили постоянные соединения, позволяющие обмениваться через них запросами и ответами продолжительное время. Клиент может закрыть такое соединение по запросу, сервер -- по истечении определённого времени неактивности.

Данные в запросах и ответах маршализуются в ASCII-строки. Для передачи файлов ресурсов используется стандарт MIME.

Ну и в стандарте HTTP 2.0 появилось много разного интересного, кое-то можно почитать, например, здесь.

Удалённый вызов процедур

Как говорилось ранее, концепция RPC ставит своей целью приблизить программирование распределённых систем к “обычному” программированию. Подобное упрощение достигается за счёт сокрытия деталей, связанных с распределённой природой вызовов (включая маршалинг и всё сетевое взаимодействие), за локальным вызовом функции.

Семантика RPC вызовов

При обсуждении способов реализации протоколов “запрос-ответ” мы столкнулись с тремя основными проблемами гарантированного обмена сообщениями:

- повторная отправка запроса до тех пор, пока либо не придёт ответ, либо мы не решим, что сервер недоступен;
- фильтрование дубликатов запросов на сервере;
- повторная отправка результатов, сохранённых на сервере для того, чтобы не выполнять операции повторно.

Комбинации этих техник дают разные возможные варианты семантики удалённых вызовов с точки зрения отправителя. На рисунке ниже представлены возможные варианты такого рода взаимодействий. Заметим, что для обычных локальных вызовов процедур семантика будет “exactly once”.

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Семантика “maybe”: удалённая процедура может быть вызвана один раз или не быть вызвана вообще ввиду того, что сообщение с запросом или ответом было потеряно (или пришло по истечении таймаута) или сервер не смог выполнить нужную операцию.

Семантика “at-least-once”: отправитель получает результат, в связи с чем уверен, что операция была выполнена хотя бы один раз, либо получает уведомление/исключение о том, что операция не была завершена успешно. Подобный результат достигается путём повторных посылок запроса до тех пор, пока на них не будет получен ответ. В запросах с такой семантикой имеет смысл посылать лишь идемпотентные операции, иначе сложно делать какие-либо предположения о внутреннем состоянии сервера.

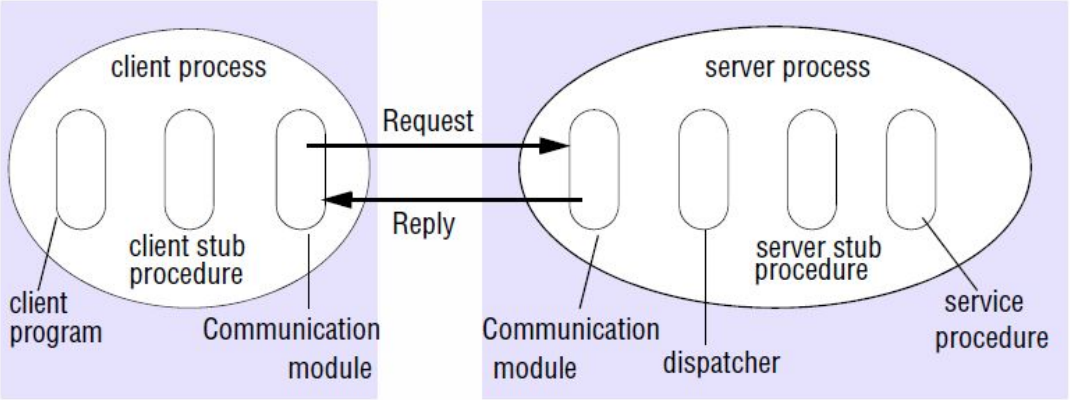
Семантика “at-most-once”: отправитель либо получает результат (и в таком случае уверен, что операция была выполнена ровно один раз), либо получает исключение, информирующее о том, что результат получен не был (и, соответственно, операция была выполнена либо один раз неудачно, либо не выполнена совсем). Подобное поведение достигается при применении всех техник по обеспечению гарантий доставки, приведённых на рисунке выше. Это наиболее привычный вид семантики вызовов, и он реализован в большей части современных RPC-библиотек.

Прозрачность RPC вызовов

Авторы изначальной идеи RPC Birrell и Nelson (1984), ставили своей целью убрать какие-либо различия между синтаксисом локального и удалённого вызова процедуры. Однако, удалённые вызовы процедур куда более склонны к отказам, чем локальные, так как зависят от работы сети и другого процесса на удалённом компьютере. Какая бы ни была реализована семантика вызовов, всегда есть вероятность, что ответ не будет получен, и клиентам нужно иметь возможность различать ситуацию, когда произошёл отказ сети или отказ удалённого сервиса. Время выполнения удалённого вызова процедуры также гораздо больше, чем локального, и это тоже нужно учитывать пользователям RPC библиотек.

В современных RPC решениях достигнуто соглашение о том, что во имя единообразия удалённые вызовы должна быть синтаксически похожи на локальные, однако различия между ними должны быть отражены в интерфейсе RPC-функций. Так, например, удалённый вызов может бросить особый тип исключения, если клиент будет неспособен выполнить процедуру на сервере. В интерфейсе RPC-функций также может быть закодирована семантика их вызовов -- всё это позволяет разработчику более осознанно и эффективно использовать удалённые вызовы в своём коде.

Структура RPC middleware

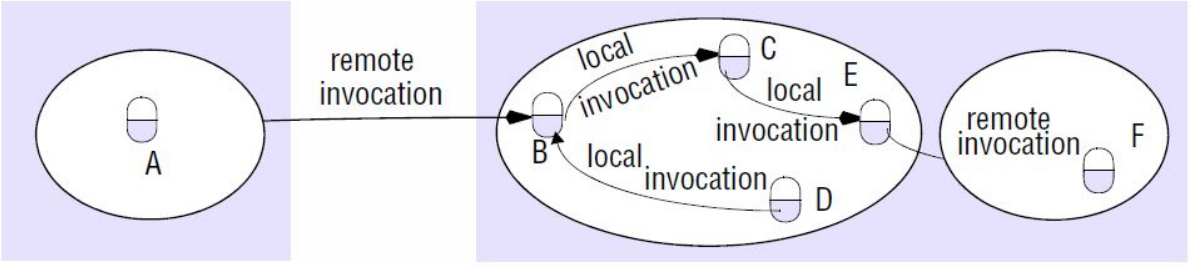


Структурные компоненты, обеспечивающие работу RPC, показаны на рисунке выше. Клиент содержит функцию-заглушку для каждой удалённой процедуры в интерфейсе сервера. Эти заглушки являются локальными процедурами для клиента и отвечают за маршализацию параметров процедуры и других служебных данных в сообщение, отправку этого сообщения на сервер, получение ответа и демаршализацию результата. На сервере находится диспетчер, который в зависимости от информации в сообщении вызывает нужную заглушку для целевой процедуры. Серверная функция-заглушка распаковывает данные из сообщения, вызывает нужную процедуру с заданными параметрами, запаковывает её результат в сообщение-ответ и передаёт его коммуникационному модулю для отправки клиенту.

Сервисные процедуры реализуют интерфейс сервера и реализуются разработчиками вручную, однако функции-заглушки, диспетчер и коммуникационные модули генерируются автоматически по описанию интерфейса сервиса.

Удалённый вызов методов

Удалённый вызов методов (RMI) является расширением идей RPC на мир объектно-ориентированного программирования. С использованием RMI становится возможным вызов методов у потенциально удалённого объекта.



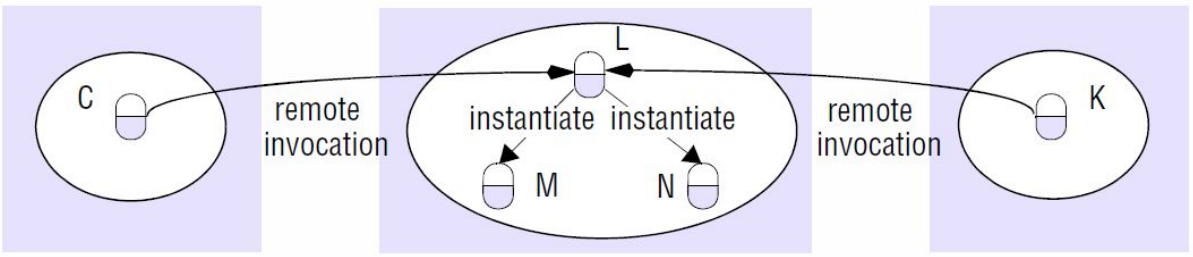
В рамках RMI каждый процесс содержит в себе набор объектов, некоторые из которых доступны только локально, а некоторые могут обрабатывать удалённые вызовы. Удалённым вызовом будем считать вызов метода между объектами, находящимися в разных процессах, запущенных на одном компьютере или на разных. Локальными будет считать вызовы, совершаемые в рамках одного процесса. Удалённые объекты -- те, которые могут принимать удалённые вызовы. На рисунке выше объекты B и F -- удалённые объекты.

Понятие ссылки объекта расширяется с учётом удалённых вызовов до понятия ссылки на удалённый объект. Это идентификатор, который уникален в рамках всей системы, и который используется для уникальной идентификации объекта.

Для объектов в других процессах удалённый объект будет доступен лишь через свой удалённый интерфейс. Например, в Java RMI удалённые интерфейсы определяются как обычные Java-интерфейсы, расширяющие интерфейс Remote.

Как и в обычно, не распределённом случае, вызов метода может приводить к последующему вызову других методов, и т.д. В случае, если эти вызовы пересекают границы процесса, в дело вступает RMI, и вызывающий должен обладать ссылкой на удалённый объект. Например, на рисунке выше объект A должен иметь ссылку на удалённый объект B. Ссылки на удалённые объекты также могут быть получены как результат вызова удалённых методов. Например, при вызове объектом A метода B последний может вернуть ему ссылку на объект F, и тогда A сможет вызывать удалённые методы у F напрямую.

Когда вызов приводит к созданию нового объекта, обычно он создаётся в рамках того процесса, где выполняется создающий его конструктор. Если у этого объекта есть удалённый интерфейс, он будет доступен для удалённых вызовов.

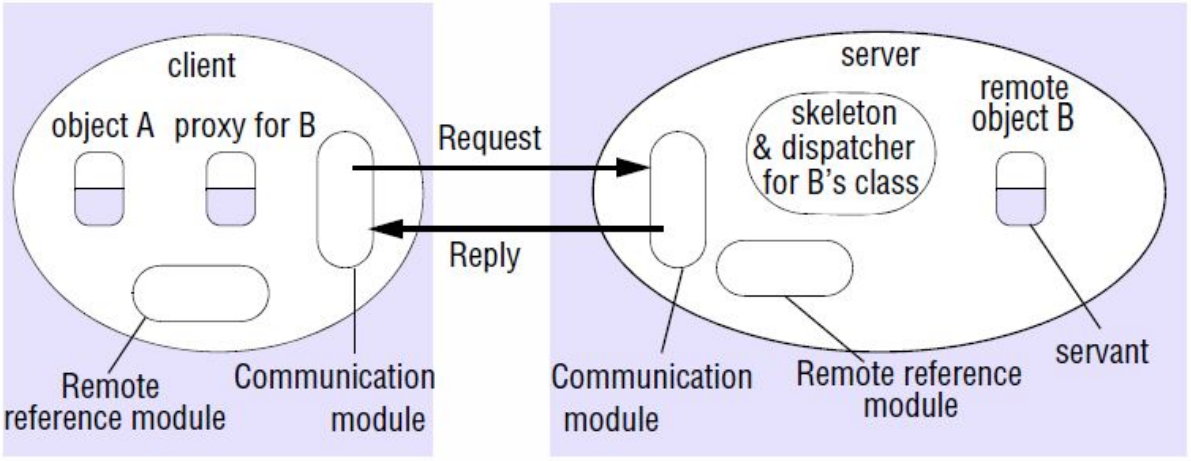


Как и в RPC, все детали реализации этого спрятаны от пользователя. Некоторые распространённые возможности RMI-библиотек приведены ниже.

- RMI позволяет передавать параметры в функции не только по значению как входные или выходные, но и по ссылке. В случае удалённой процедуры это позволяет не передавать весь объект по сети, получая доступ к его методам через удалённую ссылку на объект.
- В языках типа Java, основывающихся на сборке мусора, RMI даёт возможность осуществлять сборку мусора над распределёнными объектами. Для этого существующий сборщик мусора расширяется дополнительным модулем, учитывающий удалённые ссылки на объекты.
- Транспортный уровень RMI позволяет прозрачно обрабатывать исключения, произошедшие в удалённом процессе.

Структура RMI middleware

На рисунке ниже показана структура компонентов, обеспечивающих работу RMI. В приведённом примере объект A вызывает метод удалённого объекта B.



Два взаимодействующих коммуникационных модуля реализуют протокол ответ-запрос, передавая сообщения между клиентом и сервером, реализуя ту или иную семантику вызовов. На сервер коммуникационный модуль передаёт диспетчеру ссылку на объект, у которого нужно сделать вызов. Эту ссылку коммуникационный модуль получает у модуля удалённых ссылок по идентификатору объекта, извлечённому из полученного сообщения.

Модуль удалённых ссылок ответственен за преобразование между ссылками на локальные и удалённые объекты и за создание ссылок на удалённые объекты. Для этого в каждом процессе содержится таблица, которая хранит в себе соответствие между локальными объектами этого процесса и их удалёнными идентификаторами (уникальными в рамках всей системы). То есть подобная таблица на сервере будет хранить ссылку на удалённый объект B. Кроме того, чтобы поддерживать исходящие удалённые вызовы, таблица содержит список всех локальных прокси удалённых объектов. Например, в таблице на клиенте будет храниться ссылка на прокси-объект объекта B.

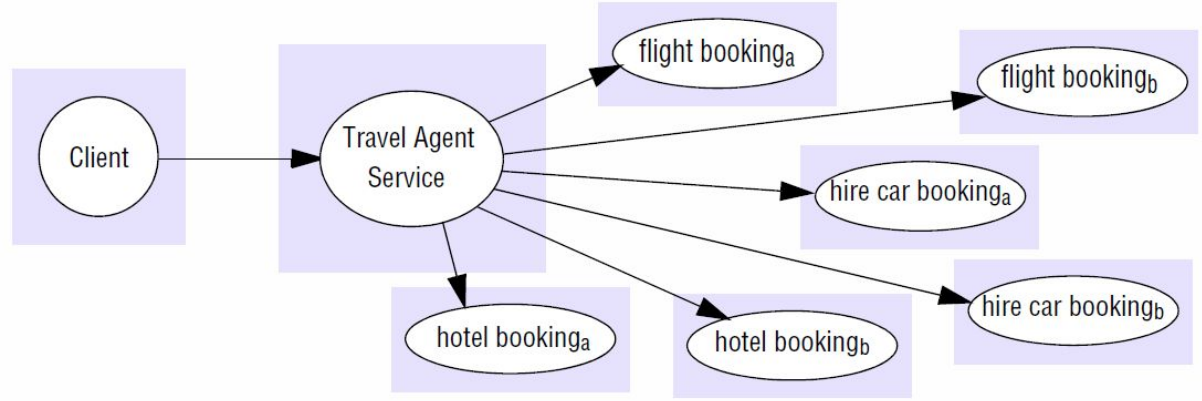
Данный модуль вызывается другими компонентами RMI, когда они сталкиваются с ссылками на удалённые объекты. Например, когда приходит запрос, таблица используется, чтобы понять, у какого локального объекта нужно сделать вызов метода.

Прокси -- аналог процедуры-заглушки RPC. Его роль -- обеспечить прозрачность вызова для клиентского кода. Он прячет внутри всю работу по преобразованию ссылок, маршалинг аргументов, демаршалинг результатов и всё сетевое взаимодействие. Для каждого удалённого объекта, на который процесс имеет удалённую ссылку, создаётся собственный прокси-объект, реализующий удалённый интерфейс замещаемого им объекта.

Назначение диспетчера аналогично назначению диспетчера в RPC.

Web-сервисы

Развитие веба за последние несколько десятилетий показало эффективность простых протоколов, используемых в интернете, как базы для большого числа сервисов и приложений. Однако, использование браузеров как приложений общего назначения даже с расширениями в виде скачиваемого и исполняемого на клиенте кода ограничивает потенциальный круг приложений. В оригинальной схеме клиент-сервер и клиент, и сервер имели свои функциональные особенности. Веб-сервисы возвращаются к этой модели, когда специализированные клиенты обращаются к сервисам, предоставляющим через интернет определённые услуги и реализующим обмен чётко структурированными сообщениями. (Ввиду подобной специализации обращаться к веб-сервисам через браузер довольно бессмысленно.) Это позволяет делать довольно сложные бизнес-приложения путём комбинации функциональности, предоставляемой различными веб-сервисами. Например, на основе веб-сервисов, позволяющих заказывать авиабилеты, отели и арендовать автомобили, можно построить комплексный веб-сервис, который будет агрегировать в себе всю эту функциональность в зависимости от заданных критериев.



Подобный пример демонстрирует две возможные альтернативы взаимодействия, реализуемые в веб-сервисах:

- обработка заказа билета требует долгого времени на исполнение и может быть реализована через обмен асинхронными сообщениями с информацией о датах и местах назначения, промежуточными статусами выполнения и результатом операции в самом конце. Производительность в этом варианте не является ключевым моментом.

- Оплата посредством кредитной карты должна проводиться посредством протокола “запрос-ответ”.

На сегодняшний день наибольшее распространение получили следующие протоколы реализации веб-сервисов:

- SOAP (Simple Object Access Protocol) -- по сути это тройка стандартов SOAP/WSDL/UDDI;
- REST (Representational State Transfer);
- XML-RPC (XML Remote Procedure Call).

Последний уже по сути давно не используется, REST мы будем рассматривать на следующем занятии, а сейчас кратко обсудим SOAP.

SOAP (Simple Object Access Protocol) — протокол обмена сообщениями между потребителем и поставщиком веб-сервиса. WSDL (Web Services Description Language) — язык описания внешних интерфейсов веб-службы. UDDI (Universal Discovery, Description and Integration) — универсальный интерфейс распознавания, описания и интеграции, используемый для формирования каталога веб-сервисов и доступа к нему. SOAP-сообщения строятся на основе XML и выглядят примерно так:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Get up at 6:30 AM</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

Правила, по которым составляются сообщения для веб-сервиса, описываются также с помощью XML и также имеют четкую структуру. То есть если веб-сервис

предоставляет возможность вызова какого-то метода, он должен дать возможность клиентам узнать, какие параметры для данного метода используются. Если веб-сервис ждет строку для метода Method1 в качестве параметра и строка должна иметь имя Param1, то в описании веб-сервиса эти правила будут указаны. Для подобных описаний используется язык WSDL. Примеры WSDL-описаний можно посмотреть здесь. Для клиентов достаточно знать URL веб-сервиса, его WSDL-описание всегда будет рядом, а по нему можно получить представление о методах и их параметрах, которые предоставляет этот веб-сервис.

Укажем некоторые достоинства подобного подхода.

- В большинстве систем описание методов и типов происходит в автоматическом режиме. То есть программисту на сервере достаточно сказать, что данный метод можно вызывать через веб-сервис, и его WSDL-описание будет сгенерировано автоматически.
- Описание, имеющее четкую структуру, читается любым SOAP-клиентом единообразно. То есть какой бы ни был веб-сервис, клиент поймёт, какие данные веб-сервис принимает. По этому описанию клиент может опять же автоматически построить свою внутреннюю структуру классов объектов, с которой программист потом уже будет работать.
- Автоматическая валидация:
    - xml-валидация: сообщение должно представлять собой корректный XML. Невалидный XML сразу приводит к возврату ошибки клиенту.
    - валидация сообщения по XML-схеме: опять же, если сообщение некорректно, клиенту возвращается ошибка.
    - проверка корректности типов данных веб-сервисом.
- Веб-сервисы могут работать как по SOAP-протоколу, так и просто по HTTP через GET-запросы. То есть если в качестве параметров идут простые данные (без структуры), то часто можно вызвать просто обычный get www.site.com/users.asmx/GetUser?Name=Vasia или POST. Очень удобно при ручной отладке.

Но есть у использования SOAP и недостатки.

- Неоправданно большой размер сообщений. Сама природа XML такова, что формат избыточный, чем больше тегов, тем больше неполезной информации. Плюс сам SOAP добавляет своей избыточности. Для локальный сетей вопрос трафика стоит менее остро, чем для интернета, поэтому SOAP-сервисы всё ещё используются внутри корпоративных систем.
- Все действия по вызову методов должны быть атомарными. Например, работая с СУБД мы можем начать транзакцию, выполнить несколько запросов, потом откатиться или закоммитить. В SOAP транзакций нет. Один запрос -- один ответ, всё остальное нужно поддерживать вручную на уровне бизнес-логики.