Representational State Transfer (REST)

Развитие веб-сервисов привело к тому, что в 2000 году один из авторов HTTP Рой Филдинг в рамках своей диссертации подвёл теоретическую основу под способ взаимодействия клиентов и серверов в интернете. Филдинг описал концепцию построения распределённого приложения, при которой каждый запрос клиента к серверу содержит в себе исчерпывающую информацию о желаемом ответе сервера, и сервер не обязан сохранять информацию о состоянии клиента (т.н. клиентской сессии). Описанный архитектурный стиль получил название REST и с тех пор крайне активно используется для построения веб-служб. Развивался REST параллельно с HTTP 1.1, разработанным в 1996-99 годах.

Существует шесть ограничений для построения распределённых REST-приложений по Филдингу.

1. **Модель клиент-сервер**. В основе данного ограничения лежит разграничение потребностей. Отделяя потребности интерфейса клиента от потребностей сервера, хранящего данные, повышается переносимость кода клиентского интерфейса на другие платформы, а упрощая серверную часть, улучшается масштабируемость.
2. **Отсутствие состояния**. Протокол взаимодействия между клиентом и сервером требует соблюдения следующего условия: в период между запросами клиента никакая информация о состоянии клиента на сервере не хранится. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Состояние сессии при этом сохраняется на стороне клиента. Информация о состоянии сессии может быть передана сервером какому-либо другому сервису (например в службу базы данных) для поддержания устойчивого состояния, например, с целью и на период установления аутентификации. Клиент инициирует отправку запросов, когда он готов (возникает необходимость) перейти в новое состояние. Во время обработки клиентских запросов, считается, что клиент находится в переходном состоянии.
3. **Кэширование**. Как и клиенты, а также промежуточные узлы могут выполнять кэширование ответов сервера. Ответы сервера в свою очередь должны иметь явное или неявное обозначение как кэшируемые или некэшируемые с целью предотвращения получения клиентами устаревших или неверных данных в ответ на последующие запросы. Правильное использование кэширования способно еще больше повысить производительность и расширяемость системы.
4. **Единообразие интерфейса**. Наличие унифицированного интерфейса являются фундаментальным требованием дизайна REST-сервисов. Унифицированные интерфейсы позволяют каждому из сервисов развивается независимо. Все ресурсы идентифицируются в запросах, например, с использованием URL. Ресурсы концептуально отделены от представлений, которые возвращаются клиентам. Например, сервер может отсылать данные из базы данных в виде HTML, XML или JSON, ни один из которых не является

типом хранения внутри сервера. Например, в библиотечной системе чтобы получить третью книгу с книжной полки, надо будет отправить запрос вида /book/3, а 35 страницу в этой книге — /book/3/page/35. Причем совершенно не имеет значения, в каком формате находятся данные по адресу /book/3/page/35 – это может быть и HTML, и отсканированная копия в виде jpeg-файла, и документ Microsoft Word.

5. **Слои**. Клиент обычно не способен точно определить, взаимодействует ли он напрямую с сервером или же с промежуточным узлом в связи иерархической структурой сетей. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределенного кэширования.
6. **Код по требованию** (необязательное ограничение). REST может позволить расширить функциональность клиента за счёт загрузки кода с сервера в виде апплетов или сценариев.

Управление данными сервиса целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол -- HTTP. Для него действие над данными задается с помощью методов: GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить). Таким образом, действия CRUD (Create-Read-Update-Delete) могут выполняться как со всеми 4-мя методами, так и только с помощью GET и POST. Типичная связь между HTTP-методами и URL приложений такова.

- Коллекции элементов (ссылки вида http://api.example.com/resources/):
  - GET: отображение списка URI всех объектов коллекции;
  - PUT: замена всей коллекции целиком на передаваемую в запросе;
  - POST: создание нового элемента в коллекции. Новому объекту присваивается URI и обычно возвращается клиенту как результат операции;
  - DELETE: удаление всей коллекции целиком.
- Элементы (ссылки вида http://api.example.com/resources/item/17):
  - GET: получение представления объекта с заданным URI;
  - PUT: замена существующего объекта на передаваемый в запросе или создание нового, если объекта с таким URI не существует;
  - POST: обычно не используется, но может создавать новый элемент в коллекции, если такого не существует;
  - DELETE: удалить заданный элемент из коллекции.

Для примера с книгами это может выглядеть примерно так:

GET /book/ — получить список всех книг

GET /book/3/ — получить книгу номер 3

PUT /book/3/ — заменить книгу (данные в теле запроса)

POST /book – добавить книгу (данные в теле запроса)

DELETE /book/3 – удалить книгу

Вообще POST может использоваться одновременно для всех действий изменения (ввиду того, что PUT и DELETE не так сильно распространены):

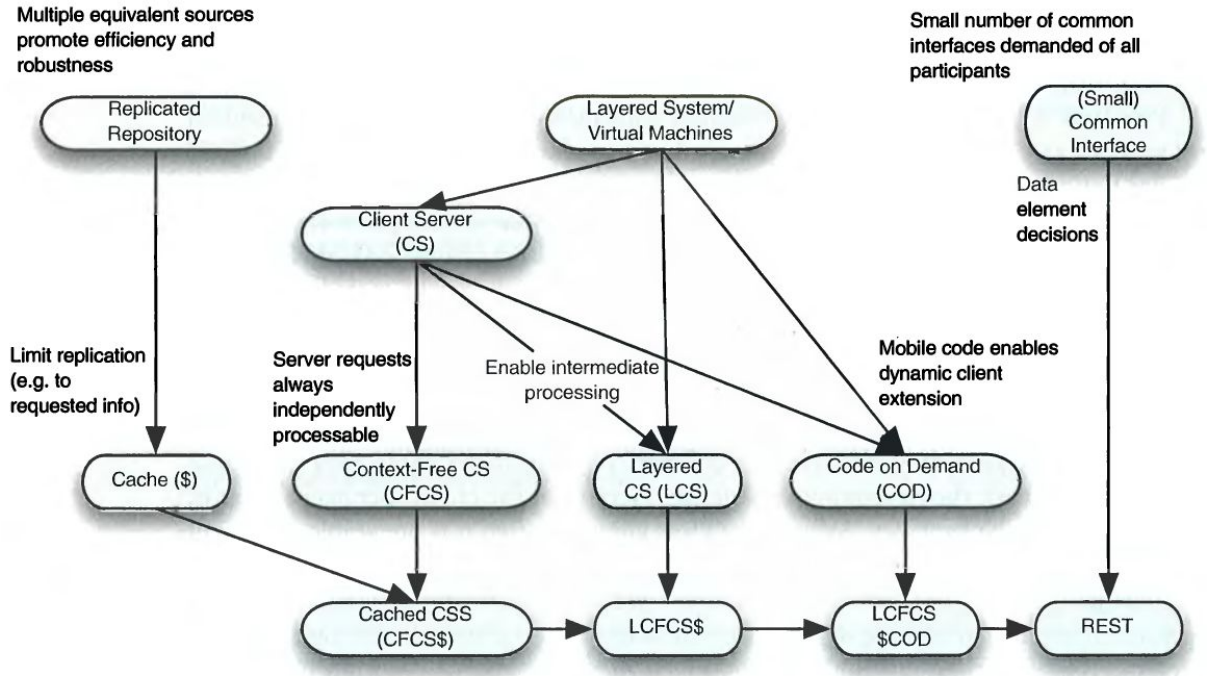
POST /book/ – добавить книгу (данные в теле запроса)

POST /book/3 – изменить книгу (данные в теле запроса)

POST /book/3 – удалить книгу (тело запроса пустое)

Как видно, в архитектура REST очень проста в плане использования. По виду пришедшего запроса сразу можно определить, что он делает, не разбираясь в форматах (в отличие от SOAP, XML-RPC). Данные передаются без применения дополнительных слоев, поэтому REST считается менее ресурсоемким, поскольку не надо парсить запрос чтобы понять, что он должен сделать, и не надо переводить данные из одного формата в другой.

REST как архитектурный стиль впитал в себя особенности многих других стилей и шаблонов, которые мы рассматривали ранее. На рисунке ниже показана подобная преемственность.



Следуя принципам REST, система получает следующие преимущества:

- надёжность (за счёт отсутствия необходимости сохранять информацию о состоянии клиента, которая может быть утеряна);
- производительность (за счёт использования кэша);
- масштабируемость;
- прозрачность системы взаимодействия (особенно необходимая для приложений обслуживания сети);
- простота интерфейсов;
- портативность компонентов;
- лёгкость внесения изменений.

Микросервисы

Развитием идеи веб-сервисов для разработки распределённых приложений, стал архитектурный стиль микросервисов. Приведём здесь некоторые наиболее важные части известной статьи Мартина Фаулера, посвящённой по этому вопросу.

Микросервисная архитектура -- это подход к разработке распределённых систем, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными, используя легковесные механизмы, как правило HTTP. Эти сервисы построены вокруг бизнес-потребностей и развертываются независимо друг от друга с использованием полностью автоматизированной среды. Централизованное управление этими сервисами практически не осуществляется. Сами по себе эти сервисы могут быть написаны на разных языках и использовать разные технологии хранения данных.

Микросервисный стиль разработки принято сравнивать с монолитом (monolithic style): приложением, построенном как единое целое. Enterprise и веб-приложения часто включают три основные части: пользовательский интерфейс (состоящий как правило из HTML страниц и javascript-а), база данных (как правило реляционная, со множеством таблиц) и сервер. Серверная часть обрабатывает HTTP запросы, выполняет бизнес-логику приложения, запрашивает и обновляет данные в БД, заполняет HTML страницы, которые затем отправляются браузеру клиента. Любое изменение в системе приводит к пересборке и развертыванию новой версии серверной части приложения.

Монолитный сервер — довольно очевидный способ построения подобных систем. Вся логика по обработке запросов выполняется в единственном процессе, при этом вы можете использовать возможности используемого языка программирования для разделения приложения на компоненты, классы и т.д.. Вы можете запускать и тестировать приложение на машине разработчика и использовать стандартный процесс развертывания для проверки изменений перед выкладыванием их в production. Вы можете масштабировать монолитное приложения горизонтально путем запуска нескольких физических серверов за балансировщиком нагрузки.

Монолитные приложения могут быть успешными, но все больше людей разочаровываются в них, особенно в свете того, что все больше приложений развертываются в облаке. Любые изменения, даже самые небольшие, требуют пересборки и развертывания всего монолита. С течением времени, становится труднее сохранять хорошую модульную структуру, изменения логики одного модуля имеют тенденцию влиять на код других модулей. Масштабировать приходится все приложение целиком, даже если это требуется только для одного модуля этого приложения.

Эти неудобства привели к архитектурному стилю микросервисов: построению приложений в виде набора сервисов. В дополнение к возможности независимого развертывания и масштабирования каждый сервис также получает четкую физическую границу, которая позволяет разным сервисам быть написанными на разных языках программирования. Они также могут разрабатываться разными командами.

Разбиение через сервисы

ПО традиционно принято разбивать на компоненты. Будем считать под компонентой единицу программного обеспечения, которая может быть независимо заменена или обновлена. Микросервисы используют библиотеки, но их основной способ разбиения приложения -- путем деления на сервисы. Под библиотеками обычно понимают компоненты, которые подключаются к программе и вызываются ею

в том же процессе, в то время как сервисы — это компоненты, выполняемые в отдельном процессе и взаимодействующие между собой через веб-запросы или RPC.

Главная причина использования сервисов вместо библиотек -- это их независимое развертывание. Если вы разрабатываете приложение, состоящее из нескольких библиотек, работающих в одном процессе, любое изменение в этих библиотеках приводит к переразвертыванию всего приложения. Но если ваше приложение разбито на несколько сервисов, то изменения, затрагивающие какой-либо из них, потребуют переразвертывания только изменившегося сервиса. Конечно, какие-то изменения будут затрагивать интерфейсы, что, в свою очередь, потребует некоторой координации между разными сервисами, но цель хорошей архитектуры микросервисов -- минимизировать необходимость в такой координации путем установки правильных границ между микросервисами, а также механизма эволюции контрактов сервисов.

Другое следствие использования сервисов как компонент — более явный интерфейс между ними. Часто только документация и дисциплина предотвращают нарушение инкапсуляции компонентов. Сервисы позволяют избежать этого через использование явного механизма удаленных вызовов.

Тем не менее, использование сервисов подобным образом имеет свои недостатки. Удаленные вызовы работают медленнее, чем вызовы в рамках процесса, и поэтому API должен быть менее детализированным, что часто приводит к неудобству в использовании. Если вам нужно изменить набор ответственностей между компонентами, сделать это сложнее из-за того, что вам нужно пересекать границы процессов.

Smart endpoints and dumb pipes

При выстраивании коммуникаций между процессами часто в механизмы передачи данных помещается существенная часть логики. Хорошим примером здесь является Enterprise Service Bus (ESB). ESB-продукты часто включают в себя изощренные возможности по передаче, оркестровке и трансформации сообщений, а также применению бизнес-правил.

Сообщество микросервисов предпочитает альтернативный подход: умные конечные точки сообщений и глупые каналы их передачи. Приложения, построенные с использованием микросервисной архитектуры, стремятся быть настолько decoupled и cohesive, насколько возможно: они сами реализуют нужную бизнес-логику и выступают больше в качестве фильтров в классическом Unix-овом смысле - получают запросы, применяют логику и отправляют ответ. Вместо сложных протоколов, таких как WS-* или BPEL, они используют простые REST-протоколы. Два наиболее часто используемых протокола — это HTTP запросы через API ресурса и легковесный месседжинг. Часто используемые ресурсы могут быть закэшированы с очень небольшими усилиями со стороны разработчиков или IT-администраторов.

В монолитном приложении компоненты работают в одном процессе и коммуницируют между собой через вызов методов. Наибольшая проблема в смене монолита на микросервисы лежит в изменении шаблона коммуникации. Наивное портирование один к одному приводит к «болтливым» коммуникациям, которые

работают не слишком хорошо. Вместо этого вы должны уменьшить количество коммуникаций между модулями.

Микросервисы и SOA

Когда заходит разговор о микросервисах, обычно возникает вопрос о том, не является ли это обычным Service Oriented Architecture (SOA), которая была популярна в 2000-х годах в связке с SOAP-ориентированными веб-сервисами. В этом вопросе есть здравое зерно, т.к. стиль микросервисов очень похож на то, что изначально закладывали в идею веб-сервисов. Однако со временем термин SOA стал иметь слишком много разных значений, слишком фокусируясь на конкретных технологиях типа ESB и стандартах типа WS*. Безусловно, многие практики, используемые в микросервисах, пришли из опыта интеграции сервисов в крупных организациях. Например, использование простых протоколов -- реакция на чрезмерно усложнённые со временем централизованные стандарты. Всё это привело к тому, что сторонники микросервисов стараются отделить используемый ими подход от “заезженного” термина SOA.

Проектирование под отказ (Design for failure)

Следствием использования сервисов как компонентов является необходимость проектирования приложений так, чтобы они могли работать при отказе отдельных сервисов. Любое обращение к сервису может не сработать из-за его недоступности. Клиент должен реагировать на это настолько терпимо, насколько возможно. Это является недостатком микросервисов по сравнению с монолитом, т.к. это вносит дополнительную сложность в приложение. Как следствие, разработчики микросервисов должны постоянно думать над тем, как недоступность сервисов влияет на user experience. Например, инструмент Simian Army от Netflix искусственно симулирует отказы сервисов и даже датацентров в течение рабочего дня для тестирования отказоустойчивости приложения и служб мониторинга.

Так как сервисы могут отказать в любое время, очень важно иметь возможность быстро обнаружить неполадки и, если возможно, автоматически восстановить работоспособность сервиса. Микросервисная архитектура делает большой акцент на мониторинге приложения в режиме реального времени, проверке как технических элементов (например, как много запросов в секунду получает база данных), так и бизнес-метрик (например, как много заказов в минуту получает приложение). Семантический мониторинг может предоставить систему раннего предупреждения проблемных ситуаций, позволяя команде разработке подключиться к исследованию проблемы на самых ранних стадиях.

Это особенно важно с случае с микросервисной архитектурой, т.к. разбиение на отдельные процессы и коммуникация через события иногда приводит к неожиданному поведению. Мониторинг крайне важен для выявления нежелательных случаев такого поведения и быстрого их устранения.

Монолиты могут быть построены так же прозрачно, как и микросервисы. На самом деле, так они и должны строиться. Разница в том, что знать, когда сервисы, работающие в разных процессах, перестали корректно взаимодействовать между

собой, намного более критично. В случае с библиотеками, расположенными в одном процессе, такой вид прозрачности скорее всего будет не так полезен.

Синхронные вызовы считаются опасными

Каждый раз, когда вы имеете набор синхронных вызовов между сервисами, вы сталкиваетесь с эффектом мультипликации времени простоя (downtime). Время простоя вашей системы становится произведением времени простоя индивидуальных компонент системы. Вы сталкиваетесь с выбором: либо сделать ваши вызовы асинхронными, либо мириться с простоями. К примеру, в www.guardian.co.uk разработчики ввели простое правило -- один синхронный вызов на один запрос пользователя. В Netflix же вообще все API являются асинхронными.

Эволюционный дизайн

Те, кто практикует микросервисную архитектуру, обычно много работали с эволюционным дизайном и рассматривают декомпозицию сервисов как дальнейшую возможность дать разработчикам контроль над рефакторингом их приложения без замедления самого процесса разработки. Контроль над изменениями не обязательно означает уменьшение изменений: с правильным подходом и набором инструментов вы можно делать частые, быстрые, хорошо контролируемые изменения.

Каждый раз, когда вы пытаетесь разбить приложение на компоненты, вы сталкиваетесь с необходимостью принять решение, как именно делить приложение. Есть ли какие-то принципы, указывающие, как наилучшим способом «нарезать» наше приложение? Ключевое свойство компонента -- это независимость его замены или обновления, что подразумевает наличие ситуаций, когда его можно переписать с нуля без затрагивания взаимодействующих с ним компонентов. Многие команды разработчиков идут еще дальше: они явным образом планируют, что множество сервисов в долгосрочной перспективе не будет эволюционировать, а будут просто выброшены на свалку.

Веб-сайт Guardian -- хороший пример приложения, которое было спроектировано и построено как монолит, но затем эволюционировало в сторону микросервисов. Ядро сайта все еще остается монолитом, но новые фичи добавляются путем построения микросервисов, которые используют API монолита. Такой подход особенно полезен для функциональности, которая по сути своей является временной. Пример такой функциональности -- специализированные страницы для освещения спортивных событий. Такие части сайта могут быть быстро собраны вместе с использованием быстрых языков программирования и удалены, как только событие закончится. Похожий подход может применяться и в финансовых системах, где новые сервисы добавлялись под открывавшиеся рыночные возможности и удалялись через несколько месяцев или даже недель после создания.

Такой упор на заменяемости -- частный случай более общего принципа модульного дизайна, который заключается в том, что модульность определяется скоростью изменения функционала. Вещи, которые изменяются вместе, должны храниться в одном модуле. Части системы, изменяемые редко, не должны находиться

вместе с быстроэволюционирующими сервисами. Если вы регулярно меняете два сервиса вместе, задумайтесь над тем, что возможно их следует объединить.

Помещение компонент в сервисы добавляет возможность более точного (granular) планирования релиза. С монолитом любые изменения требуют пересборки и развертывания всего приложения. С микросервисами вам нужно переразвернуть только те сервисы, что изменились. Это позволяет упростить и ускорить процесс релиза. Недостаток такого подхода в том, что вам приходится волноваться на счет того, что изменения в одном сервисе сломают сервисы, обращающиеся к нему. Традиционный подход к интеграции заключается в том, чтобы решать такие проблемы путем версионности, но микросервисы предпочитают использовать версионность только в случае крайней необходимости. Мы можем избежать версионности путем проектирования сервисов так, чтобы они были настолько толерантны к изменениям соседних сервисов, насколько возможно.

Опасности при работе с микросервисами

Рассмотрим некоторые сложности, которые привносит использование микросервисов в процесс разработки.

- Сложно понять, где именно должны лежать границы компонентов. Эволюционный дизайн осознает сложности проведения правильных границ и важность легкого их изменения. Когда ваши компоненты являются сервисами, общающимися между собой удаленно, проводить рефакторинг намного сложнее, чем в случае с библиотеками, работающими в одном процессе. Перемещение кода между границами сервисов, изменение интерфейсов должны быть скоординированы между разными командами. Необходимо добавлять слои для поддержки обратной совместимости. Все это также усложняет процесс тестирования.
- Если компоненты не подобраны достаточно чисто, происходит перенос сложности из компонент на связи между компонентами. Создается ложное ощущение простоты отдельных компонент, в то время как вся сложность находится в местах, которые труднее контролировать.
- Фактор уровня команды: новые техники как правило принимаются более сильными командами, но техники, которые являются более эффективными для более сильных команд, необязательно являются таковыми для менее сильных групп разработчиков. Слабые команды всегда создают слабые системы, сложно сказать улучшат ли микросервисы эту ситуацию или ухудшат.

Одна из разумных рекомендаций состоит в том, что не следует начинать разработку с микросервисной архитектуры. Начните с монолита, сохраняйте его модульным и разбейте на микросервисы, когда монолит станет проблемой. (И все же этот совет не является идеальным, т.к. хорошие интерфейсы для сообщения внутри процесса не являются таковыми в случае с межсервисным сообщением.)

Peer-to-Peer

Традиционные клиент-серверные системы предоставляют доступ к ресурсам, расположенным на едином компьютере или на кластере сильно связных серверов. Масштабирование такого централизованного сервиса путём добавления новых компьютеров, на которых они работают, может быть затруднена: сложность администрирования и поддержки работоспособности растёт, к тому же каждому компьютеру нужно устойчивое сетевое соединение. Архитектура “точка-точка” (peer-to-peer, P2P) позволяет выполнять распределённые приложения и обработку данных на обычных персональных компьютерах, связанных через интернет. Подобный подход кажется особенно актуальным в настоящее время, когда разница в производительности и пропускная способность интернет-каналов рабочих станций и серверов не такая большая, как лет 10-20 назад.

Цель -- получить полностью децентрализованный и самоорганизующийся сервис, динамически балансирующий нагрузку на вычислительные ресурсы и хранилища данных между участниками с учётом динамического изменения состава этих самых участников. P2P-системам свойственны следующие характеристики:

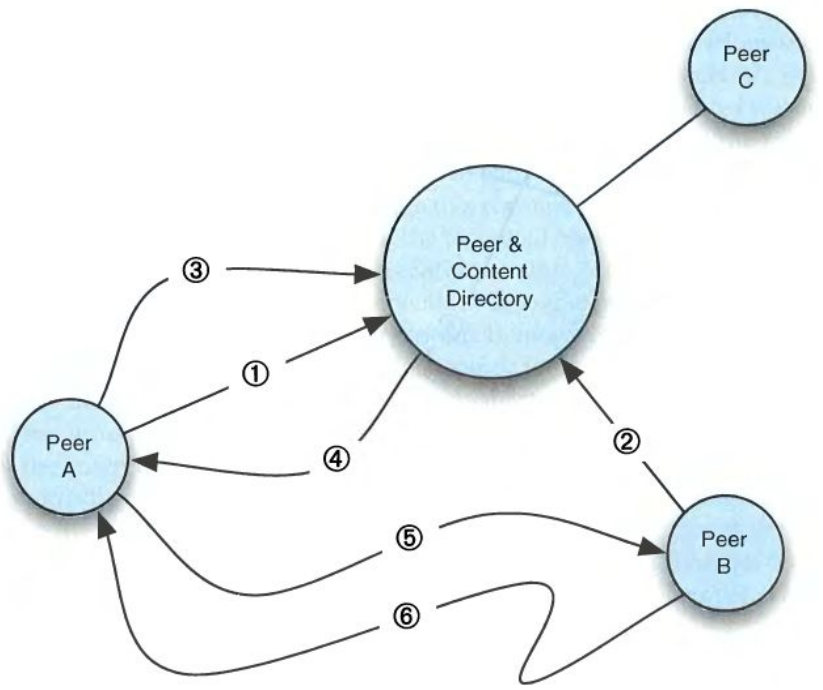
- архитектура гарантирует, что каждый участник привносит свои ресурсы в систему;
- даже если участники задействуют разный объём своих ресурсов, все узлы в P2P системе несут одни и те же обязанности и выполняют одни и те же задачи;
- корректная работа системы не требует и не зависит от существования какого-либо централизованного управления;
- P2P системы могут быть спроектированы таким образом, чтобы предоставлять своим узлам определённые гарантии анонимности;
- ключевым моментом проектирования P2P-систем являются алгоритмы размещения и последовательного доступа к данным, которые позволяют балансировать нагрузку и гарантировать доступность системы без добавления неоправданных накладных расходов.

Рассмотрим несколько вариантов такого рода архитектур.

Гибридный Клиент-сервер/P2P: Napster

Одним из первых приложений, которое стало использовать подобную глобально масштабируемую систему для хранения и получения файлов, стал сервис по скачиванию музыкальных файлов. Napster одновременно показал и применимость, и необходимость подобных решений для обмена файлами. Сервис стал крайне популярен сразу после запуска в 1999. На пике активности он имел несколько миллионов зарегистрированных пользователей и несколько сотен тысяч постоянного онлайна.

Архитектура Napster’а включала в себя централизованные индексы, но сами файлы хранились только на компьютерах пользователей. Способ взаимодействия между узлами сети показан на рисунке ниже.



Пользователи соединялись с индексным сервисом, предоставляли ему информацию о расшариваемых ими файлах, а потом соединялись напрямую с теми узлами, которые хранили нужные им файлы. Успех Napster был как раз в составлении и публичном доступе к огромному и глобально доступному массиву аудиофайлов через интернет.

Архитектурно подобную организацию можно считать гибридом между классическим клиент-сервером и P2P. Для взаимодействия между сервером и клиентом использовался проприетарный протокол, а между клиентами -- HTTP. (Проприетарный протокол, к слову, позволял обмениваться только лишь mp3 файлами.) Индексные сервисы поддерживали единый список файлов, реплицируя его между собой. Алгоритмы составления и поддержания актуальности списков файлов, их репликации были довольно простые, поскольку подразумевалось, что музыкальные файлы не будут изменяться после их добавления в сеть. Гарантий доступности файлов в сети также не было никаких: если узел, владеющий файлом, уходил из сети, предлагалось подождать появления его или кого-то другого, владеющего этим файлом.

Недостатки у такой архитектуры вполне очевидны: если приезжал чёрный фургон и увозил сервер с индексами, всё взаимодействие пиров между собой разваливалось. Что в итоге и получилось, когда халяву прикрыли в связи с многочисленными исками о нарушении авторских прав. И тем не менее, Napster продемонстрировал возможность построения масштабного сервиса, который строил свою работу всецело лишь на данных на компьютерах пользователей, распределённых по всему интернету.

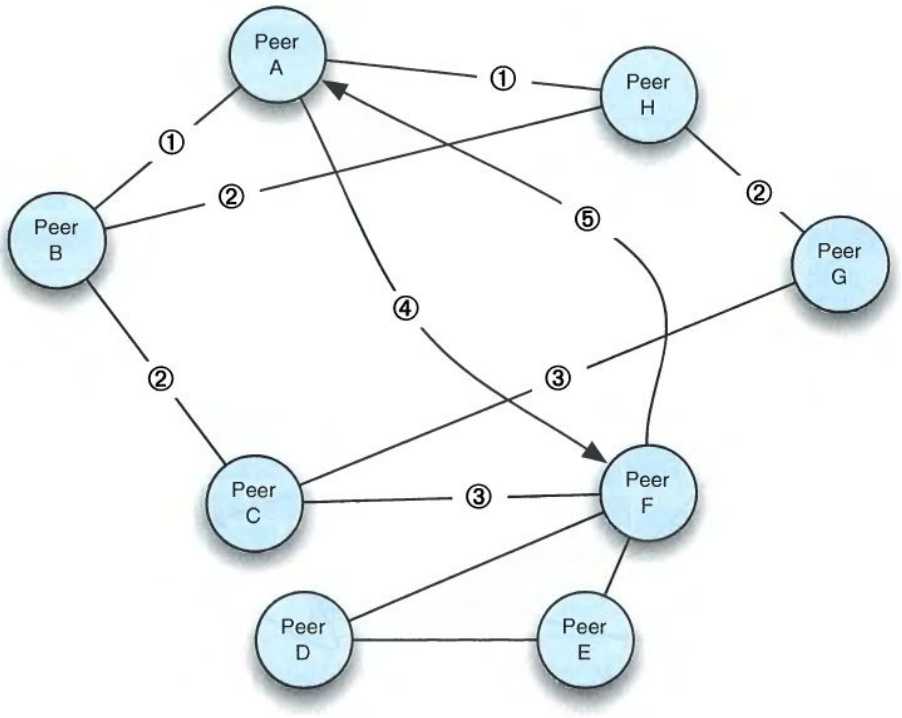
По-настоящему децентрализованный P2P: Gnutella

Решением проблем централизации Napster'а была призвана стать архитектура файлообменной сети Gnutella. В изначальной своей версии Gnutella была полностью

децентрализованной сетью, где все узлы были абсолютно равнозначны и самодостаточны.

Схема работы сети была примерно такова. Когда новый узел входил в сеть, он пытался найти хотя бы одного другого пира (через захардкоженные в клиент адреса популярных пиров, вручную находя адреса на специальных серверах в интернете или даже в IRC-часах), у которого скачивал список известных тому адресов. Ограничивался размер этого списка для каждого узла неким числом N (в ранних версиях реализации протокола не более 5), и именно этими N узлами ограничивалось прямое взаимодействие.

Если, скажем, узел A хотел получить какой-то файл, он рассылал запрос тем пирам, про которых знал (на рисунке ниже это B и H). Если у них не было этого файла, они передавали запрос дальше всем, кого уже знал каждый из них. Так запросы и передавались по сети до тех пор, пока не истекал определённый порог количества пересылок (в ранних версиях реализации протокола не более 7), либо пока не находился узел с искомым файлом. Если такой узел находился, в первых версиях протокола информация об ответе отправлялась по тому же самому пути обратно до инициатора запроса (в ранних версиях в самом запросе не передавалось никаких адресов).



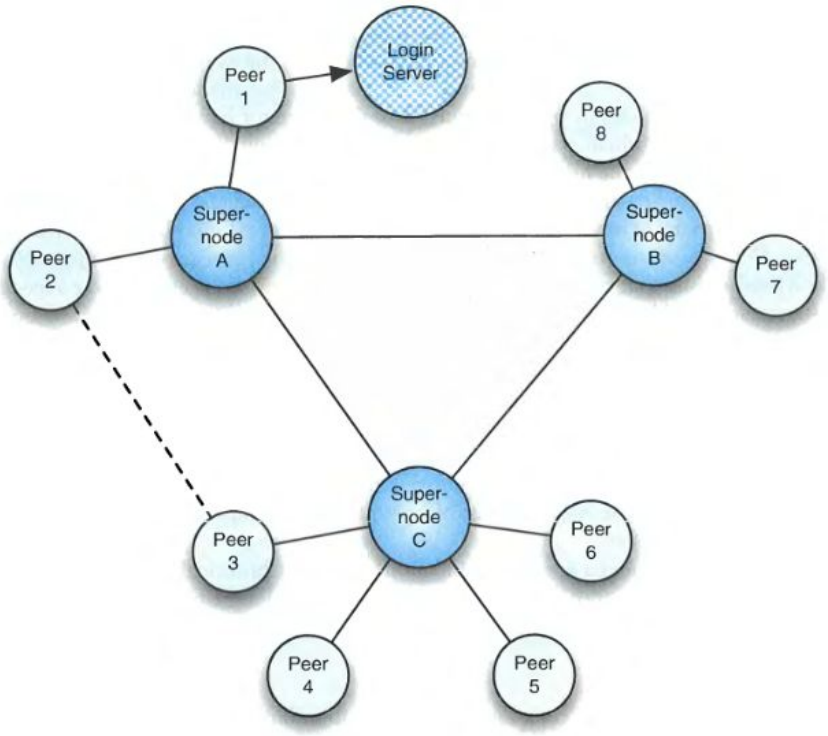
Понятно, что подобная реализация ведёт к экспоненциальному росту числа запросов и соответственно может привести к отказу в обслуживании наиболее “общительных” узлов, что и наблюдалось неоднократно на практике. Чтобы с этим бороться, разработчики усовершенствовали алгоритм, введя правила, в соответствии с которыми запросы могут пересылать далее по графу только определенные узлы -- так называемые выделенные (ultrapeers), остальные узлы (leaves) могут лишь запрашивать последние. Также была введена система кеширующих узлов, да и в сообщениях стали передавать IP-адреса узлов, так что передача файлов стала происходить напрямую по протоколу UDP. Как и у Napster’a, протокол взаимодействия

между узлами был проприетарный, однако прямой обмен музыкой проводился поверх HTTP-запросов.

Несмотря на все недостатки и сложность в работе, в отличие от Napster, сеть подобной архитектуры невозможно просто так закрыть, и поэтому она была долгое время довольно популярна. Говорят даже, что в конце 2007 года на эту сеть приходилось до 40% всего P2P трафика.

Overlayed P2P: Skype

Другой пример P2P архитектуры был у ранних версий Skype. Для начала работы узел логинится на специальный сервер, который говорит ему адрес суперноды, с которой будет взаимодействовать этот рядовой узел.



Когда узел хочет сделать звонок другому пользователю, он делает запрос суперноде. Та, в свою очередь, предаёт запрос конечному узлу, если связана с ним напрямую, или другой суперноде. Так запрос в итоге доходит до получателя. Если оба конечных узла находятся в публичной сети, обмен трафиком во время звонка будет осуществляться напрямую (узлы 2 и 3 на рисунке).

Суперноды предоставляют услуги адресации (между ними синхронизируется и реплицируется список всех пользователей, дубликат списка также хранится на логин-сервере) и перенаправления звонков. Их положение выбирается на основании их сетевой доступности и мощности компьютеров, на которых они запущены. Логин-сервер находился под контролем skype.com, однако суперноды -- обычные пользовательские узлы, которые могли получить этот статус в соответствии с историей их взаимодействия с другими узлами (обязательное условие -- внешний IP-адрес и открытый TCP-порт для Skype). Причём отключить возможность своего компьютера быть супернодой пользователь не мог, со всеми вытекающими последствиями.

Некоторые замечания по архитектуре.

- Проприетарный протокол со встроенным шифрованием данных и один официальный (опять же закрытый) клиент позволяют безопасно передавать данные через промежуточные суперноды.
- Опять же происходит слияние клиент-серверного и P2P подхода, однако теперь же функции “сервера” динамически распределяются между клиентами. Количество супернодов потенциально может регулироваться в зависимости от загруженности сети. Это позволяет сохранить децентрализацию, при этом не заполняя сеть огромным количеством поисковых запросов.
- Подобная маршрутизация через других пиров позволяла делать звонки даже компьютерам, которые не были подключены к интернету (через супернодов в локальной сети).
- По имеющейся информации, после приобретения Skype Microsoft с 2011-2012 года все суперноды перенесены на сервера Microsoft, а обычным пользователям запрещено получение этого статуса. Сделано это всё для большей масшабируемости, централизованной антивирусной проверки пересылаемых файлов (а также файлов по пересылаемым URL-ссылкам).
- С 2014 года протокол признан устаревшим, и теперь в Skype используется Microsoft Notification Protocol.

Больше информации обо всём этом здесь.

Resource Trading P2P: BitTorrent

BitTorrent -- ещё один пример P2P сети, архитектура которой специально создавалась для достижения определённых качеств системы. Основная цель -- поддержка быстрого распространения больших файлов между пирами по запросу. Используемые алгоритмы и архитектура сети ориентирована на то, чтобы максимизировать доступные ресурсы всех участников обмена, тем самым снижая нагрузку на каждого конкретного участника и повышая масштабируемость. Это та самая проблема, которая возникала в Napster и Gnutella, когда появлялся пользователь с большим и очень популярным файлом, и его заваливали запросами.

Подход, реализованный в BitTorrent сети, заключается в распределении частей файла по как можно большему числу узлов. Узел не получает файл как единый ресурс, файл разбивается на небольшие сегменты фиксированного размера, каждый из которых скачивается отдельно, и по готовности всех сегментов из них собирается оригинальный файл. При этом каждый сегмент не только скачивает части файла, но ещё и позволяет скачивать другим заинтересованным узлам уже полученные им сегменты. BitTorrent-клиент на каждом узле определяет, какие сегменты файла скачать следующими, и откуда их нужно скачивать.

Для каждой раздачи все участники взаимодействия знают о том, какие сегменты готовы к скачиванию на каких узлах. Порядок обмена выбирается таким образом, чтобы сначала клиенты обменивались наиболее редкими сегментами: таким образом повышается доступность файлов в раздаче. В то же время выбор сегмента среди наиболее редких случаен, и поэтому можно избежать ситуации, когда все клиенты начинают скачивать один и тот же самый редкий сегмент, что негативно бы отразилось на производительности.

Отметим некоторые особенности архитектуры BitTorrent сети.

- Ответственность за обнаружение нужного контента выносится за пределы протокола обмена. Пользователи могут искать нужные им файлы, как хотят, другими средствами.
- Для контроля процесса распределённого обмена файлами используется выделенный (централизованный) сервер, называемый трекером. Сам трекер не участвует в процессе обмена, он по необходимости снабжает пиров информацией друг о друге.
- С каждой раздачей ассоциируются метаданные, фиксирующие размер сегментов, на которые разбиваются файлы, контрольную сумму каждого сегмента и расположение трекера.
- Те узлы, которые не участвуют в раздачах, лишь скачивая сегменты файлов, могут быть понижены в приоритете, получая файлы по заниженной скорости.
- Каждый узел имеет возможность временно блокировать отдачу другому клиенту. Это делается для более эффективного использования канала отдачи. Кроме того, при выборе, кого разблокировать, предпочтение отдаётся пирам, которые сами передали этому клиенту много сегментов.
- На фрагменты разбиваются не отдельные файлы, а вся раздача целиком, поэтому у личера, пожелавшего скачать лишь некоторые файлы из раздачи, для поддержания целостности фрагментов нередко будет храниться также небольшой объём избыточной (для него) информации.
- В новых версиях протокола были разработаны бестрекерные системы. Отказ трекера в таких системах не приводит к автоматическому отказу всей сети. Начиная с версии 4.2.0 официального клиента, в нём реализована функция бестрекерной работы, базирующаяся на DHT Kademlia. В таких системах трекер доступен децентрализовано, на клиентах, в форме распределённой хеш-таблицы.

Репликация данных

Под процессом репликации данных будем понимать механизм синхронизации копий данных, находящихся на разных компьютерах в сети. Репликация очень широко применяется в совершенно разных задачах, к примеру, кэширование содержимого веб-серверов в браузерах вполне можно считать формой репликации.

Основная мотивация использования механизмов репликации следующая.

- **Повышение производительности**. Указанное выше кэширование данных сервера на клиенте -- по сути средство повышения производительности системы за счёт снижения задержек передачи данных. К тому же часто данные прозрачно синхронизируются между серверами с одинаковым ПО для параллельной обработки. К примеру, для одной DNS записи может быть указан сразу список IP-адресов, и запросы им будут рассылаться по очереди. При этом репликация неизменяемых данных тривиальна и даёт очевидный прирост производительности системы, а вот с изменяющимися данными всё становится гораздо сложнее. Для синхронизации постоянно изменяющихся данных могут

требоваться существенные накладные расходы, поэтому подобного рода механизмы и пользу от них нужно продумывать очень тщательно.

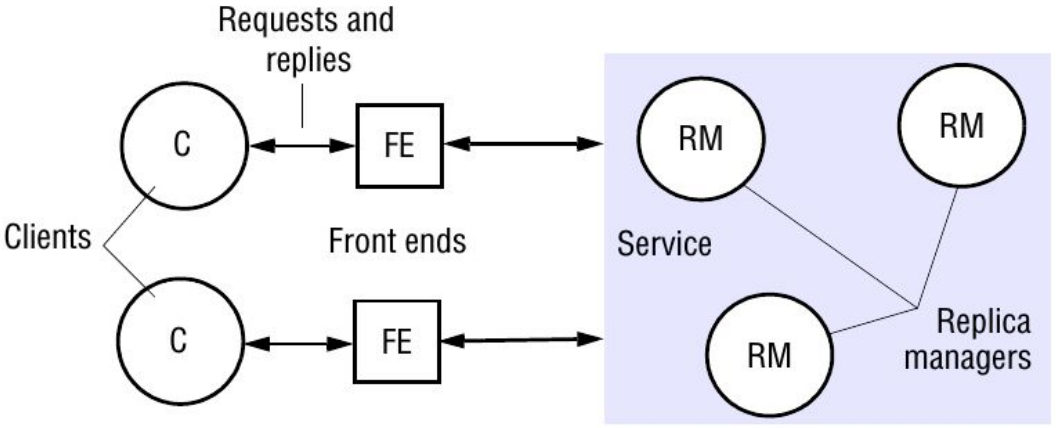
- **Повышение доступности данных**. Для бо́льшей части сервисов крайне важно быть доступными для использования (быть способным дать ответ на запрос за разумное время) как можно ближе к 100% времени их работы. Помимо прочего это значит, что сервер, к которому приходит запрос, должен работать и иметь все необходимые данные для его обработки. Используя несколько серверов и выполняя синхронизацию данных между ними, можно обеспечить доступность сервиса, если один из серверов поломается или случится отказ сетевого соединения с ним. Вероятность отказа всей системы при использовании N серверов с вероятностью отказа каждого из них P будет равна $1 - P^N$. То есть если вероятность отказа одного сервера 5%, то для системы из двух серверов доступность будет $1 - 0.05^2 = 0.9975$, то есть 99.75% времени. В этом смысле кэширование данных не сильно помогает повысить доступность, так как кэшируется лишь только часть данных, и в отсутствие основного сервера их использование весьма ограничено.
- **Отказоустойчивость**. Доступные данные -- не всегда строго корректные данные. Например, они могут быть устаревшие, или при неполадках в сети пользователи могут получать разные версии данных от серверов, между которыми нарушилась связь. Отказоустойчивость подразумевает актуальность данных и корректность пользовательских операций над ними. Часть также под отказоустойчивостью понимают укладывание времени ответа в заданные временные рамки (что не менее важно для систем реального времени, чем корректность ответа). Как и при обеспечении надёжности, своевременная и точная репликация данных на ряд серверов помогает достичь и отказоустойчивости. Если N-1 из N серверов откажут, хотя бы один сможет обрабатывать запросы. Или если в системе допускается подмена N серверов на сервера злоумышленников, система из 2N+1 сервера сможет провести голосование и получить корректный ответ. Тут, разумеется, одной репликацией не обойтись и требуется реализация специальных алгоритмов, но и на основе рассогласованных данных подобные механизмы не работают.

Типичное требование к механизму репликации данных -- прозрачность. То есть клиенты не обязаны знать, что существуют несколько физических копий данных, с которыми они работают. Что происходит с запросом, когда он доходит до сервера, на скольких компьютерах и на каких именно он выполняется, что происходит с результатом, клиента не должно беспокоить.

Модель системы

Будем считать, что наша система состоит из коллекции объектов. Под объектами будем понимать файлы, отдельные элементы данных или Java-объекты. Каждый логический объект реализуется коллекцией физических объектов, которые будем называть репликами. Реплики -- физические объекты, каждый из которых хранится на отдельном компьютере. Они не обязательно все идентичны между собой в каждый момент времени, какие-то реплики уже могли быть изменены, тогда как другие ещё не успели измениться.

Для упрощения будем считать, что в нашей асинхронной системе процессы могут отказывать, только аварийно завершаясь, сетевых сбоев не происходит. Базовая архитектурная модель для управления реплицируемыми данными такова:



С целью обобщения будем описывать архитектурные компоненты в соответствии с их ролями, не подразумевая, что они обязательно должны быть реализованы как разные процессы. В состав модели входят менеджеры репликации (репликаторы) -- компоненты, которые отвечают за хранение реплик и выполнение операций над ними. В стандартной модели клиент-сервера подобные репликаторы могут быть реализованы отдельными серверами. В другом случае подобная компонента может работать как один из процессов клиентского приложения. Например, осуществлять сохранение локальных данных на мобильном устройстве и продолжать (возможно ограниченную) работу приложения при потере связи.

Также будем считать, что менеджер репликации всегда применяет операции к репликам обратимо, то есть при некорректном завершении какой-либо операции данные не останутся в несогласованном состоянии.

В соответствии с рисунком выше, набор репликаторов предоставляет некий сервис клиентам, который для тех выглядит как способ манипуляции некоторыми данными. Каждый клиент посылает серию запросов на выполнение операций на чтение или изменение данных над одним или несколькими объектами. Каждый запрос обрабатывается front-end компонентой. Её роль -- передача сообщения от клиента одному или нескольким репликаторам. Front-end по сути и отвечает за прозрачность процесса репликации для клиента. Он может реализовываться в клиентском адресном пространстве или отдельным процессом.

В общем случае каждый запрос над реплицируемым объектом выполняется в пять стадий. Действия, выполняемые на каждой стадии, могут меняться в зависимости от типа системы. Например, сервис, который поддерживает оффлайн-операции, будет вести себя иначе, нежели отказоустойчивый сервис.

- **Запрос**. front-end посылает запрос одному или нескольким репликаторам. Тут возможны два варианта: либо запрос посылается мультикастом всем, либо направленно одному репликатору, а тот уже в свою очередь распространяет его остальным.
- **Координирование**. Репликаторы координируются, готовясь согласованно выполнить запрос. Они принимают решение, будет ли выполнена требуемая операция (операция может быть отклонена уже на этом этапе), также

принимается решение об упорядочении нового запроса относительно существующих. В большинстве приложений применяется упорядочение FIFO: если front-end инициирует запрос R, а затем запрос R', то каждый корректно работающий репликатор, который выполняет R', должен предварительно выполнить R.

- **Выполнение**. Каждый из репликаторов выполняет запрос (возможно, не фиксируя его результат окончательно, чтобы иметь возможность отменить операцию).
- **Соглашение**. Репликаторы реплик достигают соглашения по поводу результата операции, который будет зафиксирован у всех. Например, в транзакционной системе репликаторы могут на этом этапе принять коллективное решение, зафиксировать или откатить транзакцию.
- **Ответ**. Один или более репликаторов посылает ответ front-end'у, который переправляет запрос клиенту. Например, если реализуется высокодоступный сервис, клиент может переправить клиенту первый пришедший ответ. В случае высоконадёжной системы, если потенциально некоторые репликаторы реплик могут быть подменены злоумышленниками, front-end может дождаться определённого количества ответов и выбрать наиболее популярный.

Как говорилось выше, разные системы могут выполнять разные наборы стадий. Например, система, которая поддерживает работу при разрывах соединения, может хотеть дать пользователю ответ как можно скорее. К примеру, запущенное на мобильном устройстве приложение может и не ждать, пока репликатор на устройстве синхронизируется с репликатором на сервере. При этом отказоустойчивая система будет ждать с ответом до самого конца, пока не будет гарантирована его корректность.

Отказоустойчивые сервисы

Рассмотрим, как можно с помощью репликации строить отказоустойчивые сервисы. Будем считать, что каналы связи надёжные, а репликаторы работают корректно в соответствии с семантикой выполняемой ими логики (то есть делают все операции корректно и не делают очевидных глупостей).

Интуитивно сервис на основе репликации работает корректно, если он доступен даже несмотря на отказы отдельных серверов, и при этом клиенты не видят разницы между работой сервиса над реплицированными данными или при работе лишь одного экземпляра компоненты репликации. Это важное требование, поскольку при неаккуратной реализации при взаимодействии нескольких менеджеров репликации могут возникать странные состояния.

К примеру, рассмотрим наивную систему репликации, в которой есть два репликатора A и B, работающих на разных компьютерах и параллельно поддерживающих состояние двух банковских счетов, x и y. Клиенты на этих двух компьютерах обращаются к локальным репликаторам, но переходят на удалённые, если локальные уходят в отказ. Репликаторы обмениваются обновлениями в фоновом режиме после отправки ответа клиенту. Изначально на каждом счету $0. Клиент 1 обновляет баланс счёта x через свой локальный репликатор B, чтобы тот стал $1, затем пытается обновить баланс y, чтобы тот стал $2, но обнаруживает, что

репликатор B недоступен. Тогда клиент 1 пытается выполнить этот запрос через репликатора A. Теперь клиент 2 читает состояние баланса в своём локальном репликаторе A. Он обнаруживает, что на счету y $2, а на счету x -- $0: обновление от B не пришло, так как тот репликатор помер. Состояние такой системы отражено на рисунке ниже:

Client 1:	Client 2:
$setBalance_B(x, 1)$	
$setBalance_A(y, 2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

Такого рода работа не соответствует интуитивному пониманию работы банковской системы. Клиент 2 должен быть способен получить баланс $1 для x, так как он получил баланс $2 для y, а баланс y был измененён после баланса x. Подобного рода аномалия не произошла бы, если бы запросы клиента обрабатывал только один сервер. Чтобы понять, как можно построить систему на основе репликации без такого рода аномалий, рассмотрим критерии корректности выполнения операций в распределённых системах.

Линеаризуемость и последовательная согласованность

Есть разные критерии корректности реплицируемых объектов. Самый строгий из них -- линеаризуемость. Представим себе реализацию реплицируемого сервиса с двумя клиентами. Обозначим последовательность операций чтения и записи, которую выполняет клиент i, за $\{o_{i1}, o_{i2}, o_{i3}\}$. Будем предполагать, что каждая операция выполняется синхронно, то есть клиенты ждут завершения одной операции, прежде чем отправить запрос на следующую. Если запросы обрабатывает единый сервер, выполняя операции над единым набором данных, он неким образом упорядочит все поступающие операции. Например, в случае двух клиентов это могла бы быть последовательность $\{o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}\}$. Определим критерий корректности для реплицируемых объектов через виртуальное упорядочение операций клиентов, которое не обязательно физически имело место в каком-либо менеджере репликации, однако которое устанавливает корректность выполнения.

Сервис с реплицируемыми объектами называют **линеаризуемым**, если для любого его выполнения есть некое упорядочение последовательности операций, запрошенных клиентами, которое удовлетворяет двум следующим критериям:

- упорядоченная последовательность операций соответствует последовательности выполнения этих операций на единым набором объектов;
- порядок операций в этом упорядочении согласуется с реальными моментами времени, в которые операции были поданы на исполнение.

Определение отражает идею, что для каждого набора клиентских операций есть некий виртуальный канонический порядок исполнения (то самое упорядочение из

определения) над единым виртуальным представлением реплицируемых объектов. И каждый клиент наблюдает состояние объектов, которое согласовано с этим единым виртуальным представлением.

Сервис банковских счетов, который мы рассматривали ранее, очевидно, не является линеаризуемым. Даже несмотря на моменты времени исполнения операций, нет такого упорядочения клиентских операций, которые бы удовлетворили логике работы банковских счетов: если операция обновления одного счёта произошла до операции обновления второго, то изменение первого должно быть наблюдаемо клиентом, если обновление второго наблюдаемо.

Заметим, что линеаризуемость затрагивает только упорядочение индивидуальных операций и не должна поддерживать транзакционность. Линеаризуемая последовательность операций может легко поломать внутреннее представление приложения о целостности, если не применять примитивы синхронизации.

Требование реального времени в определении линеаризуемости крайне желаемо в идеальном мире, поскольку отражает наше представление о том, что клиент хотят получать актуальную информацию. Но использование понятия общего времени в распределённости системе поднимает вопрос о практической реализуемости линеаризуемости, поскольку синхронизировать часы можно лишь с определённой степенью точности.

Более слабое условие корректности -- это последовательная согласованность. Это понятие накладывает требование на порядок выполнения запросов безотносительно времени. При этом первое требование линеаризуемости сохраняется.

Сервис с реплицируемыми объектами называют **последовательно согласованным**, если для любого его выполнения есть некое упорядочение последовательности операций, запрошенных клиентами, которое удовлетворяет двум следующим критериям:

- упорядоченная последовательность операций соответствует последовательности выполнения этих операций на единым набором объектов;
- порядок операций в этом упорядочении согласуется с порядком, в котором они были исполнены каждым отдельным клиентом.

Это определение не накладывает ограничений ни на время, ни даже на порядок всех операций вместе, лишь на порядок, в котором каждый клиент выполнял эти операции у себя. Упорядочение операций может перемешивать последовательности операций каждого конкретного клиента в любом порядке до тех пор, пока порядок операций внутри последовательности каждого отдельного клиента остаётся таким же. По аналогии подобное происходит, когда перемешивают несколько колод карт так, чтобы карты каждой колоды продолжали оставаться в том же порядке относительно друг друга.

Каждый линеаризуемый сервис является последовательно согласованным, так как порядок операций во времени и задаёт последовательность программных вызовов операций. Обратно, однако, неверно. Пример выполнения операций над сервисом, который последовательно согласован, но не линеаризуем, представлен ниже:

Client 1:	Client 2:
$setBalance_B(x, 1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y, 2)$	

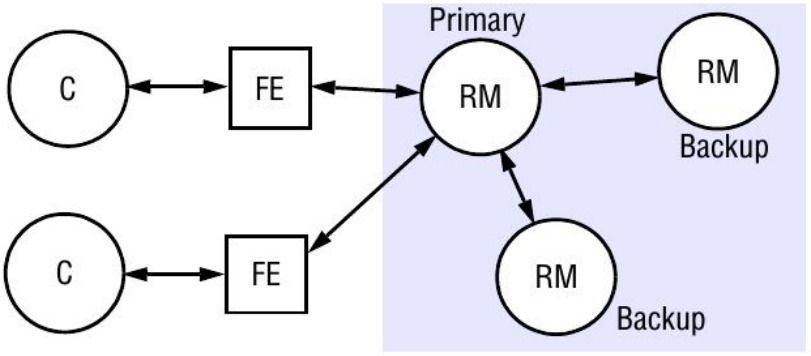
Это исполнение возможно в наивной реализации стратегии репликации, даже если оба из компьютеров A и B не ушли в отказ, но при этом обновление счёта x, которое клиент 1 на компьютере B, не дошло до компьютера A, когда клиент 2 производит чтение. Требование реального времени не удовлетворено, так как $getBalance_A(x)$ -> 0 происходит позже, чем $setBalance_B(x, 1)$, но при этом заданное упорядочение удовлетворяет обоим критериям последовательной согласованности: с точки зрения каждого клиента операции выполнились в нужном порядке.

Пассивная репликация (primary-backup)

В пассивной модели репликации, также называемой (primary-backup), для обеспечения отказоустойчивости выбирается один основной менеджер репликации и один или более вторичных -- репликаторов-бэкапов, или slave'ов. В чистом виде этой модели front-end'ы общаются только с основным репликатором, чтобы получать доступ к сервису. Основной репликатор выполняет операции и отправляет копии обновлённых данных репликаторам бэкапов. Если основной репликатор уходит в отказ, один из бэкапов назначается основным и работа продолжается.

Последовательность событий при обработке запроса клиента следующая.

1. Запрос: front-end присылает основному репликатору репликации запрос на выполнение операции, содержащий уникальный идентификатор.
2. Координация: основной репликатор принимает и обрабатывает каждый запрос в том порядке, в котором он их получил. Проверяет идентификатор запроса, и если он его уже выполнял, просто заново отправляет клиенту результат.
3. Выполнение: основной репликатор выполняет запрос и сохраняет результат.
4. Соглашение: если запрос включал в себя изменение данных, основной репликатор рассылает обновлённое состояние, ответ и идентификатор запроса репликаторам бэкапов. Те в свою очередь присылают подтверждения.
5. Ответ: основной репликатор шлёт front-end'у ответ, который перенаправляет результат операции клиенту.



Система очевидно гарантирует линеаризуемость, если основной репликатор работает корректно, так как он сам собой упорядочивает поступающие операции. Если основной сервер отказывает, система сохраняет линеаризуемость, если один из бэкапов становится новым основным менеджером репликации, и если его конфигурация в точности совпадает с конфигурацией прошлого основного менеджера перед тем, как он помер. В частности:

- основным менеджером репликации становится один из бэкапов (если клиенты начнут разговаривать с двумя разными бэкапами, система будет вести себя некорректно);
- оставшиеся в живых репликаторы приходят к соглашению, какие операции были выполнены до того момента, как появляется новый основной репликатор.

Когда front-end не дожидается ответа, он перепосылает запрос репликатору, который становится основным. Но старый основной мог умереть в любой момент выполнения операции. Если он умер до стадии соглашения (4), то остальные репликаторы не получали от него обновлённого состояния. Если он умер во время стадии соглашения, они могли получить от него новые данные, а если после этой стадии -- то данные были точно получены. Но новый основной репликатор не знает, когда умер старый, так что как только он получает новый запрос, он начинает со стадии 2.

Отметим некоторые особенности рассмотренной модели.

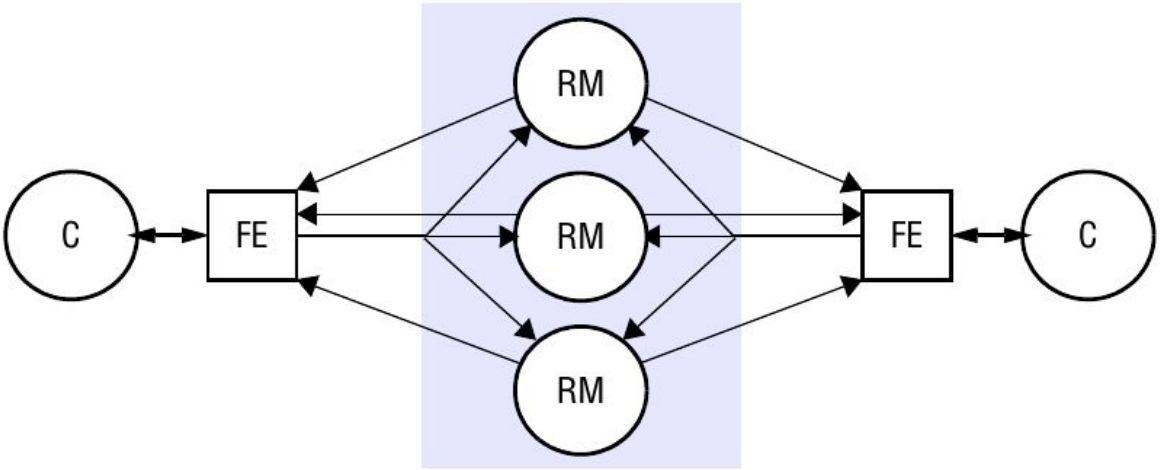
- Подобная пассивная модель репликации может быть использована, даже если основной репликатов ведёт себя недетерминированно (например, ввиду многопоточной реализации). Так как он рассылает репликаторам-бэкапам только результат выполнения операций, а не сами операции, им остаётся только молча применять это состояние.
- Чтобы пережить отказ n процессов, подобным образом реплицируемая система должна иметь n+1 репликатор (и такая система будет неустойчива к проникновению злоумышленников).
- Front-end компоненте не нужно быть слишком умным, чтобы обеспечивать отказоустойчивость. Он должен быть просто в состоянии найти нового основного репликатора, если текущий перестаёт отвечать.
- Пассивная репликация может требовать существенных накладных расходов на коммуникации между репликаторами. К тому же, если основной менеджер

отказывает, задержка ответа ещё увеличивается на время, пока выбирается новый основной.

- В некоторых реализациях подобной модели клиенты могут посылать запросы на чтение напрямую бэкап-репликаторам. Линеаризуемость очевидным образом пропадает, но клиенты всё равно имеют последовательную согласованность.

Активная репликация

В активной модели репликации отказоустойчивых систем менеджеры репликации представляют собой конечные автоматы, организованные в группу. Front-end компоненты рассылают мультикаст-запросы всем репликаторам сразу, те выполняют операции каждый у себя (но одинаково) и высылают ответ. Если какой-то репликатор отказывает, на работе системы это не должно сказываться никак, так как остальные репликаторы продолжают отвечать в нормальном режиме.



Последовательность событий при обработке запроса клиента следующая.

1. Запрос: front-end прикрепляет уникальный идентификатор к запросу и массово рассылает его группе репликаторов. Новый запрос не формируется, пока не будет получен ответ на предыдущий.
2. Координация: система групповой доставки сообщений доставляет запросы каждому репликатору в том порядке, в котором они были отправлены клиентом.
3. Выполнение: каждый репликатор выполняет запрос. Так как они являются по факту конечными автоматами и так как все запросы приходят к ним всем в одинаковом порядке, корректно работающие репликаторы обработают одинаковые запросы идентично.
4. Соглашение: эта фаза не требуется, корректность доставки обеспечивает система передачи сообщений.
5. Ответ: каждый репликатор посылает результат операции front-end’у. Сколько ответов нужно получить тому, чтобы переслать запрос клиенту, зависит от реализуемой логики приложения.

Система обладает последовательной согласованностью. Все корректно работающие репликаторы обрабатывают одни и те же запросы. Надёжность системы рассылки сообщений гарантирует, что все репликаторы получают один и тот же набор

сообщений. Так как они являются конечными автоматами, они все получают одни и те же ответы на одинаковые запросы. Все запросы front-end'а обрабатываются в порядке FIFO, т.к. front-end ждёт ответа на запрос, чтобы отправить следующий.

Но при этом линеаризуемости системы с активной репликацией не достигают: общий порядок, в котором репликаторы обрабатывают запросы клиентов, может не совпадать с порядком во времени, в котором клиенты делают эти запросы.

Задача голосования

Алгоритмами голосования называют алгоритмы, цель которых заставить несколько участников выбрать между собой одного для выполнения некоторой роли. Например, в рассматриваемой выше модели пассивной репликации репликаторы бэкапов должны выбрать среди себя нового основного репликатора при отказе текущего. В общем случае выбранный участник может отказаться от своей новой роли, в таком случае проводится повторное голосование уже без его участия.

У каждого процесса-участника заводится переменная $elected_i$, которая хранит идентификатор выбранного процесса. Когда стартует процесс голосования и процесс становится его участником, он сбрасывает значение этой переменной в специальное значение $\perp$, означающее, что координатор не выбран.

Важное требование к алгоритму в том, чтобы был выбран только один участник. даже если параллельно запущены несколько процессов голосования. В дальнейших примерах без ограничения общности будем считать, что выбирается процесс-координатор с наибольшим идентификатором. В принципе, метрика может быть любая -- минимальная загрузка процессора, пропускная способность канала, расстояние до пользователей и т.п. Дополнительные требования на процессы по завершении голосования таковы:

- E1: процесс-участник p_i имеет значение переменной $elected_i$, равное $\perp$ или P, где P -- не ушедший в отказ до завершения голосования процесс с наибольшим идентификатором.
- E2: все процессы p_i, участвующие в голосовании, либо устанавливают значение $elected_i$, не равное $\perp$, либо отказывают.

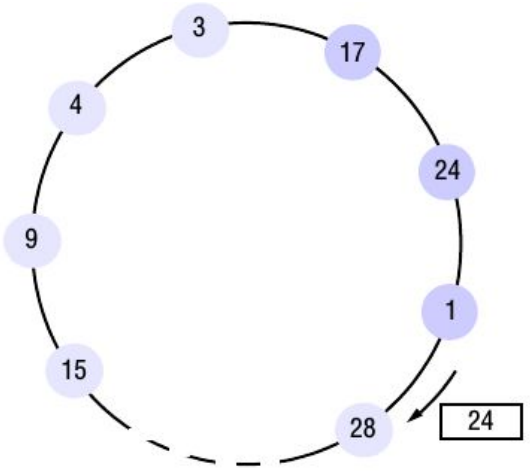
A ring-based election algorithm

Алгоритм Ченга и Робертса (1979) подходит для набора участников, организованных в топологию "кольцо". Каждый участник имеет канал для связи с участником, расположенным по часовой стрелке от него. Будем считать, что система асинхронная и не происходит отказов узлов в сети.

Изначально каждый участник помечает себя как не-голосовавшего. Начать процесс голосования может любой узел, пометив себя как голосовавшего, поместив свой идентификатор внутрь сообщения о голосовании и послав его по своему соседу по часовой стрелке в сети.

Когда узел получает сообщение о голосовании, он сравнивает свой идентификатор с тем, что в сообщении. Если его идентификатор меньше, он пересылает сообщение дальше своему соседу. Если его идентификатор больше и он

в настоящий момент помечен как не-голосовавший, он заменяет в сообщении идентификатор на свой и отправляет сообщение дальше. Но если узлу приходит сообщение о голосовании, когда он уже помечен как голосовавший, и при этом его идентификатор больше, чем тот, что в сообщении, то сообщение он дальше не посылает (чтобы остановить параллельное голосование с худшим кандидатом). В любом случае, когда процесс пересылает сообщение дальше, он помечает себя как голосовавшего.



Если, однако, уже голосовавший процесс получает сообщение, в котором хранится его идентификатор, то это значит, что это сообщение прошло круг, и он является участником с наибольшим идентификатором. Этот узел помечает себя как не-голосовавшего и отправляет своему соседу сообщение о том, что голосование завершено, включая в него свой идентификатор. Получая это сообщение, каждый узел снимает с себя статус голосовавшего, сохраняет у себя идентификатор нового координатора и отправляет сообщение дальше.

Легко заметить, что требование E1 выполняется: все идентификаторы проходят сравнение, так как процесс должен получить собственный идентификатор, чтобы считать себя выбранным. Любой процесс с идентификатором ниже получит от него сообщение, должен будет сравниться и проиграть.

Условие E2 следует из гарантированной доставки сообщений в кольце (предполагаем, что нет отказов). Вводятся статусы голосовавших и не-голосовавших, чтобы остановить параллельные процессы голосования как можно раньше, причём до начала объявления результатов.

Если процесс начинает только один процесс, в самом худшем случае (координатор будет следующим против часовой стрелки от начавшего голосование) будет отправлено 3N-1 сообщений: N-1 чтобы выявить кандидата в координаторы, N на полный круг для подтверждения кандидата и N на полный круг для уведомления всех о завершении голосования.

Подобный алгоритм полезен для понимания работы алгоритмов голосования в общем, однако в реальной жизни применяется редко, т.к. требует довольно сильного требования невозможности отказов узлов. Но если научить узлы перестраивать свою топологию при отказе узлов, вполне можно применять и его.

The bully algorithm

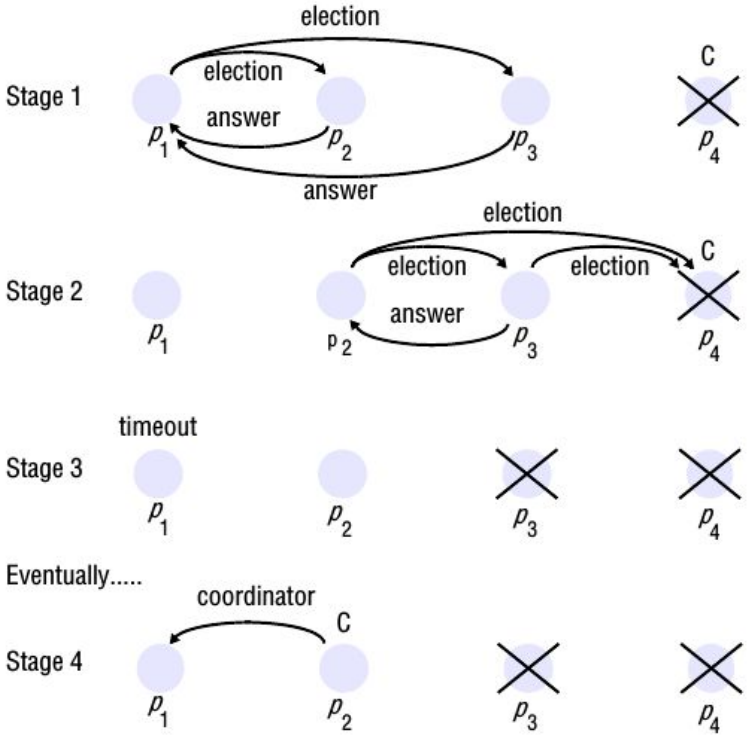
Алгоритм хулигана (1982) допускает отказы процессов во время голосования, подразумевая, что механизм доставки сообщений надёжен. В отличие от прошлого алгоритма тут подразумевается, что система синхронна: для детектирования отказов используются таймауты. Ещё одно отличие в том, что в предыдущем алгоритме узлы почти ничего не знали друг про друга, только могли отправлять сообщения одному соседу. Здесь же, напротив, предполагается, что каждый процесс знает про процессы, чьи идентификаторы выше, чем у него самого, и может взаимодействовать с ними.

Алгоритм подразумевает обмен тремя типами сообщений: (1) сообщение о начале голосования, (2) ответ на это сообщение и (3) сообщение процесса о том, что голосование завершено и он является новым координатором.

Процесс голосования начинается, когда один из процессов посредством таймаутов замечает, что текущий координатор недоступен. Процесс-инициатор, который знает, что у него самый большой идентификатор, отправляет координаторское сообщение всем процессам с более низкими идентификаторами. Процесс-инициатор, который такой уверенностью не обладает, рассылает сообщение о начале голосования всем, кто выше него, и ждёт от них ответа в подтверждение. Если за время T не приходит ни одного ответа, процесс назначает себя координатором и рассылает всем об этом сообщение. Иначе (если пришло хотя бы одно уведомление), он ждёт время T’, пока придёт сообщение от нового координатора о завершении голосования. Если его не приходит, он стартует процесс голосования заново.

Когда процесс стартует, чтобы заменить убитый процесс, он принудительно начинает голосование. Если у него окажется самый большой идентификатор, он становится новым координатором, даже с учётом того, что старый был вполне себе функционален. Собственно, от этого и название алгоритма.

Работа алгоритма показана на примере на рисунке ниже. Процесс p_1 определяет отказ координатора p_4 и начинает голосование (стадия 1 на рисунке). Получив сообщение от p_1, процессы p_2 и p_3 посылают ему ответ и начинают собственные голосования. p_3 шлёт ответ на запрос p_2, но не получает ответа от p_4 (стадия 2), поэтому он решает, что он теперь новый координатор. Но не успев отправить координаторское сообщение, он умирает (стадия 3). Когда у p_1 истекает таймаут T’ (предположим, что это наступает раньше, чем истекает T у p_2), он понимает, что координатора не выбрали, и запускает голосование заново. В итоге будет выбран p_2 (стадия 4).



Алгоритм удовлетворяет требованию E2 за счёт надёжности доставки сообщений. И если процессы не заменяются, то удовлетворяет и требованию E1: двум разным процессам невозможно одновременно решить, что каждый из них координатор, потому как идентификатор одного из них точно больше. Но если процессы могут быть заменены процессами с повторяющимися идентификаторами, E1 не гарантируется. Новый процесс, заменяющий процесс p_i, будет считать координатором себя, а старый, не получив ответа от p_i, -- себя. В итоге всё зависит от того, в каком порядке дойдут до остальных процессов.

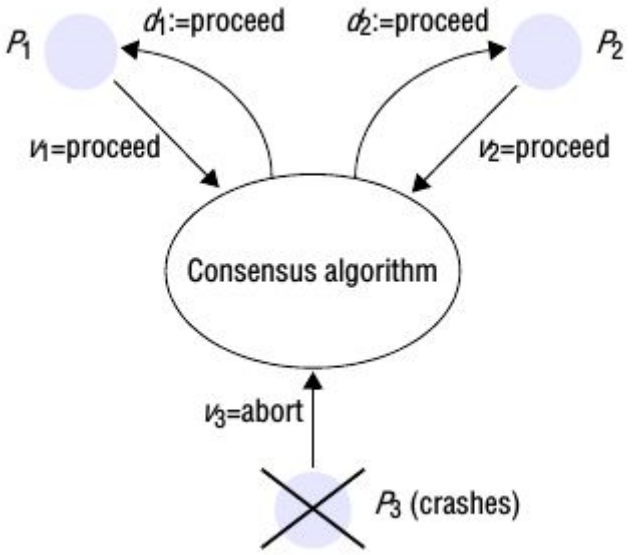
E1 также не достигается, если выбрать недостаточные значения для использующийся таймаутов. В этом смысле медленный или труднодоступный процесс становится неотличим от мёртвого.

В лучшем случае (голосование начинает второй по важности процесс) не требуется сообщений о голосовании (процесс сразу назначает себя координатором сам) и требуется N-2 координаторских сообщений. В худшем случае (голосование начинает самый низкоприоритетный процесс) -- $O(N^2)$ сообщений.

Проблема соглашения

Задача, которую решают алгоритмы соглашения -- договориться о конкретном значении, которое будет принято всеми процессами, при условии, что каждый из них может предложить своё. Например, отдельные компоненты системы управления космическим кораблём должны принять общее решение, продолжать или остановить запуск корабля, процессы должны принять общее решение о выборе нового координатора или процессы-репликаторы должны договориться о едином порядке применения операций над данными.

Будем считать, что канал связи надёжен, но процессы могут аварийно завершаться. Чтобы достичь соглашения, каждый процесс начинает в состоянии *неопределён* и предлагает некое значение v_i из множества D, i из {1..N}. Процессы обмениваются между собой этими значениями, в итоге каждый процесс устанавливает и фиксирует у себя значение переменной d_i, переходя в состояние *определившегося*. На рисунке ниже показан пример трёх процессов, пытающихся решить задачу соглашения.



На каждый шаг алгоритмов консенсуса накладываются следующие требования:

- *завершение*: в конечном итоге каждый процесс устанавливает свою переменную выбора d_i;
- *соглашение*: переменные d_i всех корректных процессов одинаковы: если p_i и p_j корректны и перешли в состояние определившихся, то $d_i = d_j$ (для всех i и j из 1..N);
- *целостность*: если корректные процессы предложили одно и то же значение, каждый корректный процесс в состоянии определившегося тоже выбрал это значение.

В зависимости от приложений требование целостности может быть смягчено: например, что только значение d будет равно значениям лишь некоторых корректных процессов, не обязательно всех.

Особый вариант задачи консенсуса -- задача о византийских генералах (Лэмпорт, 1982). В ней один процесс предлагает некое значение, а остальные пытаются все вместе согласиться или не согласиться с ним. Требования завершения и соглашения остаются неизменными, а требование целостности модифицируется так:

- *целостность*: если процесс-командир корректен, все корректные процессы выбирают предложенное им значение.

Заметим, что это требование ничего не говорит о том, что делать, если процесс-командир некорректен.

Консенсус в синхронных системах

Рассмотрим алгоритм решения задачи консенсуса в синхронной системе, работающей со смягчённым требованием целостности. Алгоритм использует только базовую мультикаст-рассылку сообщений и предполагает, что до f из N процессов могут аварийно завершиться в процессе. Псевдокод представлен ниже:

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f+1)$

B-multicast$(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f+1)$ *rounds*

Assign d_i = *minimum*$(Values_i^{f+1})$;

Пременная $Values_i^r$ содержит набор предложенных значений, известных процессу p_i в начале раунда r. Каждый процесс рассылает только те значения, которые он не видел в предыдущих раундах. Затем он принимает такие же сообщения ото всех других процессов и обновляет свою Values. Каждый раунд ограничен таймером. После f+1 раунда каждый процесс выбирает в качестве результата минимальное значение из Values.

Требование завершаемости очевидно следует из факта синхронности системы. Покажем, что все процессы придут к одному и тому же набору значений в конце последнего раунда. Предположим, что это не так, два процесса имеют в конце различающиеся наборы значений. Пусть, например, p_i имеет значение v, которого нет у процесса p_j (i != j). Это могло бы быть, только если в предпоследнем раунде некий процесс p_k успел отправить значение v процессу p_i, однако аварийно завершился до того, как успеть отправить это значение p_j. Аналогично процесс, который рассылал значение v в предпредпоследнем раунде, успел отправить его p_k, но не успел отправить остальным, поэтому его тогда и не получил p_j. Продолжая таким образом, получим, что в каждом раунде умирал хотя бы один процесс, но мы предположили, что может быть не более f аварийных завершений, а раундов всего было f+1. Так что получаем противоречие.

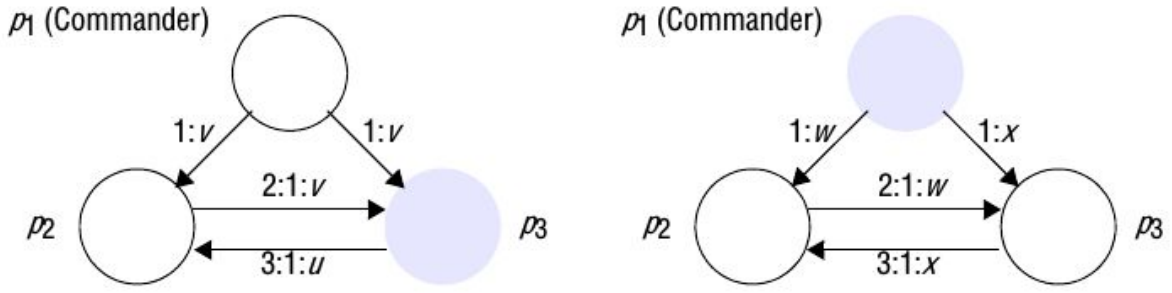
Задача о византийских генералах в синхронной системе

Будем теперь предполагать, что процессы не только могут аварийно завершаться, но и выдавать некорректные значения или вообще не отправлять сообщений (умышленно или нет). Корректно работающие процессы могут обнаруживать отсутствие сообщений посредством таймаутов, но они не могут обнаружить, что процесс аварийно завершился: например, процесс может молчать какое-то время, а потом начать снова отправлять сообщения.

Будем также предполагать, что каналы коммуникаций между парами процессов закрытые и защищённые. Если бы процесс мог читать все передаваемые между остальными процессами сообщения, он мог бы легко увидеть несоответствия, которые исходят от некорректного процесса.

Лэмпорт показал, что в случае трёх процессов задача в общем случае не решается, если сообщения процессами не подписываются. Этот результат был также обобщён для любого случая N <= 3f.

Рисунок ниже показывает два сценария, в которых один из процессов некорректен. На левой части рисунка это один из лейтенантов, на правой -- генерал. На рисунке показаны два раунда сообщений: рассылка сообщений генералом лейтенантам и обмен сообщениями между лейтенантами. Запись “2:1:v” стоит читать как “2 говорит, что 1 говорит v”.



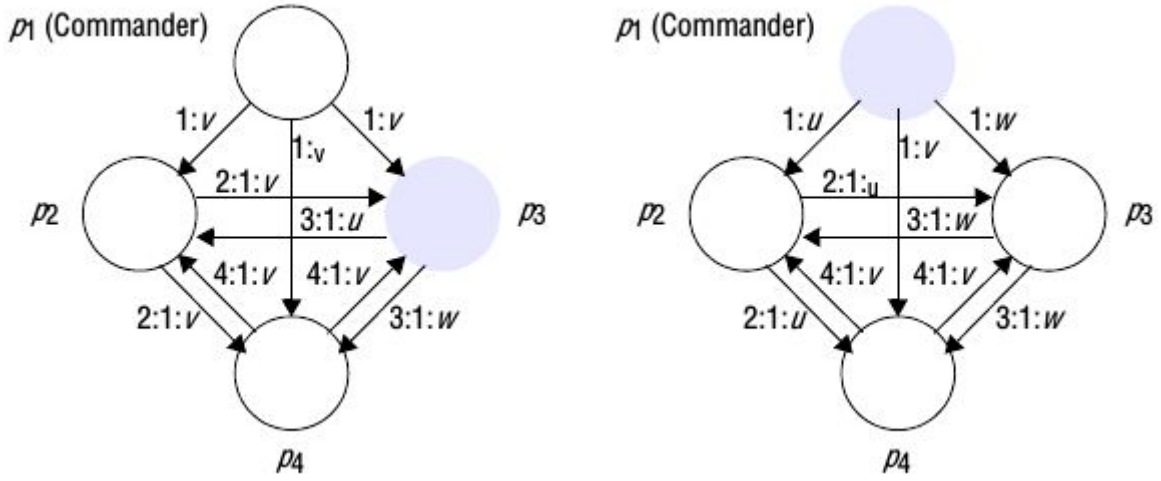
В левом сценарии командир корректно рассылает значение v обоим процессам, и p_2 корректно переправляет его p_3. Но p_3 отсылает p_2 некорректное сообщение u. В этот момент p_2 знает только, что получил два разных значения, он не знает, которое из них пришло от командира. На правой картинке некорректный генерал, однако p_2 ровно так же получает два разных значения, и не знает, какое из них от кого.

Если существует решение этой проблемы, то p_2 должен выбрать значение v, когда командир корректен в соответствии с требованием целостности. Но то же самое верно и для процесса p_3, который выбирает значение от командира, а это противоречит требованию соглашения.

Чтобы доказать невозможность в общем случае N <= 3f, проведём симуляцию, в которой разделим N процессов на три группы n_1, n_2 и n_3, и представим, что три процесса p_1, p_2 и p_3 будут представлять собой поведение этих трёх групп. Те из процессов p_1, p_2 и p_3, которые корректные, симулируют работу корректных генералов: они обмениваются сообщениями внутри группы и посылают другим группам получившееся решение. Процесс p, симулирующий некорректных генералов, посылает

остальным неправильные сообщения. Если бы решение существовало, то это значило бы, что существует решение и для трёх процессов, что неверно.

Чтобы успешно решить задачу консенсуса в синхронной системе, когда процессы обмениваются неподписанными сообщениями, необходимо N >= 3f+1 узлов. Полное доказательство рассматривать мы не будем, рассмотрим лишь случай для f = 1 и N = 4. В этом случае генералы достигают соглашения двумя раундами сообщений: на первом командир рассылает значение всем лейтенантам, на втором лейтенанты посылают сообщения каждый всем остальным.



Каждый лейтенант получает сообщение от командира и два сообщения от коллег-лейтенантов. Просто выбирая самое популярное значение, каждый корректный лейтенант может получить правильный ответ.

Заметим, что этот алгоритм корректно обрабатывает возможность некорректных процессов не отправлять сообщений вовсе. Если процесс не получает всех N-1 сообщений за определённый таймаут, он ровно так же переходит к выбору самого популярного значения.

В общем случае алгоритм для неподписанных сообщений требует f+1 раунда рассылки, что даёт $O(N^{f+1})$ сообщений. Было доказано, что любое детерминированное решение задачи консенсуса (а, следовательно, и задачи о византийских генералах) не может быть получено за меньше, чем f+1 раунд, однако есть варианты алгоритмов, которые требуют меньшего количества отправляемых сообщений.

В случае подписанных сообщений требуется пересылка порядка $O(N^2)$ сообщений.

Решение для асинхронных систем

Приведённые алгоритмы работают только для синхронных систем, то есть предполагают, что обмен сообщениями происходит раундами и если процесс не прислал сообщения в течении определённого таймаута, то он аварийно завершил свою работу. Фишер в 1985 в своей работе доказал, что никакой алгоритм не может гарантированно решить задачу консенсуса (а, значит, и задачу о византийских генералах) в асинхронной системе, даже если будет аварийно завершаться только

один процесс. В асинхронной системе процессы могут реагировать на сообщения произвольное количество времени, так что аварийно завершившийся процесс будет неотличим от медленно работающего. Доказательство Фишера показывает, что всегда найдётся такое продолжение выполнения процессов, что консенсус не будет достигнут.

При этом стоит заметить, что этот результат не значит, что процессы в распределённой асинхронной системе никогда не достигнут консенсуса, если один из них работает некорректно. К тому же, из практики известно, что эта задача вполне решается при определённых условиях. Например, транзакционные системы уже эффективно решают эту задачу десятками лет. Один из обходных путей заключается в частично синхронных системах, которые в большинстве случаев ведут себя как синхронные, но иногда происходят существенные задержки. Кому интересно, можно почитать тут. Другой подход -- маскирование происходящих отказов. Например, у процесса есть персистентное хранилище данных, которое переживает его аварийное завершение. Если процесс умирает, он перезапускается, вычитывает информацию из хранилища, восстанавливает своё состояние и пытается продолжить или перезапустить прерванную задачу. Другими словами, пытается вести себя так, как будто он просто долго, но корректно, работает.

Литература

- R. N. Taylor, N. Medvidovic, E. Dashofy. Software Architecture: Foundations, Theory, and Practice. Wiley, 2009, 740 p.
- G. Coulouris, J. Dollimore, T. Kindberg, G. Blair. Distributed Systems. Concept and Design. Addison Wesley, 2011, 1066 p.