GeekBrains

Архитектура ПО

Методичка к уроку N11

Сервис-ориентированные архитектуры

Оглавление

Введение

Большинство веб-приложений для бизнеса следуют одному и тому же общему потоку запросов: запрос от браузера попадает на веб-сервер, затем на сервер приложений и, наконец, на сервер базы данных.

Хотя эта схема отлично работает для небольшого числа пользователей, при увеличении нагрузки на пользователей начинают появляться узкие места, сначала на уровне веб-сервера, затем на уровне сервера приложений и, наконец, на уровне сервера баз данных. Обычная реакция на узкие места, возникающие при увеличении нагрузки на пользователей, заключается в увеличении масштаба веб-серверов. Это относительно просто и недорого, и иногда помогает решить проблему узких мест. Однако в большинстве случаев при высокой пользовательской нагрузке масштабирование уровня веб-серверов просто переносит узкое место на сервер приложений.

Масштабирование серверов приложений может быть более сложным и дорогостоящим, чем веб-серверов, и обычно просто перемещает узкое место на сервер базы данных, который еще сложнее и дороже масштабировать. Даже если вы сможете масштабировать базу данных, в итоге вы получите топологию в форме треугольника, где самая широкая часть треугольника - это веб-серверы (их легче всего масштабировать), а самая маленькая - база данных (ее труднее всего масштабировать).

Решая эти задачи, еще в 2005 году Питер Роджерс ввел термин «микро-Веб-сервисы" во время презентации на конференции Web Services Edge. Где вопреки традиционному мышлению вокруг архитектуры SOAP-SOA он выступал за небольшие REST-сервисы.

Он сообщил: «Микросервисы создаются с использованием Unix-подобных конвейеров. Сервисы при этом могут вызывать другие сервисы, а также дополнительно к этому, может существовать мультиязычная среда выполнения. Для этого сложные сервисные сборки абстрагируются простыми URI интерфейсами, и при этом любой сервис любого масштаба может быть обнаружен другими сервисами или клиентами».

В итоге развития этой идеи появился архитектурный стиль «Микросервисная архитектура», который лежит в основе всех современных приложений. Вокруг данного подхода сформировалось огромное количество паттернов, библиотек, фреймворков и технологий.

На данном занятии рассмотрим наиболее часто встречающиеся идеи относящиеся к сервис-ориентированным архитектурам.

На этом уроке

- Узнаем что такое монолитная, сервисная, событийная и масштабируемые архитектуры
- Разберём разницу между монолитами, сервисами и микросервисами
- Узнаем что такое Service Mesh и Service Discovery
- Узнаем что такое распределённые хранилища данных и разберём понятия реплицирования и шардирования данных
- Изучим Паттерны сервисных архитектур
    - Обнаружение сервисов
    - Развёртывание сервисов
    - Мониторинг сервисов

Глава 1. Монолиты

Слово монолит (Monolith) описывает известный, часто используемый способ разработки программного продукта.

Команда определяется с набором требований к продукту и делает примерный выбор технологий и архитектуры. Далее создаётся репозитория для исходного кода, вы выделяете общую функциональность и библиотеки (пытаясь сократить количество повторного кода, DRY – don’t repeat yourself!), и вся команда добавляет новый код и функциональность в этот единственный репозитория, как правило, через ветви кода (branch). Код компилируется единым блоком, собирается одной системой сборки, и все модульные тесты прогоняются также сразу, для всего кода целиком. Рефакторинг и масштабные изменения в таком коде сделать довольно просто.

Монолитное приложение - однослойное программное приложение, в котором пользовательский интерфейс и коды доступа к данным объединены в одну программу с одной платформы. Монолитное приложение является автономным и независимым от других вычислительных приложений. Философия дизайна заключается в том, что приложение отвечает не только за конкретную задачу, но и может выполнять каждый шаг, необходимый для выполнения определенной функции.

Монолит - приложения без какой-либо модульной конструкции. Все функции монолитного приложения или основная их часть сосредоточены в одном процессе или контейнере, который разбивается на внутренние слои или библиотеки.

Единый репозиторий кода и одна система сборки естественным образом ведут к выбору основной технологии и языка, которые будут исполнять большую часть работы. Компиляция и сборка разнородных языков в одном репозитории неудобны и чрезмерно усложняют скрипты сборки и время этой сборки. Взаимодействие кода из разных языков и технологий не всегда легко организовать, проще использовать сетевые вызовы (HTTP/REST), что еще сильнее может запутать код, который находится рядом друг с другом, однако общается посредством абстрактных сетевых вызовов.

Тем не менее, для каждой задачи есть свой оптимальный инструмент, и языки программирования не исключение.

Преимущества монолитной архитектуры:

- Простота реализации;

- Согласованность функций;
- Удобный межмодульный рефакторинг;
- Высокая скорость выпуска малой профессиональной командой;
- Устойчивость к сбоям в консистентной среде;
- Лучшая безопасность;
- Простота тестирования;
- Улучшенная производительность приложения.

Однако, если брать разработку в облаке, и зачастую мгновенно и кардинально меняющиеся требования современных Web и мобильных приложений, описанные удобства грозят некоторыми недостатками.

Проблемы монолитной архитектуры

- Трудность опробования инноваций

Монолитная, сильно связанная система, где код может легко получить доступ к другим модулям, и начать использовать их в тех же благих целях не делать ту же работу заново (общие модули и библиотеки), может в результате затруднить создание новых возможностей, не способных идеально вписаться в существующий дизайн.

- Дорогое масштабирование

Монолитное приложение собирается в единое целое и, в большинстве случаев, начинает работать в одном процессе. При возрастании нагрузки на приложение возникает вопрос увеличения его производительности, с помощью или вертикального масштабирования (vertical scaling, усиления мощности серверов на которых работает система), или горизонтального (horizontal scaling, использования более дешевых серверов, но в большем количестве для запуска дополнительных экземпляров (replicas, или instances).

Монолитное приложение проще всего ускорить с помощью запуска на более мощном сервере, но, как хорошо известно, более мощные компьютеры стоят непропорционально дороже стандартных серверов, а возможности процессора и размер памяти на одной машине ограничены. С другой стороны, запустить еще несколько стандартных, недорогих серверов в облаке не составляет никаких проблем. Однако взаимодействие нескольких экземпляров монолитного приложения надо продумать заранее (особенно если используется единая база данных!), и ресурсов оно требует немало – представьте себе запуск

10 экземпляров серьезного корпоративного Java-приложения, каждому из них понадобится несколько гигабайт оперативной памяти. В коммерческом облаке все это приводит к резкому удорожанию.

- Сложность внесения изменений

Отдельные классы и функции крепко связаны друг с другом. Изъятие одного модуля неизбежно вызывает изменения в работе всей системы:

➔ Чтобы внести одно изменение, необходимо пересобрать всё приложение;

➔ Невозможно масштабировать отдельный модуль – придётся переделывать всё приложение;

➔ Разработка ограничена изначально выбранным набором языков программирования, фреймворков и других инструментов. Хочется использовать что-то альтернативное – «извините, у нас такого нет, работайте с тем, что есть»;

➔ Сломался один модуль – работать, скорее всего, перестанет весь сервис;

➔ Сложная структура связей между модулями замедляет выход обновлений. Каждое требует серьезного тестирования на дефекты и регрессии кода (новые ошибки в уже протестированном функционале). Соответственно, в одном обновлении появляется большой объём изменений.

- Сложность понимания системы

Поначалу приложение имеет понятную структуру и быстро развивается. А потом: длинный и сложный код, рамки модулей постепенно стираются, между компонентами возникает все больше неочевидных взаимосвязей.

Но можно и заметить, что качественно сделанная система с разбиением на модули, правильной инкапсуляцией и скрытием внутренних винтиков системы будет не сложнее для понимания, чем сеть из десятков микросервисов, взаимодействующих по сети. Многое зависит от дисциплины и культуры команды.

- Сложности с командой проекта.

Большая, созданная единым монолитом система сложна для понимания для новых членов команды. Нередко размер команды резко растет, требуется срочно создать новую функциональность, и ключевым фактором становится скорость начала работы с ней и ее кодом ранее незнакомых с ней программистов.

Каждому участнику необходимо владеть экспертизой по всем бизнес-функциям, что со временем все труднее. Намного больше требуется времени и для добавления нового программиста в работу – ему потребуется изучить весь объёмный код.

В общем случае стоит признать, что созданная командой (с ее внутренней дисципли- ной и культурой) система скорее будет более прозрачной и понятной в виде микросервисов и качественно разделенных друг от друга репозиториев, чем в виде огромного кода размером в сотни тысяч строк, особенно если новый программист начинает работу над четко определенной задачей в одном микросервисе.

Глава 2. Понятие распределенной системы

Архитектуры независимых компонентов

Программные системы архитектура которых состоит из независимо работающих компонентов, общающихся друг с другом – называются распределёнными.

Распределённая система, это система, для которой отношения местоположений элементов (или групп элементов) играют существенную роль с точки зрения функционирования системы, а следовательно, и с точки зрения анализа и синтеза системы.

Для распределённых систем характерно распределение функций, ресурсов между множеством элементов (узлов) и отсутствие единого управляющего центра, поэтому выход из строя одного из узлов не приводит к полной остановке всей системы.

Основными **преимуществами** распределенных приложений являются:

- хорошая масштабируемость - при необходимости вычислительная мощность распределенного приложения может быть легко увеличена без изменения его структуры, система может легко интегрировать в своей транспортной среде новые вычислительные ресурсы;

- возможность управления нагрузкой - промежуточные уровни распределенного приложения дают возможность управлять потоками запросов пользователей и перенаправлять их менее загруженным серверам для обработки;

- глобальность - распределенная структура позволяет следовать пространственному распределению бизнес-процессов и создавать клиентские рабочие места в наиболее удобных точках.

Распределенные вычислительные системы обладают такими **общими свойствами**, как:

- управляемость - подразумевает способность системы эффективно контролировать свои составные части. Это достигается благодаря использованию управляющего ПО;
- производительность - обеспечивается за счет возможности перераспределения нагрузки на серверы системы с помощью управляющего ПО;

- расширяемость - к распределенным приложениям можно добавлять новые составные части (серверное ПО) с новыми функциями.

Требования к распределенным системам

Чтобы достигнуть цели своего существования – улучшения выполнения запросов пользователя – распределенная система должна удовлетворять некоторым необходимым требованиям. Можно сформулировать следующий набор требований, которым в наилучшем случае должна удовлетворять распределенная вычислительная система.

Открытость.

Все протоколы взаимодействия компонент внутри распределенной системы в идеальном случае должны быть основаны на общедоступных стандартах. Это позволяет использовать для создания компонент различные средства разработки и операционные системы. Каждая компонента должна иметь точную и полную спецификацию своих сервисов. В этом случае компоненты распределенной системы могут быть созданы независимыми разработчиками. При нарушении этого требования может исчезнуть возможность создания распределенной системы, охватывающей несколько независимых организаций.

Масштабируемость.

Масштабируемость вычислительных систем имеет несколько аспектов. Наиболее важный из них для – возможность добавления в распределенную систему новых компьютеров для увеличения производительности системы, что связано с понятием балансировки нагрузки (load balancing) на серверы системы. К масштабированию относятся так же вопросы эффективного распределения ресурсов сервера, обслуживающего запросы клиентов.

Поддержание логической целостности данных.

Запрос пользователя в распределенной системе должен либо корректно выполняться целиком, либо не выполняться вообще. Ситуация, когда часть компонент системы корректно обработали поступивший запрос, а часть – нет, является наихудшей.

Устойчивость.

Под устойчивостью понимается возможность дублирования несколькими компьютерами одних и тех же функций или же возможность автоматического распределения функций внутри системы в случае выхода из строя одного из компьютеров. В идеальном случае это означает полное отсутствие уникальной

точки сбоя, то есть выход из строя одного любого компьютера не приводит к невозможности обслужить запрос пользователя.

Безопасность.

Каждый компонент, образующий распределенную систему, должен быть уверен, что его функции используются авторизированными на это компонентами или пользователями. Данные, передаваемые между компонентами, должны быть защищены как от искажения, так и от просмотра третьими сторонами.

Эффективность.

В узком смысле применительно к распределенным системам под эффективностью будет пониматься минимизация накладных расходов, связанных с распределенным характером системы. Поскольку эффективность в данном узком смысле может противоречить безопасности, открытости и надежности системы, следует отметить, что требование эффективности в данном контексте является наименее приоритетным. Например, на поддержку логической целостности данных в распределенной системе могут тратиться значительные ресурсы времени и памяти, однако система с недостоверными данными вряд ли нужна пользователям. Желательным свойством промежуточной среды является возможность организации эффективного обмена данными, если взаимодействующие программные компоненты находятся на одном компьютере. Эффективная промежуточная среда должна иметь возможность организации их взаимодействия без затрагивания стека TCP/IP. Для этого могут использоваться системные сокеты (unix sockets) в POSIX системах или именованные каналы (named pipes).

Устойчивость распределенной системы связана с понятием масштабируемости, но не эквивалентна ему. Допустим, система использует набор обрабатывающих запросы серверов и один диспетчер запросов, который распределяет запросы пользователей между серверами. Такая система может считаться достаточно хорошо масштабируемой, однако диспетчер является уязвимой точкой такой системы. С другой стороны, система с единственным сервером может быть устойчива, если существует механизм его автоматической замены в случае выхода его из строя, однако она вряд ли относится к классу хорошо масштабируемых систем. На практике достаточно часто встречаются распределенные системы, не удовлетворяющие данным требованиям: например, любая система с уникальным сервером БД, реализованным в виде единственного компьютера, имеет уникальную точку сбоя. Выполнение требований устойчивости и масштабируемости обычно связано с некоторыми дополнительным расходами, что на практике может быть не всегда

целесообразно. Однако используемые при построении распределенных систем технологии должны допускать принципиальную возможность создания устойчивых и высоко масштабируемых систем.

Монолитная архитектура vs **Распределенная архитектура**

Архитектурные стили можно разделить на две основные группы: монолитные (единая единица развертывания всего кода) и распределенные (несколько единиц развертывания, соединенных через протоколы удаленного доступа).

Монолитные:

- Многослойная архитектура
- Конвейерная архитектура
- Микроядерная архитектура

Распределенные:

- Событийно-ориентированная архитектура
- Сервис-ориентированная архитектура
- Архитектура микросервисов

Стили распределенной архитектуры, хотя и являются гораздо более мощными с точки зрения производительности, масштабируемости и доступности, имеют некоторые недостатки и особенности, которые будут рассмотрены ниже.

Первое с чем пришлось столкнуться проектировщикам первых распределенных систем, так это с особенностями сетевого взаимодействия. В отличии от работы системы в рамках одной машины, взаимодействие по сети чаще подвержено отказам и нестабильностям. Поэтому в то время архитекторы, часто делали неверные предположения о дизайне системы основываясь на опыте проектирования монолитных систем. Впервые список наиболее частых заблуждений о распределенных вычислениях появился в компании Sun Microsystems.

Заблуждения о распределенных системах:

Заблуждение №1. Сеть надежна

И разработчики, и архитекторы полагают, что сеть надежна, но это не так. Хотя со временем сети стали более надежными, факт заключается в том, что сети все еще остаются в целом ненадежными. Это важно для всех распределенных архитектур, потому что все стили распределенной архитектуры полагаются на сеть для связи с сервисами и между сервисами. Чем больше система зависит от сети, тем потенциально менее надежной она становится.

Заблуждение №2. Задержки равны нулю

В отличии от взаимодействия в рамках одного процесса операционной системы, отправка сообщений по сети требует намного больше времени, что необходимо учитывать при проектировании сервисов чувствительных к задержкам

Если предположить, что в среднем на один запрос приходится 100 миллисекунд задержки, то объединение 10 вызовов служб для выполнения определенной бизнес-функции добавляет к запросу 1 000 миллисекунд! Знать среднее значение задержки очень важно, но еще важнее знать 95-99-й процентили. Если средняя задержка может составлять всего 60 миллисекунд то 95-й процентиль может составлять 400 миллисекунд! Обычно именно эта "длинная хвостовая" задержка убивает производительность в распределенной архитектуре. В большинстве случаев архитекторы могут получить значения задержки от сетевого администратора

Заблуждение №3. Пропускная способность бесконечна

Пропускная способность обычно не является проблемой в монолитных архитектурах, поскольку, как только обработка идет в монолите, для обработки бизнес-запроса не требуется или почти не требуется пропускная способность. Когда системы разбиваются на более мелкие единицы развертывания (сервисы) в распределенной архитектуре связь с этими сервисами и между ними значительно использует пропускную способность, что приводит к замедлению работы сети, тем самым влияя на задержку и надежность.

Чтобы проиллюстрировать важность этого заблуждения, рассмотрим две службы, показанные на рисунке. Предположим, что служба A управляет списками желаний на сайте, а служба B - профилем клиента. Когда запрос на список пожеланий поступает в службу A, она должна сделать межсервисный вызов в службу B профиля клиента, чтобы получить имя клиента, потому что эти данные необходимы в контракте ответа для списка пожеланий, но служба списка пожеланий на стороне A не имеет этого имени. Служба профиля клиента

возвращает 45 атрибутов общим объемом 500 кб службе списка пожеланий, которой нужно только имя (200 байт). Запросы на элементы списка пожеланий происходят примерно 2 000 раз в секунду. Это означает, что межсервисный вызов от службы списка пожеланий к службе профиля клиента происходит 2 000 раз в секунду. При 500 кб на каждый запрос объем полосы пропускания, используемой для этого одного межсервисного вызова (из сотен, выполняемых в эту секунду), составляет 1 Гб.

Заблуждение №4. Сеть безопасна (защищена)

Безопасность становится намного сложнее в распределенной архитектуре. Как показано на рисунке 9-5, каждая конечная точка каждой распределенной единицы развертывания должна быть защищена, чтобы неизвестные или плохие запросы не попали в эту службу. Площадь поверхности для угроз и атак увеличивается в разы при переходе от монолитной к распределенной архитектуре. Необходимость защищать каждую конечную точку, даже при межсервисном взаимодействии, является еще одной причиной того, что производительность, как правило, ниже в синхронных, высокораспределенных архитектурах, таких как микросервисы или архитектуры на основе сервисов.

Заблуждение №5. Топология никогда не меняется

Это заблуждение относится к общей топологии сети, включая все маршрутизаторы, концентраторы, коммутаторы, брандмауэры, сети и устройства, используемые в общей сети. Архитекторы предполагают, что топология никогда не меняется. Но чаще всего топология меняется, что может приводит к неустойчивости сети.

Заблуждение №6. Администратор всегда один

Если сеть достаточно большая, то как правило обслуживанием её занимаются несколько администраторов. Данное заблуждение следует учитывать при сборе сведений о сети.

Заблуждение №7. Каналы связи ничего не стоят

Иногда архитекторы ошибочно полагают, что необходимая инфраструктура уже создана и достаточна для выполнения простейшего REST-запроса или разделения монолитного приложения. Распределенные архитектуры обходятся значительно дороже монолитных, в первую очередь из-за увеличения потребностей в дополнительном оборудовании, серверах, шлюзах, брандмауэрах, новых подсетях, прокси-серверах и так далее.

Приступая к созданию распределенной архитектуры, следует проанализировать текущую топологию сети с точки зрения емкости, пропускной способности, задержек и зон безопасности. Конечно часть этих вопросов может быть решена, если система развертывается в облачной инфраструктуре.

Заблуждение №8. Сеть гомогенная

Данное заблуждение заключается в том, что в реальности сеть редко строится на оборудовании и технологиях одного производителя. Это может приводить к тому, что оборудование одного производителя плохо интегрируется с оборудованием другого производителя. Как и с заблуждением №7 следует учитывать, что при использовании облачного окружения данной проблемой можно пренебречь.

Помимо восьми заблуждений распределенных вычислений, описанных ранее, существуют и другие вопросы и проблемы распределенной архитектуры, которые отсутствуют в монолитных архитектурах.

Глава 3. Service-based (сервис-ориентированные) архитектуры SOA и Микросервисы

Архитектура микросервисов и SOA считаются архитектурами, основанными на сервисах, что означает, что они представляют собой архитектурные модели, в которых большое внимание уделяется сервисам как основному компоненту архитектуры, используемому для реализации и выполнения бизнес- и не бизнес-функций. Хотя микросервисы и SOA - это совершенно разные архитектурные стили, у них много общих черт.

Одна общая черта всех архитектур на основе сервисов - это то, что они, как правило, являются распределенными архитектурами, то есть доступ к компонентам сервисов осуществляется удаленно с помощью какого-либо протокола удаленного доступа - например, Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Advanced Message Queuing Protocol (AMQP), Java Message Service (JMS), Microsoft Message Queuing (MSMQ), Remote Method Invocation (RMI) или .NET Remoting.

Распределенные архитектуры имеют значительные преимущества перед монолитными и многоуровневыми архитектурами, включая лучшую масштабируемость, лучшую развязку и лучший контроль над разработкой, тестированием и развертыванием. Компоненты в распределенной архитектуре, как правило, более автономны, что позволяет лучше контролировать изменения и упрощает обслуживание, что, в свою очередь, приводит к созданию более надежных и быстро реагирующих приложений.

Распределенные архитектуры также способствуют созданию более свободно связанных и модульных приложений.

В контексте архитектуры, основанной на сервисах, модульность - это практика инкапсуляции частей приложения в самостоятельные сервисы, которые можно индивидуально проектировать, разрабатывать, тестировать и развертывать с минимальной или нулевой зависимостью от других компонентов или сервисов приложения. Модульные архитектуры также поддерживают концепцию предпочтения перезаписи над обслуживанием, позволяя рефакторить или заменять архитектуру небольшими частями с течением времени по мере роста бизнеса - в отличие от замены или рефакторинга всего приложения с помощью подхода "большого взрыва".

К сожалению, очень немногие вещи в жизни бесплатны, и преимущества распределенных архитектур не являются исключением. Компромиссы, связанные с этими преимуществами, это прежде всего, повышенная сложность и стоимость.

Поддержание контрактов на обслуживание, выбор правильного протокола удаленного доступа, борьба с неответом или недоступностью сервисов, защита удаленных сервисов и управление распределенными транзакциями - вот лишь некоторые из многих сложных вопросов, которые вам придется решать при создании архитектуры на основе сервисов.

SOA

Следующим логическим продолжением технологии Web-сервисов стала архитектура, ориентированная на сервисы (службы) (SOA). Несмотря на то, что SOA придумана в конце 1980-х. Настоящее влияние и реализацию получала благодаря развитию распределённых вычислительных узлов – облачных вычислений.

В основе SOA лежат **принципы**:

- многократное использование функциональных элементов информационных технологий;
- ликвидация дублирования функциональности в программной системе;
- унификация типовых операционных процессов;
- обеспечение перевода операционной модели компании на централизованные процессы;
- функциональная организация бизнеса на основе промышленной интеграционной ИТ платформы.

Компоненты программы в SOA могут быть распределены по разным узлам сети, и предлагаются как независимые, слабо связанные, заменяемые сервисы-приложения. Программные комплексы, разработанные в соответствии с SOA, часто реализуются как набор веб-сервисов, интегрированных при помощи известных стандартных протоколов (SOAP, WSDL, REST, gRPC и т. п.).

Архитектуру этого типа можно отнести к архитектуре независимых компонент. Основное достоинство такого рода сервисов – это простота создания. Упрощается и взаимодействие между компонентами, поскольку основным транспортным протоколом является http. Приложение может без проблем работать не только в Intranet-, но и в Internet-сетях. Кроме того, эта технология не

привязана к какой-либо одной платформе, что открывает возможность создания распределенных приложений в гетерогенных средах.

Ее появление является следствием новых задач и потребностей, возникающих при создании и эксплуатации современных информационных систем:

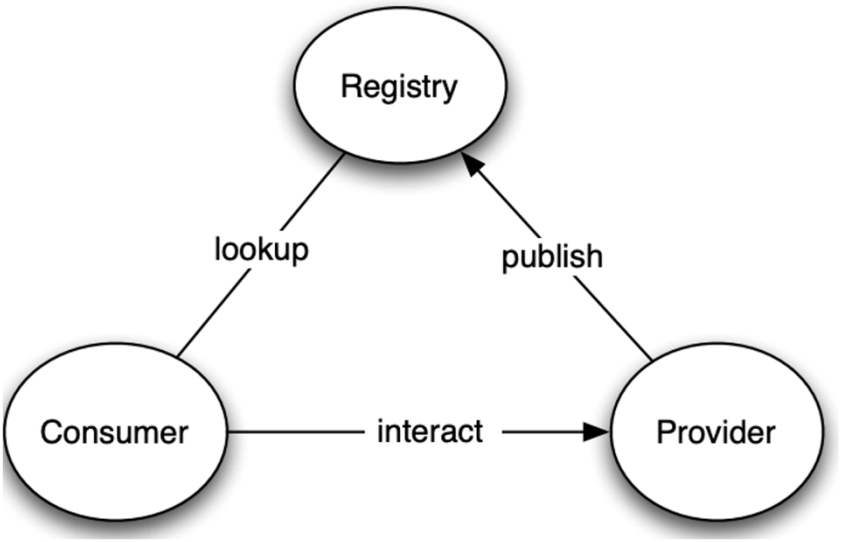
- оперативное реагирование на изменение требований и быстрая адаптация к новым задачам;
- оптимизация управления бизнес-процессами;
- эффективное обеспечение внешних взаимодействий.

К основным **характеристикам SOA**, отличающим ее от других архитектур информационных систем, следует отнести:

- распределенность;
- слабосвязанные интерфейсы (отсутствие жестких связей между элементами системы), что упрощает конфигурирование системы и координирование работы ее элементов;
- базирование архитектуры на международных открытых стандартах;
- возможность динамического поиска и подключения нужных функциональных модулей;
- построение систем с расчетом на процессы с использованием сервисов, ориентированных на решение определенных бизнес-задач.

Три сущности SOA:

1. Провайдер определяет сервисы и публикует их в реестре
2. Потребитель может искать сервис и взаимодействовать с провайдером
3. Реестр собирает сервисные метаданные.



Сущности SOA

Особенности SOA

SOA имеет ряд характеристик, которые выделяют его среди других архитектурных стилей, в первую очередь:

1) SOA способствует использованию открытых стандартов и интерфейсов для достижения функциональной совместимости и независимости от местоположения;

2) службы и процессы разработаны явным образом для функционирования как внутри организации, так и между организациями;

3) SOA требует четких описаний предлагаемой службы;

4) службы и процессы разработаны таким образом, чтобы отражать реальную деятельность бизнеса;

5) представление службы использует бизнес-описание для задания контекста (т.е. бизнес-процесс, цель, правило, политика, интерфейс службы и компонент службы);

6) SOA требует надлежащего руководства представлением и реализацией службы;

7) композиция служб используется как средство реализации бизнес-процессов;

8) SOA устанавливает критерии, позволяющие потребителям служб определять, была ли служба правильно и полностью выполнена в соответствии с описанием службы.

9) сервисы публикуются таким образом, что они позволяют разработчикам быстро находить их и повторно использовать для сборки новых приложений.

По сути, SOA можно свести к нескольким идеям, причём архитектура не диктует способы их реализации:

- Сочетаемость приложений, ориентированных на пользователей.
- Многократное использование бизнес-сервисов.
- Независимость от набора технологий.
- Автономность (независимые эволюция, масштабируемость и развёртываемость).

Архитектура SOA

SOA - представляет собой набор архитектурных принципов и архитектурный стиль проектирования программного обеспечения, не зависящих от технологий и продуктов, в котором бизнес-системы и ИТ-системы разрабатываются с точки зрения сервисов, доступных через интерфейс, а также результатов действий этих сервисов.

SOA - это общекорпоративный подход к разработке программного обеспечения для компонентов приложений, который использует преимущества многократно используемых программных компонентов или сервисов (служб).

Служба - это логическое представление набора действий, порождающих заданные результаты; служба является автономной, может состоять из других служб, при этом потребители данной службы не обязаны знать ее внутреннюю структуру.

В рамках SOA "сервис - служба" является основным элементом для сборки и интеграции информационных систем, которые способны удовлетворять различным требованиям при решении той или иной задачи. SOA позволяет осуществлять взаимодействие между бизнесами без необходимости выделять характерные особенности любого конкретного бизнес-домена. Использование архитектурного стиля SOA позволяет повысить эффективность разработки информационных систем, интеграции и повторного использования ИТ-ресурсов. Кроме того, использование архитектурного стиля SOA может помочь

информационным системам гибко и быстро реагировать на постоянно меняющиеся потребности бизнеса.

Каждый сервис состоит из кода и интеграции данных, необходимых для выполнения определенной бизнес-функции - например, проверки кредитоспособности клиента, входа на веб-сайт или обработки заявки в банке на ипотеку.

SOA предлагает модульный подход к разработке программного обеспечения, базирующийся на обеспечении удаленного по стандартизированным протоколам использования распределённых, слабо связанных, легко заменяемых компонентов (сервисов) со стандартизированными интерфейсами.

Интерфейсы сервисов обеспечивают свободную связь, что означает, что их можно вызывать, практически не зная, как реализована интеграция. Благодаря такой свободной связи и способу публикации сервисов команды разработчиков могут экономить время, повторно используя компоненты в других приложениях в масштабах предприятия. Это одновременно и преимущество, и риск. В результате общего доступа к шине корпоративных служб (ESB), если возникают проблемы, они могут затронуть и другие подключенные службы.

Также, интерфейсы компонентов в сервис-ориентированной архитектуре инкапсулируют детали реализации от остальных компонентов, таким образом обеспечивая комбинирование и многократное использование компонентов для построения сложных распределённых программных комплексов, обеспечивая независимость от используемых платформ и инструментов разработки, способствуя масштабируемости и управляемости создаваемых систем.

SOA предлагает **существенные преимущества** по сравнению с другими архитектурами:

- Повышение гибкости бизнеса, ускорение выхода на рынок: возможность сборки приложений на основе многоразовых интерфейсов сервисов позволяет быстрее реагировать на новые коммерческие возможности, поскольку не требуется тратить время на повторное создание и интеграцию интерфейсов в каждом новом проекте.
- Возможность применения имеющихся функций на новых рынках: хорошо спроектированная SOA позволяет разработчикам легко и просто предоставлять доступ к специализированным функциям вычислительных платформ в новых средах и на новых рынках. Например, многие компании использовали SOA для открытия доступа к функциям финансовых систем

на основе мейнфреймов через Интернет. В результате их клиенты получали возможность работать с процессами и данными, которые раньше были доступны исключительно в рамках взаимодействия с сотрудниками компании или бизнес-партнерами.

- Расширение возможностей совместной работы между бизнес-пользователями и ИТ-специалистами: сервисы SOA можно описывать с помощью бизнес-терминологии (например, «создать предложение по страхованию» или «вычислить рентабельность основного оборудования»). В результате аналитики могут с большей эффективностью взаимодействовать с разработчиками по важным вопросам, таким как область действия бизнес-процесса, описываемого сервисом, или последствия изменения процесса для бизнеса.

В центре этой инфраструктуры находится **технология XML**:

- XML является фундаментом практически всех стандартов web-сервисов, в том числе XML Schema, SOAP, WSDL (Web Services Description Language) и UDDI (Universal Description, Discovery, and Integration). Эти стандарты опираются на основополагающую концепцию основанных на XML представлений - поддерживаемый во всем мире формат обмена информацией между провайдерами сервисов и инициаторами запросов в SOA;
- Использование XML решает проблему работы с различными форматами данных в различных приложениях, работающих на разных платформах;
- Преимущество XML заключается в простоте представления, являющегося по своей природе текстовым, гибким и расширяемым.

Реестры сервисов - Service Registry

Реестр сервисов представляет собой каталог сервисов, доступных в системе SOA. Он содержит физическое месторасположение сервисов, версии и их срок действия, а также другую полезную информацию

Реестр сервисов является одним из основных строительных блоков архитектуры SOA. Его **роль**:

- Реестр сервисов реализует SOA слабое связывание. Храня месторасположения оконечных точек сервисов, он устраняет тесное связывание, приводящее к жесткой привязке потребителя к провайдеру. Он также облегчает потенциальные сложности замены одной реализации сервиса другой при необходимости.

- Реестр сервисов позволяет системным аналитикам исследовать корпоративный портфель бизнес-сервисов. Исходя из этого, они могут определить, какие сервисы доступны для автоматизации процессов с целью удовлетворения актуальных бизнес-потребностей, а какие нет. Это в свою очередь позволяет узнать, что нужно реализовать и добавить в портфель, формируя каталог доступных сервисов.

- Реестр сервисов может выполнять функцию управления сервисами, обязывая подписывающиеся сервисы быть согласованными. Это помогает гарантировать целостность руководства (governance) сервисами и стратегий.

Преимущества и недостатки SOA

К **преимуществам** SOA можно отнести:

- Повторное использование общих сервисов со стандартными интерфейсами расширяет возможности бизнеса и технологий и снижает затраты;

- Сервисы автономны и доступны через формальный контракт, что обеспечивает свободную связь и абстракцию;

- Поскольку протоколы и форматы данных основаны на отраслевых стандартах, поставщик и потребитель сервиса могут быть созданы и развернуты на разных платформах;

- Сервисы взаимодействуют с другими приложениями, используя общий язык, что означает, что они не зависит от платформы;

- Сервисы могут быть гранулированы для обеспечения конкретной функциональности, а не повторять функциональность в нескольких приложениях, что устраняет дублирование;

- Легко редактировать и обновлять любой сервис;

- Сервисы имеют одинаковую структуру запросов к данным хранящимся в разных источниках, что позволяет потребителям каждый раз получать доступ к данным независимо от смены источника данных;

- Сервисы обычно имеют небольшой размер по сравнению с полноценным приложением, поэтому их легче отлаживать и тестировать;

- SOA позволяет повторно использовать сервис существующей системы, при постепенном создании новой системы;

- Есть возможность быстро подключить новые или модернизировать существующие сервисы, чтобы реализовать новые бизнес-требования;
- Можно оперативно повысить производительность, функциональность службы и легко выполнить обновление системы за счёт настроенной инфраструктуры;
- Можно разрабатывать новые бизнес-функции без замены существующих сервисов и приложений;
- Небольшие сервисы более надежны по сравнению с большими приложениями, содержащим много кода.

Вот **недостатки** использования сервис-ориентированной архитектуры:

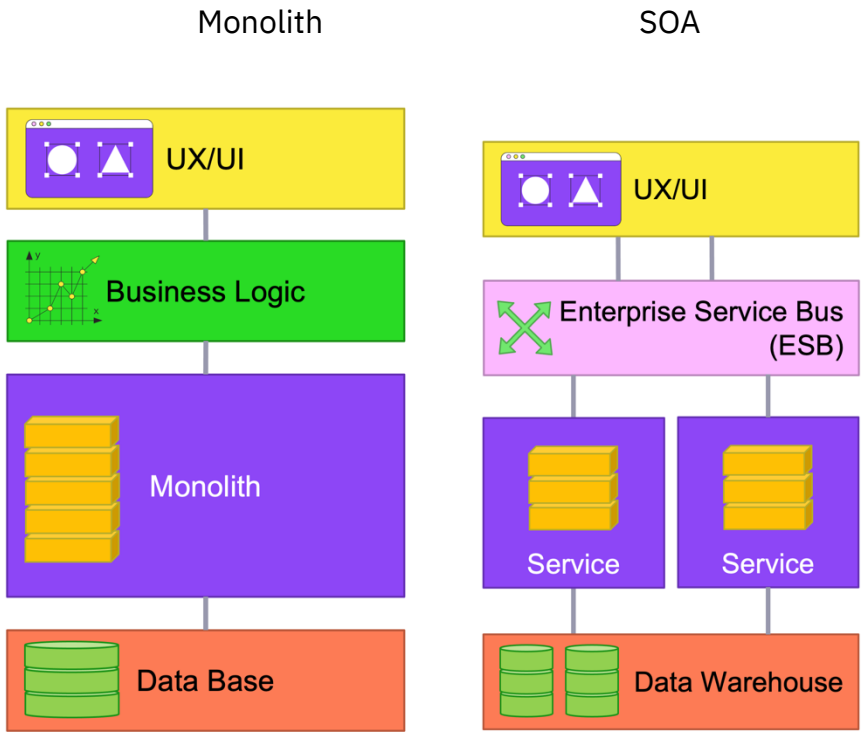
- Перед отправкой в сервис все входные данные должны быть проверены, а проверка входных данных выполняется при каждом взаимодействии сервисов, что снижает производительность, так как увеличивает нагрузку и время отклика;
- Когда сервис взаимодействует с другими сервисами, это приводит к большим издержкам, что увеличивает время отклика;
- Некоторым веб-службам необходимо часто отправлять и получать сообщения и информацию, поэтому они легко достигают миллиона запросов в день;
- SOA не подходит для решений ориентированных на насыщенный графический интерфейс (GUI), так как SOA потребуется интенсивный обмен данными между GUI и множеством сервисов;
- Сложное управление сервисами (развёртывание, безопасность, мониторинг), так как создание системы, ориентированной на повторное использование, приводит также к созданию огромного количества связей между компонентами;
- SOA требует высоких начальных инвестиционных затрат, как с точки зрения компетенций человеческих ресурсов, так и с точки используемых технологий.

Разница между монолитами, сервисами и микросервисами

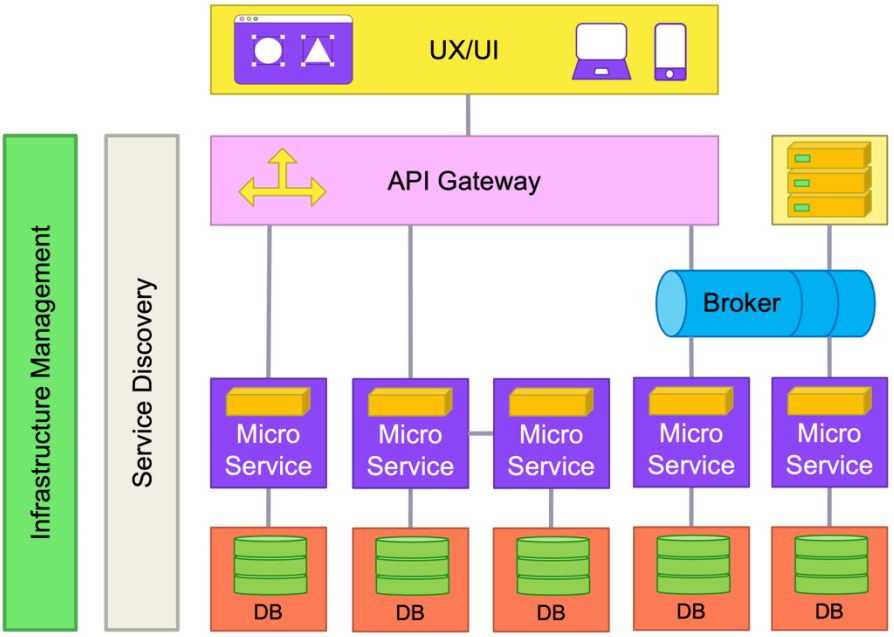
Чтобы уточнить отличия микросервисов от других архитектур, их чаще всего сравнивают с монолитной архитектурой и сервис-ориентированной архитектурой (service-oriented architecture, SOA).

Микросервисное приложение состоит из множества мелких независимых и слабо связанных между собой сервисов, в то время как в монолите все его компоненты тесно взаимосвязаны и работают как единый сервис. Помимо прочего, это значит, что если какой-то один процесс в приложении с монолитной архитектурой становится более востребованным, приходится масштабировать всё приложение в целом. Сбой в каком-то одном процессе может поставить под угрозу всю систему. Наконец, такая сложность ограничивает возможности модернизации и затрудняет внедрение новых идей.

Отличия микросервисов от SOA (рисунок) не столь очевидны. Можно пойти по сложному пути и перечислить множество технических деталей, в том числе связанных с ролью сервисной шины (ESB), но можно поступить проще и оценить уровни, на которые распространяются эти архитектуры. Если SOA — это архитектура уровня предприятия, призванная стандартизировать взаимодействие всех служб, то микросервисы относятся к какому-то одному конкретному приложению.



MSA



Пример сравнения монолитной, SOA и микросервисной архитектур.

Разумеется, не бывает идеальных решений, и у микросервисов тоже есть слабые места. Главный недостаток микросервисов кроется в самой их природе: распределённая система со множеством независимых элементов сложнее как в организационном, так и в архитектурном плане. Усложняется управление командами разработки и развёртывания, и здесь не обойтись без методологий Agile и DevOps. Распределённый доступ к сервисам означает увеличение сетевых задержек и потенциальных сбоев, с чем можно бороться путём асинхронности и сокращения числа вызовов.

Не стоит забывать и о необходимости обеспечивать согласованность приложения: из-за децентрализации и модульности возможно возникновение неконсистентности данных, также приводящее к сбоям и недоступности приложения в целом. В этом случае стоит искать компромиссы между доступностью сервисов и консистентностью.

Сложности, присущие любой, даже самой простой распределённой системе:

- Сетевые вызовы часто приносят с собой неизвестные заранее задержки (latency), и для быстрой работы приходится выполнять работу асинхронно (asynchronous), заранее не зная, в какой момент приходит ответ от остальных компонентов системы.
- Сложность асинхронной, распределенной, динамично развивающейся системы в разы превышает сложность единого, монолитного приложения.

- Разделить систему на микросервисы зачастую очень сложно, если в компании «все делают все», и нет четко очерченных бизнес-областей.

В таких случаях вместо микросервисов получается «распределенный монолит» (distributed monolith), неповоротливая, сложная система, вместо которой мог бы иметь место более эффективный монолит.

Вы можете совершенно спокойно развернуть классическое, монолитное приложение Enterprise Java в контейнере под управлением Kubernetes, получив многие преимущества без излишней сложности.

Микросервисы решают этот вопрос именно с помощью своего размера. Запустить небольшой микросервис проще, ресурсов требуется намного меньше, а самое интересное, увеличить количество экземпляров можно теперь не всем компонентам системы и сервисам одновременно (как в случае с монолитом), а точечно, для тех микросервисов, которые испытывают максимальную нагрузку. Kubernetes делает эту задачу тривиальной.

Если говорить о глобальных **отличиях SOA и архитектуры микросервисов**, то:

SOA — это концепция, охватывающая предприятие в целом. Она позволяет открывать доступ к существующим приложениям с помощью слабо связанных интерфейсов, каждый из которых соответствует конкретной бизнес-функции. В результате приложения в одной части предприятия могут повторно использовать функции других приложений.

Архитектура микросервисов — это концепция, охватывающая только приложения. Она позволяет разбить внутренние компоненты отдельного приложения на небольшие элементы, которые можно изменять, масштабировать и администрировать независимо друг от друга. Она не описывает способы взаимодействия приложений — для этой цели служат интерфейсы сервисов SOA.

SOA — это идеальный метод архитектуры для больших и сложных бизнес-приложений. Он наиболее подходит для сред, требующих интеграции со многими разнообразными приложениями.

Микросервисы - это специализация подхода к реализации для сервис-ориентированные архитектуры (SOA) используется для создания гибких, независимо развертываемых программные системы.

Однако приложения, основанные на рабочих процессах, которые имеют четко определенный поток обработки, сложно реализовать с помощью шаблонов архитектуры SOA. Поэтому небольшие приложения также не идеальны для SOA,

поскольку они не требуют компонентов обмена сообщениями промежуточного программного обеспечения.

Микросервис архитектура – вариант Сервис-Ориентированной Архитектуры (SOA) структурный стиль – который упорядочивает приложение как набор слабо связанных сервисов. В архитектуре микросервисов сервисы детализированы, а протоколы легковесны.

Микросервисы (или микросервисная архитектура) — это облачный подход, при котором единое приложение строится из множества слабосвязанных компонентов меньшего размера (так называемых сервисов), поддерживающих независимое развертывание.

Как правило, это:

- Сервисы в микросервисной архитектуре - это часто процессы которые общаются через сеть, используя независимые технологии и протоколы например HTTP, REST API, потоки событий (event) и агентов сообщений;
- Сервисы организованы вокруг бизнес-возможностей и изолированы с помощью ограниченного контекста;
- Сервисы имеют собственный стек технологий и могут быть реализованы с использованием разных языков программирования, баз данных, фреймворков, связующего программного обеспечения, выполняться в различных средах контейнеризации, виртуализации, под управлением различных операционных систем на различных аппаратных платформах: приоритет отдаётся в пользу наибольшей эффективности для каждой конкретной функции, нежели стандартизации средств разработки и исполнения, в зависимости от того, что подходит лучше всего;
- Сервисы небольшие по размеру, разрабатываются автономно и развертываются независимо, децентрализованны, построены и выпущены с помощью автоматизированных процессов (CI/CD и DevOps);
- Сервисы можно легко заменить в любое время: акцент на простоту, независимость развёртывания и обновления каждого из микросервисов;
- Сервисы организованы вокруг функций: микросервис по возможности выполняет только одну достаточно элементарную функцию;
- Сервисы могут быть реализованы с использованием различных языков программирования,

- Микросервисная архитектура симметричная, а не иерархическая: зависимости между микросервисами одноранговые.

Микросервис - это не уровень внутри монолитного приложения (например, веб-контроллера или внутреннего интерфейса для внешнего интерфейса). Скорее, это автономная часть бизнес-функциональности с понятными интерфейсами, которая может через свои внутренние компоненты реализовывать многоуровневую архитектуру. Со стратегической точки зрения архитектура микросервисов по существу следует Философия Unix "Делай одно и делай это хорошо".

Архитектура микросервисов обычно применяется для облачных приложений, бессерверных вычислений и приложений, использующие легкие контейнеры для развертывание.

По словам Мартина Фаулера, из-за большого количества (по сравнению с реализациями монолитных приложений) сервисов, децентрализованной непрерывной доставки и DevOps с целостным сервисным мониторингом необходимы для эффективной разработки, обслуживания и эксплуатации таких приложений. Следствием (и обоснованием) этого подхода является возможность индивидуального масштабирования отдельных микросервисов.

При монолитном подходе приложение, поддерживающее три функции, должно быть масштабировано полностью, даже если только одна из этих функций имеет ограничение ресурсов. При использовании микросервисов необходимо масштабировать только микросервис, поддерживающий функцию с ограниченными ресурсами, что обеспечивает преимущества по оптимизации ресурсов и затрат.

Особенности SOA	**Особенности Микросервисов**
• SOA использует интерфейсы, которые решают сложные проблемы интеграции в больших системах; • SOA взаимодействует с клиентами, поставщиками и поставщиками с использованием схемы XML;	• В Microservices модули слабо связаны; • Управление проектом, также может быть модульным; • Стоимость масштабируемости скудна;

• SOA использует мониторинг сообщений для улучшения измерения производительности и обнаружения атак безопасности; • При повторном использовании службы стоимость разработки и управления программным обеспечением несколько ниже; • Большая роль отводится компонентам-«посредникам» (middleware), таким как сложные очереди сообщений (message queue), адаптерам данных, и общему набору логики между различными компонентами системы, называемому интеграционной шиной ESB (enterprise service bus). Зачастую львиная доля сложной логики всей системы находится не в самих компонентах, а именно в шине ESB.	• Очень легко использовать несколько технологий в качестве нескольких функций в приложении; • Это идеальный сервис для эволюционных систем, где вы не можете предвидеть типы устройств, которые могут однажды получить доступ к вашему приложению. • Микросервисы – быстрый цикл разработки и постоянный выпуск.

Детализация сервиса, Закон Конвея

Задача Микросервисного подхода - это создание приложения в виде набора небольших автономных сервисов, нужных бизнесу, а также - это архитектурный стиль разработки, который позволяет создавать приложения в виде набора небольших автономных сервисов, разработанных для бизнес-сферы. Что может быть выражено с помощью Закона Конвея.

Закон Конвея

Закон Конвея — «Организации проектируют системы, которые копируют структуру коммуникаций в этой организации». Изречение, названное в честь программиста Мелвина Конвея, выразившего идею в 1967 году. Данное

определение появилось после публикации статьи Мелвина Конвея в журнале Harvard Business Review в 1968 году. Оригинальное высказывание звучит так:

Organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.

[Любая организация, которая разрабатывает систему (в широком смысле), вынуждена создавать проекты, структуры которых являются копией структуры связей организации.]

Суть Закона Конвея заключается в том, что при декомпозиции инженерами крупных задач на более мелкие для передачи части работ коллегам, возникает сложность координации между функциональными подразделениями. Во многих организациях команды разделены в соответствии с тем функционалом, который они выполняют, и существуют в отрыве от других команд. Поэтому структура программного интерфейса системы будет отражать социальные границы организации (организаций), которые её создали, что затрудняет общение. В рамках своего функционала команды могут отлично справляться с выполнением своих задач. Однако для того, чтобы создать что-то новое (функцию, продукт и т. д.), командам необходимо кросс-функциональное взаимодействие и совместная работа.

Делать сервис слишком маленьким считается плохой практикой, поскольку в этом случае накладные расходы времени выполнения и операционная сложность могут подавить преимущества такого подхода. Когда все становится слишком детализированным, необходимо рассмотреть альтернативные подходы - такие как упаковка функции в виде библиотеки, перемещение функции в другие микросервисы.

Если Домен-ориентированный дизайн используется при моделировании области, для которой создается система, тогда микросервис может быть таким маленьким, как Aggregate, или таким большим, как Bounded Context.

Компонент (component) – одна из основополагающих идей в разработке программ. Идея проста – компонент отвечает за определенную функцию, интерфейс API, или целый бизнес модуль. Главное – компонент можно убрать из системы, заменив на другой, со сходными характеристиками, без больших изменений для этой системы. Замена компонентов целиком встречается не так часто, но независимое обновление (upgrade) компонента без потери работоспособности системы в целом очень важно для легкости ее обновления и поддержки. Классический пример – качественная библиотека (к примеру, JAR),

модуль, или пакет, в зависимости от языка, которую можно подключить и использовать в своем приложении.

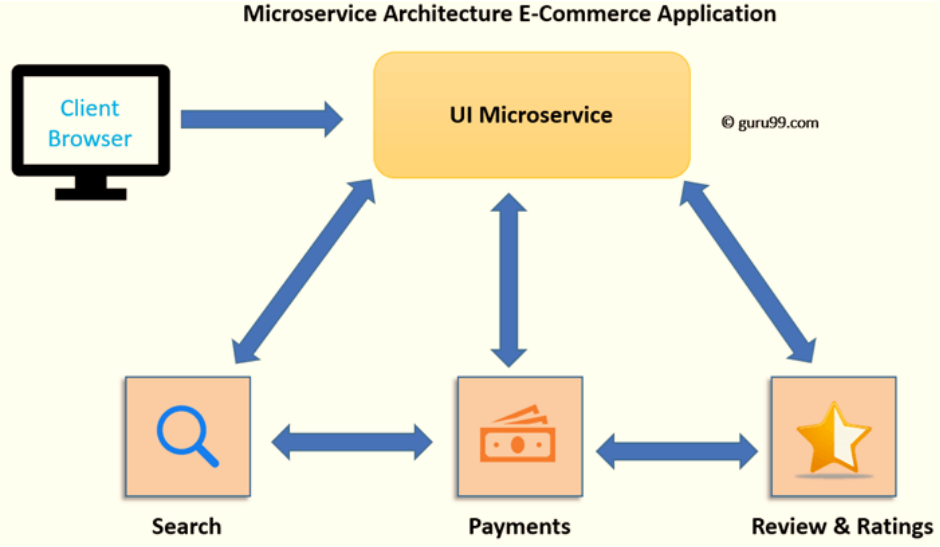
Логика в микросервисах, но не в сети и посредниках (в отличии от SOA) В мире микросервисов это явно выраженный анти-шаблон (anti-pattern). Вся логика в обязательном порядке должна находиться внутри микросервисов («разумные» точки доступа к сервису, smalt endpoint), даже если она повторяется в различных микросервисах. Сеть (и любой механизм передачи данных в целом), должна только передать данные, но не обладать никакой логикой (как еще говорят, «неразумные» линии передачи данных, dumb pipes).

Децентрализованное хранение данных

В монолитной архитектуре данные зачастую хранятся в единой, обычно реляционной, базе данных, со строго определенной схемой и обширным использованием запросов SQL. Микросервисы снова требуют противоположного подхода – в идеале никакого разделения данных, особенно через единую базу данных. Весь доступ, любые изменения данных происходят только через программный интерфейс API, предоставляемый микросервисом.

Микросервисы, таким образом, свободны в своем выборе способа хранения данных. Это может быть любой тип базы данных, например, нереляционная база NoSQL, база на основе документов, графовая база, или что-либо еще. Конечно, возникает значительная избыточность данных, но хранение данных не так дорого, а автономность и независимость развития каждого микросервиса, в идеале, должна окупить избыточность данных.

Пример приложения электронной коммерции, разработанного с использованием микросервисной архитектуры. В этом примере каждый микросервис ориентирован на отдельные возможности бизнеса. Поиск, оценка, обзор и оплата имеют свой экземпляр (сервер) и общаются друг с другом.



Пример архитектуры микросервисов

В монолитной архитектуре все компоненты объединяются в единый модуль. Но в архитектуре микросервисов они разбиты на отдельные модули (микросервисы), которые взаимодействуют друг с другом.

Связь между микросервисами — это связь без сохранения состояния, в которой каждая пара запроса и ответа независима. Следовательно, микросервисы могут общаться без особых усилий.

В микросервисной архитектуре данные объединяются. Каждый микросервис имеет отдельное хранилище данных.

Требования, налагаемые на систему основанной на микросервисах

Микросервисаная архитектура привносит дополнительные сложности, новые требования и проблемы, с которыми приходится иметь дело, например:

- сетевая задержка
- формат сообщений
- резервирование
- производительность
- доступность
- согласованность
- балансировка нагрузки
- отказоустойчивость.

Другие места, где проявляется сложность, - это увеличение сетевого трафика и, как следствие, снижение производительности. Кроме того, приложение, состоящее из любого количества микросервисов, имеет большее количество точек интерфейса для доступа к соответствующим экосистемам, что увеличивает архитектурную сложность.

Различные принципы организации структуры микросервисов и контрактов (например, HATEOAS, документация по интерфейсу и модели данных, полученная через Swagger и т. д.) были созданы, чтобы уменьшить влияние таких дополнительной сложностей.

Технологии

Компьютерные микросервисы могут быть реализованы на разных языках программирования и могут использовать разные инфраструктуры. Таким образом, наиболее важными технологическими решениями являются способ взаимодействия микросервисов друг с другом (синхронный, асинхронный, интеграция пользовательского интерфейса) и протоколы, используемые для связи (RESTful HTTP, обмен сообщениями, GraphQL ...).

В традиционной системе большинство технологических решений, таких как язык программирования, влияют на всю систему. Поэтому подход к выбору технологий совсем другой.

Фонд Eclipse Foundation опубликовал спецификацию разработки микросервисов Eclipse MicroProfile.

Устройство микросервисов и описание API в микросервисной архитектуре

API являются одним из важнейших аспектов разработки программного обеспечения. Приложение состоит из модулей. У каждого модуля есть интерфейс, определяющий набор операций, которые могут вызываться клиентами модуля. Хорошо спроектированный интерфейс делает доступными полезные функции, скрывая при этом их реализацию. Благодаря этому внутренние изменения не влияют на клиентов.

В монолитном приложении интерфейсы обычно описываются с помощью конструкций языка программирования — например, в виде Java-интерфейса, который определяет набор методов, доступных клиенту. Класс-реализация от клиента спрятан. Более того, поскольку Java — статически типизированный язык, любая несовместимость между интерфейсом и клиентом будет мешать компиляции.

API и локальные интерфейсы играют одинаково важную роль в микросервисной архитектуре. API сервиса является контрактом между ним и его клиентами.

API состоит из операций, которые клиент может вызывать, и событий, публикуемых сервисом. У операции есть имя, параметры и тип возвращаемого значения. Событие имеет тип и набор полей, оно публикуется в канале сообщений.

Трудность заключается в том, что определение API сервиса не основано на простых конструкциях языка программирования. Как и полагается, сервис и его клиенты компилируются отдельно. Если будет развернута новая версия сервиса с не совместимым API, вы не получите ошибку компиляции — вместо этого возникнут сбои на этапе выполнения.

Независимо от того, какой механизм IPC выбрать, важно создать четкое определение API сервиса, используя некий *язык описания интерфейсов* (interface definition language, IDL).

Кроме того, хорошая практика в пользу того, что описание сервиса должно начинаться с его API:

1) Первым делом описывается интерфейс.
2) Затем полученный результат рассматривается с клиентскими разработчиками.
3) И только после окончания работы над API реализуется сам сервис.

Такой подход повышает шансы на то, что ваш сервис будет удовлетворять требованиям клиентов.

Характер определения API зависит от выбранного механизма IPC. Например, если вы используете обмен сообщениями, API будет состоять из каналов, типов и форматов сообщений. Если применяете HTTP, API будет основан на URL-адресах, HTTP-командах и форматах запроса и ответа. Позже в этой главе я покажу, как описываются API.

API сервиса редко оказывается неизменным. Скорее всего, он будет эволюционировать со временем.

Сервисная сеть - Service Mesh

Современные приложения обычно проектируются как распределенные наборы микросервисов, каждый из которых выполняет какую-либо отдельную бизнес-функцию. Service Mesh - это специальный инфраструктурный уровень, который можно добавить к приложениям. Он позволяет прозрачно добавлять такие возможности, как наблюдаемость, управление трафиком и безопасность, не добавляя их в собственный код. Термин " Service Mesh" описывает как тип программного обеспечения, которое вы используете для реализации этого паттерна, так и безопасность или сетевой домен, который создается при использовании этого программного обеспечения.

Service Mesh - это способ управления тем, как различные части приложения обмениваются данными друг с другом. В отличие от других систем управления этим взаимодействием, Service Mesh представляет собой специальный инфраструктурный слой, встроенный прямо в приложение. Этот видимый инфраструктурный слой может документировать, насколько хорошо (или плохо) взаимодействуют различные части приложения, поэтому становится проще оптимизировать взаимодействие и избежать простоев по мере роста приложения.

Service Mesh - это конфигурируемый инфраструктурный уровень с низкой задержкой, предназначенный для обработки большого объема сетевого межпроцессного взаимодействии между сервисами инфраструктуры приложений с использованием интерфейсов прикладного программирования (API). Service Mesh обеспечивает быстрое, надежное и безопасное взаимодействие между контейнерными и часто эфемерными сервисами инфраструктуры приложений. Service Mesh обеспечивает критически важные возможности, включая обнаружение сервисов, балансировку нагрузки, маршрутизацию, шифрование, наблюдаемость, отслеживаемость, аутентификацию и авторизацию, а также поддержку схемы "выключатель" (Circuit Breaker).

Service Mesh обычно реализуется путем предоставления экземпляра прокси-сервера, называемого sidecar, для каждого экземпляра сервиса. Прокси-серверы обрабатывают межсервисные коммуникации, мониторинг и вопросы безопасности - в общем, все, что можно абстрагировать от отдельных сервисов. Таким образом, разработчики могут заниматься разработкой, поддержкой и обслуживанием кода приложения в сервисах; операционные команды могут обслуживать сетку сервисов и запускать приложения.

По мере роста размера и сложности развертывания распределенных сервисов, например, в системе на базе Kubernetes, их становится все труднее понимать и управлять ими. Его требования могут включать обнаружение,

балансировку нагрузки, восстановление после сбоев, метрики и мониторинг. Сервисная сетка также часто решает более сложные операционные требования, такие как A/B-тестирование, канареечное развертывание, ограничение скорости, контроль доступа, шифрование и сквозная аутентификация.

В служебной сети каждый экземпляр службы связан с экземпляром обратного прокси-сервера, который называется служебным прокси, дополнительным прокси или дополнительным сервером. Экземпляр службы и дополнительный прокси-сервер совместно используют контейнер, а контейнеры управляются инструментом оркестровки контейнеров, например Kubernetes, Nomad, Docker Swarm, или же Прокси-серверы службы отвечают за связь с другими экземплярами службы и могут поддерживать такие возможности, как обнаружение службы (экземпляра), балансировка нагрузки, аутентификация и авторизация, безопасная связь и другие.

Говорят, что в Service Mesh экземпляры службы и их вспомогательные прокси составляют плоскость данных, которая включает не только управление данными, но также обработку запросов и ответ. Service Mesh также включает в себя плоскость управления для управления взаимодействием между службами через их прокси-серверы. Существует несколько вариантов архитектуры служебной сети: Open Service Mesh, Istio (совместный проект Google, IBM и Lyft), Linkerd (CNCF проект во главе с Плавучий), Consul (a HashiCorp product), NGINX Service Mesh и многие другие в ландшафте Layer5 Service Mesh Landscape.

В плоскости управления сервисной сеткой Meshery, обеспечивает управление жизненным циклом, конфигурацией и производительностью в развертываниях сервисной сети.

Обнаружение сервисов – Service Discovery

Как клиент будет взаимодействовать с приложением, насчитывающим сотни или тысячи микросервисов, особенно в том случае, если доступ к каждому микросервису может осуществляться посредством нескольких веб-служб (например, для разных версий клиентов). Это незначительная проблема в монолитной архитектуре, т. к. клиент делает только один вызов, а обо всем остальном позаботится само приложение.

Но в архитектурах на основе микросервисов возникают **две сложности**:

1) клиенты вынуждены одновременно вызывать несколько микросервисов для достижения того же эффекта, который в монолитном приложении обеспечивался единственным вызовом;

2) клиенты должны знать адреса сервисов.

Проиллюстрируем эти сложности на простом примере. Представьте, что пользователь обращается к приложению книжного онлайн-магазина и хочет просмотреть страницу своей учетной записи. На странице отображаются история заказов, рекомендации, текущее состояние корзины, платежи, настройки учетной записи и т. д. В монолитном приложении, когда пользователь переходит по ссылке Моя учетная запись, он получает страницу Моя учетная запись, а все волшебство творится в серверной части приложения, где вызываются разные функции и извлекаются сведения из базы данных. В портативных и мобильных устройствах может потребоваться выполнить другой набор вызовов, учитывая ограниченность объема памяти и вычислительной мощности, что добавляет сложности.

В архитектуре на основе микросервисов клиент сам мог бы вызывать все необходимые микросервисы, возвращающие, например, состояние корзины, информацию о платежах и параметры учетной записи. Но такой подход очень неэффективен и предусматривает жестко ограниченный порядок взаимодействий. Если использовать его, мы потеряем гибкость внесения изменений, таких как дальнейшее разделение микросервисов на несколько микросервисов, когда это необходимо, или наоборот.

Кроме того, клиент должен знать адреса всех микросервисов, которые нужно вызвать, чтобы сформировать страницу Моя учетная запись. Поэтому нам нужна система, которая будет играть роль общей точки входа для клиентов и внешних вызовов, и еще одна система, хранящая адреса микросервисов.

Например, есть код, который вызывает сервис через REST API. Для выполнения запроса этому коду необходимо знать местоположение экземпляра сервиса в сети (IP-адрес и порт). В традиционных приложениях, работающих на физическом оборудовании, сетевое местоположение экземпляров сервисов обычно статическое. Ваш код, к примеру, мог бы извлечь сетевые адреса из конфигурационного файла, который время от времени обновляется. Но в современных микросервисных приложениях, основанных на облачных технологиях, все может быть не так просто. Современная система куда более динамичная и Экземпляры сервисов имеют динамически назначаемые IP-адреса.

Сетевое местоположение назначается экземплярам сервисов динамически. Более того, набор этих экземпляров постоянно меняется из-за автоматического масштабирования, отказов и обновлений. Из-за этого ваш клиент должен использовать обнаружение сервисов.

Обзор механизмов обнаружения сервисов – Реестр сервисов (Service Registry)

При наличии тысяч микросервисов наш API-шлюз должен знать адреса всех служб, участвующих в работе приложения. Для этой цели часто используется реестр служб – база данных микросервисов с их адресами, к которой можно обратиться при необходимости. Разработчик должен позаботиться о создании записи для своего микросервиса в этом реестре.

Service Registry - является ключевой частью обнаружения сервисов. Он представляет собой базу данных, содержащую сетевые местоположения экземпляров сервисов для их взаимодействия на уровне приложений. Он служит центральным местом, где разработчики приложений могут зарегистрироваться и найти схемы, используемые для конкретных приложений.

Service Registry - должен быть высокодоступным и актуальным, так как клиенты могут кэшировать сетевые местоположения, полученные из Service Registry. Однако со временем эта информация устаревает, и клиенты не могут обнаружить экземпляры услуг. Следовательно, технологически Service Registry состоит из кластера серверов, которые используют протокол репликации для поддержания согласованности.

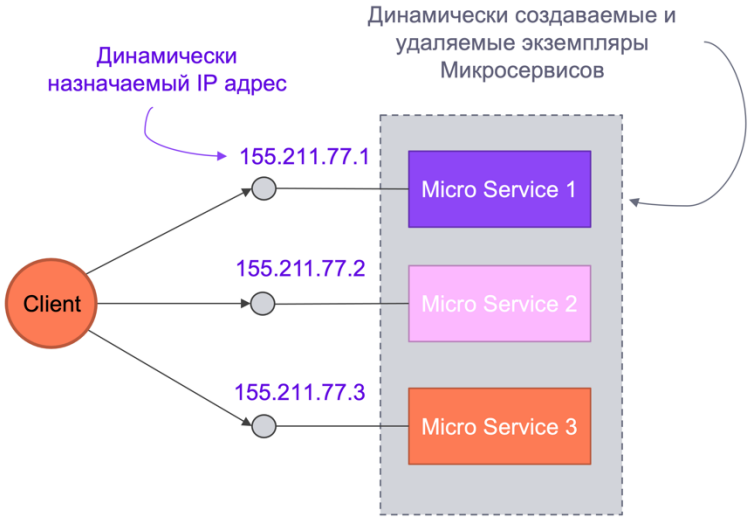
Суть заключается в том, чтобы при запуске микросервис регистрировал себя в реестре. Когда клиент выполняет вызов, API-шлюз определяет адреса необходимых микросервисов, вызывает их и объединяет полученные результаты для передачи обратно клиенту.

Как быть, если мы потеряем реестр с данными? Для решения этой проблемы можно использовать несколько инструментов с открытым исходным кодом, таких как Consul и SkyDNS, которые фактически обнаруживают микросервисы и удостоверяются, что они запущены и работают. Например, Consul – это зрелый инструмент, способный использовать DNS-имена для доступа к микросервисам и хранить эту информацию в реестре. Он также может выполнять периодические проверки работоспособности и поддерживать работу кластеров.

Поэтому, клиента нельзя сконфигурировать статически, предоставив ему IP-адреса сервисов. Приложение должно задействовать механизм динамического обнаружения. По своей сути обнаружение сервисов является довольно простым: его ключевым компонентом выступает реестр сервисов — база данных с информацией о том, где находятся экземпляры сервисов приложения.

Когда экземпляры сервисов запускаются и останавливаются, механизм обнаружения обновляет реестр. Когда клиент обращается к сервису, механизм

обнаружения получает список его доступных экземпляров, запрашивая реестр, и направляет запрос одному из них.



Механизм обнаружения сервисов

Есть **два основных способа реализации механизма** обнаружения сервисов.

- Сервисы и их клиенты напрямую взаимодействуют с реестром.
- За обнаружение сервисов отвечает инфраструктура развертывания.

Пример стека технологий системы разделенной на микросервисы

Основные принципы системы

Концепция микросервисов, включает в себя:

Модульность

Приложение состоит из нескольких сервисов, представляющих собой законченные приложения. Разделение не по функционалу, а по бизнес

логике. Каждый сервис сам отвечает за изменение и хранение состояния своих объектов.

Независимость реализации (гетерогенность)

Сервисы могут быть реализованы на различных языках программирования, с использованием различных технологий.

Независимость развертывания

Сервисы могут быть развернуты независимо друг от друга. Сервис должен иметь минимум разделяемых зависимостей - в идеальном случае, только стандартные библиотеки времени исполнения.

Технологический стек

Front

ReactJS

Redux

Moment js

Invariant js

Core-js

Lodash

Reselect

Back

ApacheDS

Oracle Access Manager

Spring Cloud

Spring Boot

Spring Security

Hibernate

Netflix OSS

Zuul - имплементация API Gateway, поддерживающая динамический роутинг и фильтры

Hystrix - имплементация паттерна Circuit Breaker + мониторинг

Eureka - компонент, представляющий собой Registry service

Turbine - агрегатор данных, дашбоард

OpenZipkin - Распределённая система сбора и обработки метрик внутренних операций приложений

Prometheus - Система мониторинга

Grafana - Платформа для визуализации данных, построения графиков и дашбордов

EFK

Включает в себя:

Elasticsearch - Поисковая система, основанная на Lucene, позволяющая

производить индексирование и поиск на различных структурированных и

неструктурированных данных, метрик, документов

FluentD – оптимизированный для Kubernatis коллектор данных

(лог). Позволяет собирать данные из множества источников,

преобразовать их и отправлять их в Elasticsearch

Kibana- позволяет визуализировать данные из Elasticsearch

Jasper Report Java библиотека для создания отчётов на основе XML-шаблонов

Включает в себя:

iText – библиотека для работы с PDF

Apache POI – библиотека для работы с MS форматом

Postgres – база данных

Интеграция

Apache Kafka - Высокопроизводительный брокер сообщений

Apache Zookeeper - Координатор распределённых взаимодействий

Yahoo Kafka Manager - Средство управления кластерами Apache Kafka

IBM Integration Bus - Интегратор сообщений, ESB

IBM MQ - брокер сообщения

Преимущества и недостатки микросервисной архитектуры

Микросервисы обладают множеством разнообразных преимуществ. Многие из них могут быть присущи любой распределенной системе. И микросервисы нацелены на достижение максимума этих преимуществ.

Преимущества микросервисов

1) Бизнес возможности

a) Разбиение сложной задачи на составные более простые части, изоляция сложности, и поиск абстракций, позволяет упростить и сделать задачу управляемой, решаемой и адаптируемой к сменяемому бизнес-контексту. Разбиение программы на пакеты, функции, классы, а затем и на совершенно независимо работающие друг от друга компоненты логически вытекает из анализа задачи.

b) Приложения независимо от других приложений предприятия

c) Компоненты могут масштабироваться независимо друг от друга, что снижает затраты и стоимость масштабирования всего приложения, если узким местом выступает лишь какая-то одна бизнес-функция.

2) Технологический стек

Имея систему, составленную из нескольких совместно работающих сервисов, можно прийти к решению использовать внутри каждого из них различные технологии. Это позволит выбрать для каждого задания наиболее подходящий инструментарий, не выискивая какой-либо стандартный подход на все случаи жизни.

Если какой-то части системы требуется более высокая производительность, можно принять решение по использованию иного набора технологий, более подходящего для достижения требуемого уровня производительности.

Можно также принять решение об изменении способа хранения данных для разных частей нашей системы. Например, пользовательский обмен сообщениями в социальной сети можно хранить в графоориентированной базе данных, отражая тем самым присущую социальному графу высокую степень взаимосвязанности, а записи в блоге, создаваемые пользователями, можно хранить в документоориентированном хранилище данных, давая тем самым повод для использования разнородной архитектуры.

Микросервисы также позволяют быстрее внедрять технологии и разбираться в том, чем именно нововведения могут нам помочь. Одним из величайших барьеров на пути принятия новой технологии является риск, связанный с ее использованием. Если при работе с монолитным приложением захочется попробовать новые язык программирования, базу данных или структуру, то влияние распространится на существенную часть системы. Когда

система состоит из нескольких сервисов, то появляются несколько новых мест, где можно проверить работу новой части технологии.

Можно выбрать такой сервис, с которым риск будет наименьшим, и воспользоваться предлагаемой им технологией, зная, что могу ограничить любое потенциально отрицательное воздействие. Возможность более быстрого внедрения новых технологий рассматривается многими организациями как существенное преимущество.

Разумеется, использование нескольких технологий не обходится без определенных издержек. Некоторые организации предпочитают накладывать на выбор языков ограничения. В Netflix и Twitter, к примеру, в качестве платформы используется преимущественно Java Virtual Machine (JVM), поскольку они очень ценят надежность и производительность этой системы.

a) Возможность сервисам развивать технологию независимо друг от друга без чрезмерной связанности;

b) Микросервисы позволяют упростить использование разнообразных технологий;

c) Гетерогенность – возможность реализации модулей системы на разных языках; гетерогенные информационные системы помогают избегать наличия тесных связей между модулями системы благодаря использованию разных языков программирования.

3. Структура команды, управление и процессы

a) Решение организационных вопросов

Если команда разбросана по разным местам, проблема с объемными базами исходного кода может только усугубиться. Также общеизвестно, что небольшие команды, работающие с небольшим объемом исходного кода, как правило, показывают более высокую продуктивность.

Микросервисы позволяют эффективнее приспособить архитектуру к решению организационных вопросов, позволяя свести к минимуму число разработчиков каждого отдельно взятого фрагмента исходного кода, чтобы найти баланс между размером команды и продуктивностью ее работы. Можно также распределить принадлежность сервисов между командами, чтобы люди, работающие над тем или иным сервисом, трудились вместе;

b) Скорость разработки

Микросервисы сокращают риски при разработке, тем самым ускоряя ее. Добавление новой функции к монолитному сервису может затронуть все другие его функции. Разработчикам следует тщательно рассмотреть последствия добавления кода и удостовериться, что они ничего не испортят. С другой стороны, правильная архитектура микросервисов включает новый код для функции, которая добавляется в новый сервис. Разработчики могут быть уверены, что никакой создаваемый код не затронет уже имеющийся код, если только не создать явное соединение между двумя микросервисами;

c) Разумный размер кода делает процесс разработки быстрым и удобным. Это же позволяет проще и настроить системы постоянного контроля качества и развертывания сделанных изменений на сервере (CI/CD, continuous integration and delivery), и сделать их работу быстрой, позволяя программистам быстро проверить, было ли их последнее изменение удачным;

d) Гибкость и продуктивность работы разработчиков

Гораздо меньший размер и менее связанная с другими компонентами функциональность позволяет программистам быстро проводить в жизнь новые идеи, рефакторинг кода, и пробовать новые подходы и процессы разработки, не затрагивая при этом остальное приложение;

e) Внесение командой изменений, тестирование и развертывание отдельных компонентов можно выполнять независимо от других компонентов и команд, сократив при этом итерационные циклы;

f) Команды могут использовать различные стеки и языки программирования для разных компонентов.

4) Архитектура и развитие

a) Компонуемость

Одной из ключевых перспектив, открывающихся в результате использования распределенных систем и сервис-ориентированных архитектур, является возникновение новых возможностей повторного использования функциональности.

Применение микросервисов позволяет пользоваться какой-либо функциональной возможностью различными способами и для достижения различных целей. Это свойство может приобрести особую важность при обдумывании порядка использования клиентами программ;

b) Модульность

Это упрощает понимание, разработку, тестирование приложения и делает его более устойчивым к эрозии архитектуры. Это преимущество часто сравнивают со сложностью монолитных архитектур;

c) Передача данных

Связь между микросервисами осуществляется по сети, по хорошо известным протоколам, поддерживаемыми практически всеми известными языками и их библиотеками. Микросервисы больше не являются частью единого проекта и репозитория в системе контроля версия, и разрабатывающие их команды теперь свободны делать любой выбор, эффективно позволяющий решить задачу, стоящую перед компонентом. Это открывает двери для быстро меняющегося мира технологий, и когда-то сделанный выбор архитектуры и языка для одного компонента больше не диктует того же новым компонентами и сервисам;

d) Интеграция гетерогенных и унаследованные системы

Микросервисы считаются эффективным средством модернизации существующего монолитного программного обеспечения. Процесс для Модернизация программного обеспечения устаревших приложений выполняется с использованием инкрементного подхода;

e) Оптимизация программ с целью последующей замены

Если в организации среднего или крупного размера используется большая, непонятная, унаследованная от прошлых времен система, стоящая где-то в углу, то никто не хочет к ней даже прикасаться. Но компания не может без нее работать, несмотря на то что она написана на каком-то странном диалекте и работает только на оборудовании, которое давно пора списать. Почему же никто эту систему не заменил? Это слишком объемная и рискованная работа. В случае же использования отдельных небольших по объему сервисов с их заменой на более подходящую реализацию или даже полным удалением справиться гораздо легче;

f) Переписывание сервиса

Команды, использующие подходы, связанные с микросервисами, не видят ничего страшного в полной переработке сервиса, когда это потребуется, и даже в полном избавлении от него, когда потребность в нем отпадет. Когда исходный код состоит всего из нескольких сотен строк, какая-либо эмоциональная привязка к нему практически отсутствует, а стоимость его замены совсем не велика;

g) Обновление кода требует меньших усилий: для добавления новых компонентов или функций не нужно перестраивать все приложение;

h) Простота проверки и тестирования отдельно взятого микросервиса;

i) Более простой шаблон архитектуры, который легко понять разработчикам.

5) Возможности работы и адаптации под разные нагрузки

a) Масштабирование

В больших монолитных сервисах расширять приходится все сразу. Одна небольшая часть всей системы может иметь ограниченную производительность, но, если в силу этого тормозится работа всего огромного монолитного приложения, расширять нужно все как единое целое. При работе с небольшими сервисами можно расширить только те из них, которые в этом нуждаются, что позволяет запускать другие части системы на небольшом, менее мощном оборудовании.

Именно по этой причине микросервисами воспользовалась компания Gilt, занимающаяся продажей через Интернет модной одежды. Начав в 2007 году с использования монолитного Rails-приложения, к 2009 году компания столкнулась с тем, что используемая система не в состоянии справиться с возлагаемой на нее нагрузкой. Разделив основные части своей системы, Gilt смогла намного успешнее справляться со всплесками трафика, и теперь компания использует 450 микросервисов, каждый из которых запущен на нескольких отдельных машинах.

При использовании систем предоставления сервисов по требованию, подобных тем, которые предлагает Amazon Web Services, можно даже применить такое масшта- бирование по требованию для тех частей приложения, которые в этом нуждаются. Это позволит более эффективно контролировать расходы. Столь тесная корреляция архитектурного подхода и практически сразу же проявляющейся экономии средств наблюдается крайне редко;

b) Независимое масштабирование

Если функции разбиты на микросервисы, то используемые каждым классом микросервиса компоненты инфраструктуры и инстансы можно независимо масштабировать в сторону увеличения и уменьшения. Это помогает рассчитать стоимость конкретной функции, определить функции, которые следует оптимизировать в первую очередь, а также обеспечить высокую производительность других функций, если одна из них вышла из-под контроля вследствие нехватки ресурсов.

Независимое масштабирование компонентов позволяет уменьшить потери времени и средств, связанные с масштабированием всего приложения из-за одного компонента, создающего чрезмерную нагрузку;

c) Адаптивность к нагрузке

Микросервисы позволяют получить максимальную отдачу от масштабируемости облачных сред — любой компонент можно отслеживать и масштабировать независимо от остальных компонентов, обеспечивая при этом максимально быстрое реагирование на изменения нагрузки и наиболее эффективное использование вычислительных ресурсов;

d) Независимое управление компонентами приложения

Эластичность и значительная вычислительная мощность облака дает возможность разбить приложение на логические компоненты и запускать их и управлять ими индивидуально. При необходимости легко увеличить пропускную способность приложения, увеличив количество экземпляров компонентов (работающих в виде микросервисов), испытывающих наибольшую нагрузку. Это горизонтальное масштабирование – при работе в облаке его возможности широки. Вертикальное масштабирование же подразумевает рост мощности одного сервера и его аппаратных возможностей, что крайне ограничено, и более того, самые мощные серверы обычно очень дороги.

6) Надёжность и устойчивость

a) Устойчивость

Ключевым понятием в технике устойчивости является - перегородка. При отказе одного компонента системы, не вызывающем череду связанных с ним отказов, проблему можно изолировать, сохранив работоспособность всей остальной системы. Именно такими перегородками станут границы сервисов.

В монолитном сервисе при его отказе прекращается вся работа. Работая с монолитной системой, можно уменьшить вероятность отказа, запустив систему сразу на нескольких машинах, но, работая с микросервисами, мы можем создавать такие системы, которые способны справиться с тотальными отказами сервисов и снизить соответствующим образом уровень их функциональных возможностей.

Чтобы убедиться в том, что системы, составленные из микросервисов, могут воспользоваться улучшенной устойчивостью должным образом, нужно разобраться с новыми источниками отказов, с которыми должны справляться распределенные системы. Отказать могут как сети, так и машины, и такие отказы

неизбежны. Необходимо знать, как с этим справиться и как это может (или не может) повлиять на конечного пользователя программного средства;

b) Изоляция сбоев

Даже лучшие компании-разработчики сталкиваются с критическими сбоями в ходе своей деятельности. Чтобы свести влияние таких сбоев к минимуму, помимо соблюдения стандартных рекомендаций также следует разрабатывать микросервисы. Преимущество качественной архитектуры микросервисов состоит в том, что сбой затрагивает только один компонент сервиса. Другие компоненты продолжают работать;

c) Изоляция в целях безопасности

В случае монолитного приложения, если есть брешь в системе защиты хоть одной функции (например, уязвимость, которая позволяет удаленно выполнять код), злоумышленник может получить доступ ко всем другим функциям системы. Если, например, нарушена безопасность функции отправки аватара, злоумышленники могут взломать базу данных с паролями пользователей. Разделение функций между микросервисами позволяет обеспечить безопасность доступа к ресурсам, предоставив каждому сервису собственную роль. Если соблюдены рекомендации в отношении микросервисов, при взломе одного сервиса злоумышленник получает доступ только к ресурсам этого сервиса. Для доступа к другим сервисам ему приходится взламывать их по отдельности;

d) Надёжность

Благодаря разделению приложения, сбой одного микросервиса никак не сказывается на работе других. Каждый микросервис может работать в соответствии с собственными требованиями к доступности, не ограничивая другие компоненты или приложение в целом.

7) Управление приложением, развёртывание, публикация

a) Простота развертывания

Можно расширять только те микросервисы, которые в этом нуждаются.

b) Внесение изменений

Для реализации внесения изменений в одну строку монолитного приложения, состоящего из миллионов строк кода, требуется, чтобы было развернуто все приложение. Это развертывание может быть весьма рискованным и иметь крайне негативные последствия. На практике подобные рискованные развертывания из-за вполне понятных опасений происходят нечасто. К

сожалению, это означает, что изменения копятся и копятся между выпусками, пока в производство не будет запущена новая версия приложения, имеющая массу изменений. И чем больше будет разрыв между выпусками, тем выше риск, что что-нибудь будет сделано не так!

При использовании микросервисов можно вносить изменения в отдельный микросервис и развертывать его независимо от остальной системы. Это позволит развертывать код быстрее. Возникшую проблему можно быстро изолировать в рамках отдельного сервиса, упрощая тем самым быстрый откат. Это также означает, что новые функциональные возможности могут дойти до клиента быстрее. Именно то, что такая архитектура позволяет устранить максимально возможное количество препятствий для запуска приложения в эксплуатацию, и стало одной из основных причин, по которой такие организации, как Amazon и Netflix, воспользовались ею.

c) Распределенная разработка

Распараллеливание разработки даёт возможности небольшим автономным командам развиваться, развертывать и независимо масштабировать соответствующие сервисы. Это также позволяет управлять архитектурой отдельного сервиса за счет непрерывного рефакторинга.

d) Скорость развёртывания

Контейнер с микросервисом запускается быстрее, это помогает ускорить процесс разработки и развёртывания. Что упрощает непрерывную интеграция и непрерывную доставку.

e) Возможность обновления отдельных сервисов независимо от большого приложения;

f) Возможность настройки и проведения различных видов частичного развертывание.

Недостатки микросервисов:

Обратной стороной компонентной разработки в распределенной среде является отсутствие гарантии работоспособности – любой сетевой вызов, в отличие от вызова функции внутри единого процесса, подвержен отказам и сбоям, иногда в течение долгого времени.

a) Дополнительная сложность. Да, сложность решения не может быть определена количественно и сравнивать ее можно только в относительном выражении. Хотя изначально микросервисы предназначены для того, чтобы упростить архитектуру приложения за

счет разделения монолита на части, микросервисная архитектура сложна в развертывании и обслуживании.

b) Если бизнес-функция спроектирована для реализации в монолитном приложении, то нет явной возможности для её реализации в распределенном приложении.

c) Высокая связанность компонентов. Некоторые части сервиса тесно связаны друг с другом. Попытка разделить их только для того, чтобы «вписаться» в архитектуру, может закончиться катастрофой. Отсутствие опыта. Отсутствие опыта имеет решающее значение при решении любых проблем — в том числе вопросов, связанных с сервисно-ориентированной архитектурой. Когда дело доходит до микросервисов, ущерб приумножается: если вы разворачиваете сервисы в неправильном порядке или происходит сбой, при котором один из зависимых сервисов выходит из строя, менять что-то может быть слишком поздно.

d) Сложнее тестировать крупные связки микросервисов. Сквозное тестирование. Традиционное монолитное приложение позволяет запускать тесты почти мгновенно. Наличие нескольких сервисов с взаимозависимостью приведет к задержке тестирования без жизнеспособной оркестрации. Хаотические контракты на данные. Разработка и хранение контрактов на данные внутри команды сильно отличается от обмена ими между командами. При работе с микросервисами ваша команда может не находиться в одном регионе, не говоря уже об использовании одного и того же языка программирования. Выработка контрактов с данными для особых нужд будет стоить вам времени и места.

e) Проблемы при отладке. Каждая служба будет иметь собственный набор файлов журнала для анализа. Чем больше сервисов, тем больше файлов.

f) Размытые границы между микросервисами диктуют аккуратный выбор протоколов и передаваемых структур данных. Тестировать взаимодействии микросервисов, взаимодействующих по сети, иногда бывает крайне сложно.

g) Реализация вариантов использования, охватывающих несколько служб, требует координации между командами.

h) Сложность управления разнородной инфраструктурой размещения микросервисов и их интеграций к родственным решениям (SOA, монолиты).

i) Устаревшая база кода. Большинство команд каждый день работает с устаревшей базой кода. Быстро развитие технологий двигает нас вперед, но в то же время они еще больше изолируют нас от legacy-кода. Вы уверены, что только что разработанный фреймворк на RabbitMQ, будет хорошо работать с устаревшим приложением?

j) Стоимость развертывания. Микросервисы — это распределенные системы с молекулярной структурой, а распределение имеет свою цену. Если монолит можно развернуть на одной виртуальной машине или в контейнере, то каждой службе в микросервисной структуре (по крайней мере, в идеальном мире) требуется отдельная виртуальная машина или контейнер. Их объем меньше, чем у монолита, но, скорее всего, они обойдутся дороже даже без учета стоимости обслуживания.

k) Микросервисная архитектура стоит дорого, так как вам всегда нужно поддерживать различное серверное пространство для разных бизнес-задач (разработка, тестирование, проверка на аудитории, продуктовое использование).

l) Команда DevOps. Без команды DevOps поддерживать и контролировать микросервисы не получится, а найм таких специалистов может и помочь, и навредить компании. С одной стороны, DevOps — это широко распространенное и проверенное операционное решение. Но если компания небольшая, найм таких специалистов скорее принесет убытки, чем сделает организацию более прогрессивной.

Глава 4. Событийно-ориентированная архитектура

Событийно-ориентированная архитектура (Event Driven Architecture – EDA) - это интеграционная модель программных систем, построенная на публикации, захвате, обработке и хранении (или персистенции) событий. В частности, когда приложение или сервис выполняет действие или претерпевает изменение, о котором может захотеть узнать другое приложение или сервис, оно публикует событие - запись этого действия или изменения, - которое другое приложение или сервис может использовать и обрабатывать для выполнения одного или нескольких действий в свою очередь.

Событийно-ориентированная архитектура обеспечивает свободное соединение между связанными приложениями и сервисами - они могут взаимодействовать друг с другом, публикуя и потребляя события, не зная друг о друге ничего, кроме формата события. Эта модель предлагает значительные преимущества по сравнению с архитектурой запрос/ответ (или интеграционной моделью), в которой одно приложение или служба должны запрашивать конкретную информацию у другого приложения или службы, ожидающих конкретного запроса (пример REST).

Событийно-ориентированная архитектура максимально раскрывает потенциал "облачных" приложений и позволяет использовать мощные прикладные технологии, такие как аналитика в реальном времени и поддержка принятия решений.

EDA – это архитектура, в которой события начинают обмен сообщениями в реальном времени между свободными приложениями.

EDA базируется на так называемых программах-агентах, которые обрабатывают события, чтобы находить события на предприятии и, воспользовавшись толкающим (push) подходом, ставить в известность все другие приложения, которые нужно известить о данных событиях.

Все это происходит в реальном времени. Источники опубликовывают события, а подписчики получают их в процессе поступления. В архитектуре EDA публикация событий, подписка на события с поддержкой их буферизации в очередях и фильтрации, залог совершения доставки событий в случае выхода из строя оборудования или сети должны иметь поддержку интеграционного программного обеспечения.

EDA - является шаблоном архитектуры программного обеспечения, позволяющим создание, определение, потребление и реакцию на события.

Можно сказать, это целое научное направление, основоположником которого считается профессор Дэвид Лукхэм.

Назначение EDA

EDA призвана придавать системам максимальную гибкость, что дает возможность внесения в них быстрых и дешевых изменений. Сверхзадачей EDA является приближение информационных систем к реальным операциям компании.

Основой EDA считается:

- Поддержание объединения многие-ко-многим.
- Эксплуатация алгоритма управления потоком данных, который определяется принимающей стороной на основе самого сообщения.
- Поддержание через сеть модулей динамических, параллельных, асинхронных потоков данных.
- Возможность реагировать на новые внешние входные данные, поступающие в любое время.

Этот архитектурный шаблон может применяться при разработке и реализации приложений и систем, передающих события среди слабосвязанных программных компонентов и сервисов.

Создание приложений и систем в рамках архитектуры, управляемой событиями, позволяет им быть сконструированными способом, способствующим лучшей интерактивности, поскольку системы, управляемые событиями, по структуре более ориентированы на непредсказуемые и асинхронные окружения.

Архитектура, управляемая событиями, соответствует Service-Based-архитектуре (SOA+MSA), поскольку сервисы могут активироваться триггерами, срабатывающими от входящих событий.

Ключевые особенности:

- События обрабатываются не человеком, а автоматизированной системой;
- События используются как инициаторы для вызова сервисов;
- Технологическая и логическая развязка обработчика и инициатора;
- Асинхронность;

- Получение события в реальном времени и возможность управлять предприятием в режиме реального времени;
- Push-уведомление
- События отправляются в стиле «отправил и забыл»
- Потребитель отвечает немедленно
- Существует разница между событиями и командами

Обладая характеристиками асинхронной модели системы сообщений, event-driven также имеет асинхронные характеристики. Традиционные вызовы методов, такие как вызов b.xmethod(), представляют собой синхронную модель. В это время вы должны дождаться окончания выполнения метода b, прежде чем продолжить выполнение другого кода. Удаленные вызовы методов RPC также являются синхронной моделью, а для асинхронной модели, после того как производитель события посылает событие, ему не нужно ждать ответа, вы можете продолжить выполнение следующего кода.

Разница между **событием и сообщением:**

Событие	**Сообщение**
▪ Событие означает, что что-то произошло (после того, как что-то произошло); ▪ Событие передается любому отслеживаемому коду (код может реагировать на событие).	▪ Тип информации, информация, передаваемая системе
Событие (Event) - это действие, вызванное пользователем.	Сообщение (Message) - это информация передаваемая системе.

В компьютерах понятия события и сообщения более запутаны, но суть в другом: событие запускается пользователем (человеком, управляющим компьютером) И может быть запущено только пользователем, операционная система может почувствовать событие, запущенное пользователем, и преобразовать это событие в (конкретное) сообщение, передаваемое в очередь сообщений программы.

Событие

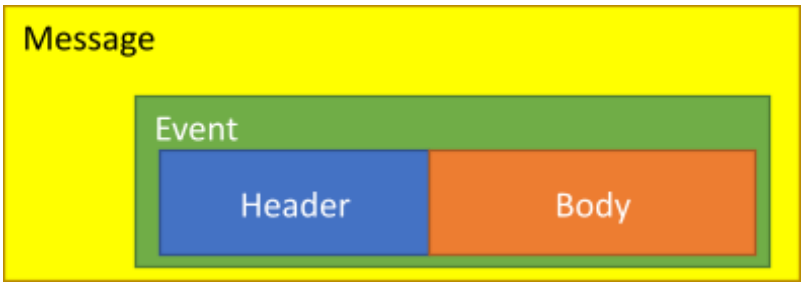
Событие можно определить как «существенное изменение состояния». Например, когда покупатель приобретает автомобиль, состояние автомобиля изменяется с «продаваемого» на «проданный». Системная архитектура продавца автомобилей может рассматривать это изменение состояния как событие, создаваемое, публикуемое, определяемое и потребляемое различными приложениями в составе архитектуры.

Сообщения о событиях, которые поступают в систему, распространяются по списку, а получателей таких сообщений называют подписчиками (subscriber); в роли подписчика может быть представлен человек или автомат.

Событие может состоять из двух частей: заголовка (header) события и тела (event body) события:

- Заголовок состоит из относящихся к событию метаданных, в том числе: идентификатора спецификации события, указателя типа, имени события, отметки времени и источника.

- Тело события включает в себя описание самого события, то, что в действительности произошло. Для того чтобы получатель мог предпринимать какие-то действия, не делая дополнительных запросов используя предоставленные данные, тело должно быть достаточно содержательным, а также оно должно содержать описание, подготовленное на принятом в бизнесе языке, или полную онтологию, чтобы смысл события был понятен подписчику. Тело события не следует путать с шаблоном или логикой, которая может быть применена в качестве реакции на события.



Сообщение о событии распределяется между подписчиками, которые каким-либо образом на него реагируют. Это может быть как вызов определенного сервиса так и перестройка бизнес-процесса, а также дальнейшее распространение сведений о событии, в том числе дополнительных.

Архитектура такого плана является не просто слабо связанной, а очень слабосвязанной (extreme loose coupling) и распределенной. Источник или создатель сообщения знает только то, что сообщение передано, не участвуя при этом в его дальнейшей судьбе. По правде говоря, отследить траекторию

отработки сообщения в условиях его распространения по подписке, практически невозможно. Вот почему наличие асинхронных потоков работ и входных данных подразумевается в EDA.

Большое количество входных данных способны содержать только значимые события, которые оформлены в виде сообщений и готовы к отправке подписчикам, потоки единичных событий (такие как сведения, считанные с меток RFID), и потоки сложных событий.

В зависимости от того, что входит в состав потоков событий архитектуры, EDA может быть систематизирована по стилю обработки. В одном случае происходит простое обрабатывание событий (simple event processing), которое подразумевает прямую передачу событий из входного потока в процессор обработки событий. В другом случае, в процессе многочисленной обработки событий (stream event processing), единичные события в первую очередь поступают в генератор, где из них с помощью фильтрации или иной предварительной обработки формируются значимые события, которые в свою очередь передаются в процессор событий. Третий случай - это обработка сложных событий (complex event processing). Он отличается тем, что единичные события не являются однородными.

Компоненты EDA

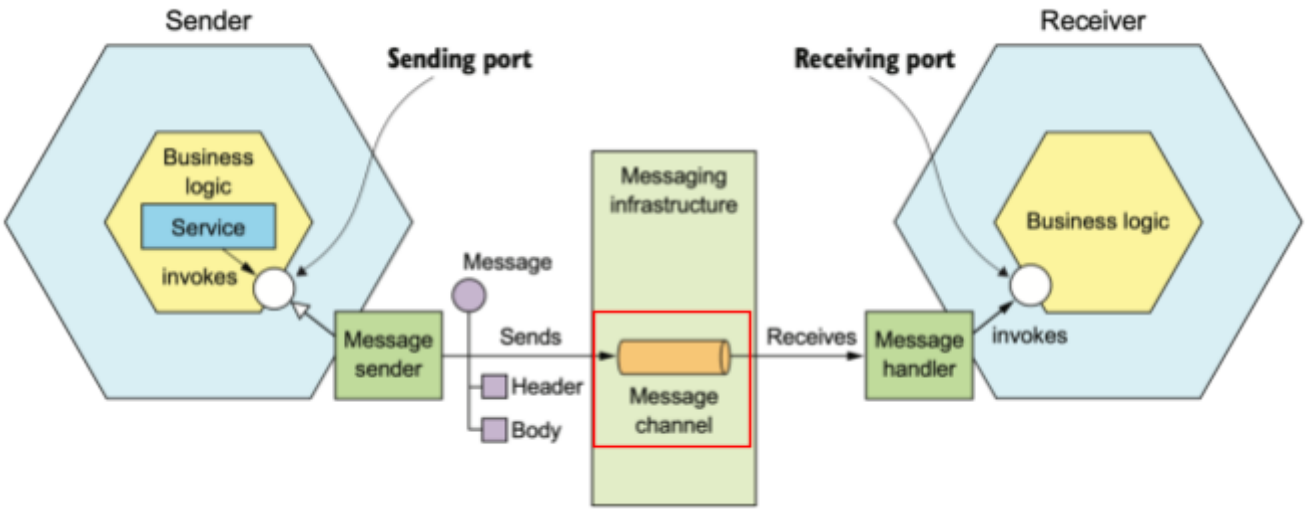
EDA состоит из **четырех основных компонентов**:

1. Генератор событий (event generator). Имеет своей отличительной особенностью функциональность, которая зависит от стиля обработки событий; задачей этого компонента является первичная обработка, выделение из входного потока значимых событий.

2. Канал событий (event channel). Является внутренней магистралью, которая способна обезопасить транспортировку значимых событий из генератора в процессор.

3. Процессор событий (event processor). Соотносит полученные события с имеющимися правилами обработки, оценивает их, вырабатывает правильные решения, в том числе запуск определенных сервисов и бизнес-процессов, осуществляет прямые пересылки сведений о событиях и архивирование, а также направляет сообщения подписчикам.

4. Управление последующими событиями (downstream event-driven activity). Необходимы соответствующие средства для управления, поскольку одно вводное событие может потребовать выполнения целого комплекса последующих действий.



Архитектура Event Driven приложения

Генератор событий(event generator)

Первым логическим слоем является генератор событий, который регистрирует факт и представляет этот факт событием. Поскольку фактом может быть практически все, что может быть воспринято, им может быть и генератор событий. В качестве примера генератором может быть клиент электронной почты, система электронной коммерции или некоторый тип датчика. Преобразование различных данных, полученных от датчиков, в единую стандартизированную форму данных, которые могут быть оценены, является основной проблемой при разработке и реализации этого слоя. Однако, учитывая, что событие является строго декларативным, можно легко применять любые операции трансформации, тем самым устраняя необходимость обеспечения высокого уровня стандартизации.

Канал событий(event channel)

Канал событий — механизм, по которому передается информация от генератора событий к обрабатывающему события механизму или стоку.

Это может быть соединение TCP/IP или входной файл любого типа (простой текст, формат XML, e-mail, и т.д.). В одно и тоже время может быть открыто несколько каналов событий. Обычно из-за требований обработки событий в режиме, приближенном к реальному времени, каналы событий считываются асинхронно. События сохраняются в очереди, ожидая последующей обработки механизмом обработки событий.

Процессор событий (event processor)

Механизм обработки событий является местом, где событие идентифицируется и выбирается соответствующая реакция на него, которая затем выполняется. Это также может привести к созданию ряда утверждений. Например, если пришедшее в механизм обработки событие сообщает, что «Продукт N заканчивается», этот факт может породить реакцию «Заказать продукт N» и «Уведомить персонал».

Управление последующими событиями (downstream event-driven activity).

Здесь проявляются последствия события. Оно может проявляться различными способами и формами; к примеру, сообщение, посланное кому-либо, или приложение, выводящее какое-либо предупреждение на экран. В зависимости от предоставляемого стоком (механизмом обработки событий) уровня автоматизации эти действия могут не потребоваться.

Преимущества EDA:

1) Оптимальная масштабируемость и распределённость.

Одним из самых больших преимуществ EDA является его масштабируемость. Поскольку микросервис в EDA обычно выполняет только одну задачу, реагируя на событие, легко добавлять или убирать компоненты и выявлять узкие места в данных. Если конкретная служба не отвечает должным образом, можно масштабировать только эту службу, вывести ее в автономный режим, исправить и восстановить без нарушения процессов других микрослужб в системе EDA. Также можно добавлять новые сервисы по мере роста спроса. EDA масштабируется и реагирует на потребности и запросы бизнеса по мере их изменения.

2) Асинхронность.

Поскольку архитектуры, основанные на событиях, работают асинхронно, действия других микросервисов не блокируют их и не мешают им. Они могут свободно переходить от одной задачи к другой. Когда происходят события, они реагируют на них и двигаются дальше. Им даже не нужно беспокоиться о том, что одно событие произойдет раньше другого. Они могут ставить события в очередь или буферизировать их, чтобы пользователи не оказывали слишком большого давления на систему, что может привести к блокировке обслуживания.

3) Более простое восстановление после аварий.

EDA также может оперативно реагировать в случае возникновения каких-либо проблем в процессе многочисленных одновременных транзакций данных. Проблемы могут быть отнесены к определенному событию для

затронутой службы. Благодаря воспроизведению прошлых событий, записанных в хранилище событий, EDA может автоматически откатывать и восстанавливать данные из предыдущего состояния, чтобы пользователь не потерял их. Это повышает гибкость и устойчивость к непредсказуемым ситуациям.

4) Отправители и получатели независимы друг от друга.
5) Нет интеграции "точка — точка". Очень легко добавлять в систему новые объекты-получатели.
6) Объекты-получатели могут реагировать на события сразу при их поступлении.
7) Подсистемы получают независимые представления потока событий.

Сложности EDA:

1) Гарантированная доставка. В некоторых системах, особенно в среде Интернета вещей, важно гарантировать доставку событий.
2) Обработка событий в строгом порядке и (или) строго один раз. Каждый тип потребителя обычно выполняется на нескольких экземплярах, чтобы обеспечить надежность и масштабируемость. Это создает некоторые трудности, если события должны обрабатываться в строгом порядке (для каждого типа потребителя) или логика их обработки не является идемпотентной.

Глава 5. Space-Based Architecture

Большинство веб-приложений для бизнеса следуют одному и тому же общему потоку запросов: запрос от браузера попадает на веб-сервер, затем на сервер приложений и, наконец, на сервер базы данных.

Хотя эта схема отлично работает для небольшого числа пользователей, при увеличении нагрузки на пользователей начинают появляться узкие места, сначала на уровне веб-сервера, затем на уровне сервера приложений и, наконец, на уровне сервера баз данных. Обычная реакция на узкие места, возникающие при увеличении нагрузки на пользователей, заключается в увеличении масштаба веб-серверов. Это относительно просто и недорого, и иногда помогает решить проблему узких мест. Однако в большинстве случаев при высокой пользовательской нагрузке масштабирование уровня веб-серверов просто переносит узкое место на сервер приложений.

Масштабирование серверов приложений может быть более сложным и дорогостоящим, чем веб-серверов, и обычно просто перемещает узкое место на сервер базы данных, который еще сложнее и дороже масштабировать. Даже если вы сможете масштабировать базу данных, в итоге вы получите топологию в форме треугольника, где самая широкая часть треугольника - это веб-серверы (их легче всего масштабировать), а самая маленькая - база данных (ее труднее всего масштабировать).

В любом многопользовательском приложении с очень большой одновременной нагрузкой на пользователей база данных обычно является конечным ограничивающим фактором в том, сколько транзакций вы можете обрабатывать одновременно. Хотя различные технологии кэширования и продукты для масштабирования баз данных помогают решить эти проблемы, факт остается фактом: масштабирование обычного приложения для экстремальных нагрузок - очень сложная задача.

Архитектурный подход на основе пространственнаой архитектуры специально разработан для решения проблем масштабируемости и параллелизма. Этот архитектурный паттерн также полезен для приложений с переменным и непредсказуемым объемом одновременных пользователей. Решение проблемы экстремальной и переменной масштабируемости архитектурным путем часто является лучшим подходом, чем попытки масштабировать базу данных или внедрить технологии кэширования в немасштабируемую архитектуру.

Архитектура на основе распределения и масштабирования или пространственная архитектура (Space-Based Architecture - SBA) - это архитектура распределенных вычислений для достижения линейной масштабируемости высокопроизводительных приложений, основанных на состоянии, с использованием парадигмы пространства кортежей (объектов в пространстве разделяемой памяти). SBA минимизирует факторы, ограничивающие масштабирование приложений.

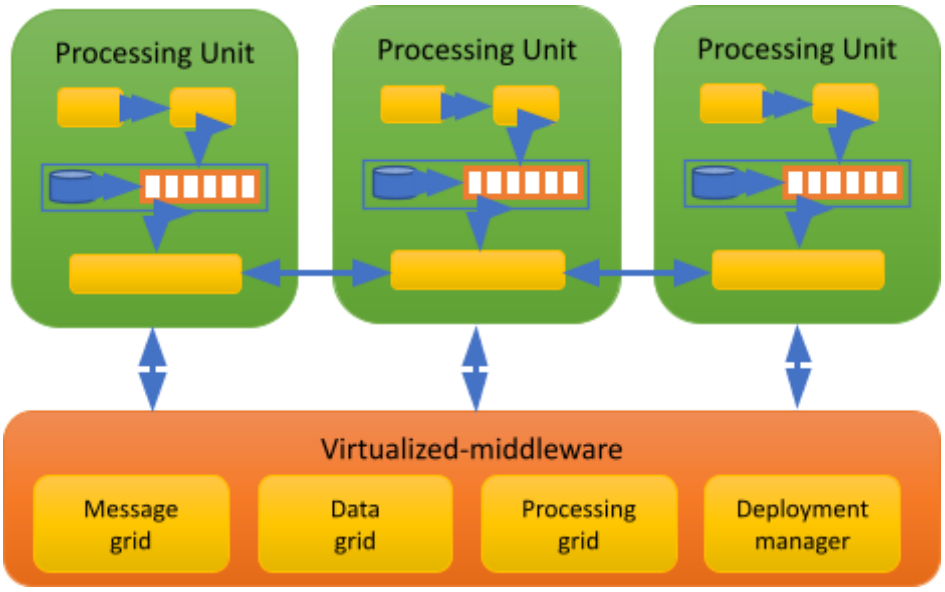
Она следует многим принципам репрезентативной передачи состояния (REST), сервис-ориентированной архитектуры (SOA) и архитектуры, управляемой событиями (EDA), а также элементам grid-вычислений.

SBA тесно связана с Grid Computing. Это вычислительная модель, в которой вычислительные ресурсы (обработка и/или хранение) объединены в параллельную и распределенную систему, охватывающую несколько географических и административных областей.

Grid Computing связаны с системами PeerToPeer и кластерными решениями. Она отличается от PeerToPeer тем, что включает в себя совместное использование аппаратного обеспечения. Он отличается от кластеризации, поскольку кластеризация в основном работает на узлах с идентичным оборудованием и операционными системами, централизованно.

Компоненты архитектуры SBA

В этом архитектурном шаблоне есть два основных компонента: блок обработки (Процессоры - Processing unit - PU) и виртуализированное промежуточное ПО (VM), рисунок.



Базовая схема SBA и ее основные архитектурные компоненты.

Компонент блока обработки содержит компоненты приложения (или части компонентов приложения). Сюда входят веб-компоненты, а также внутренняя бизнес-логика. Содержание блока обработки зависит от типа приложения - небольшие веб-приложения, скорее всего, будут развернуты в одном блоке обработки, в то время как более крупные приложения могут разделить функциональность приложения на несколько блоков обработки в зависимости от функциональных областей приложения. Блок обработки обычно содержит модули приложения, а также сетку данных в памяти и дополнительное асинхронное постоянное хранилище для обеспечения отказоустойчивости. Он также содержит механизм репликации, который используется виртуализированным промежуточным ПО для репликации изменений данных, сделанных одним блоком обработки, на другие активные блоки обработки.

Processing Unit

PU - блок масштабируемости и отказоустойчивости. PU независимы друг от друга, поэтому приложение может масштабироваться путем добавления дополнительных блоков. Модель SBA тесно связана с другими моделями, которые доказали свою успешность в решении проблемы масштабируемости приложений, такими как архитектура "разделяемое ничто" (SN), используемая Google, Amazon.com и другими известными компаниями. Эта модель также применяется многими фирмами в индустрии ценных бумаг для реализации масштабируемых приложений электронной торговли ценными бумагами.

Например, в Java Spring Framework блок обработки строится из контейнера POJO (Plain Old Java Object).

PU способствует разработке и развертыванию приложений, состоящих из автономных, самодостаточных единиц - очень похоже на микросервисную архитектуру, что обеспечивает разделение ответственности.

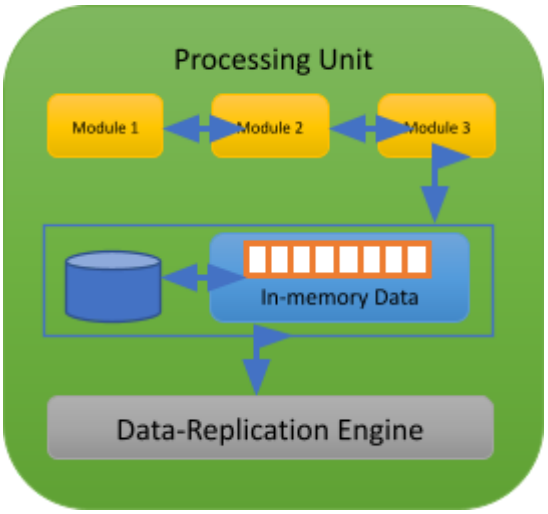
Концепция PU обеспечивает функциональное разделение задач, где развернутые PU в основном независимы, каждый из них имеет свое собственное поведение масштабирования жизненного цикла, в то время как хорошо известный публичный интерфейс раскрывается через удаленные методы или веб-сервис.

Как микросервисы, PU являются мелкозернистыми единицами исполнения. Они предназначены для выполнения одной задачи очень хорошо. Например, PU может инкапсулировать веб-сервис (Web PU), который раскрывает четко определенную функциональность с известной точкой входа, упакованную как одна единица выполнения. Он также может иметь пространственный экземпляр (узел сетки данных в памяти); это делает его stateful PU, где он может сохранять

свое состояние. PU с состоянием обычно развертывается с резервным экземпляром, что делает его высокодоступным и позволяет пережить отказ без потери данных.

PU как микросервис должен придерживаться принципа атомарной бизнес-сущности, которая должна реализовать все для достижения желаемой бизнес-функциональности. Для оптимизации межмикросервисного взаимодействия следует использовать подход map-reduce.

На рисунке показана типичная архитектура блока обработки PU, содержащая прикладные модули, сетку данных в памяти, дополнительное асинхронное хранилище постоянных данных для отказоустойчивости и механизм репликации данных.



Компонент блока обработки PU.

Виртуальное промежуточное ПО - Virtualized-Middleware

Особенность SBA заключается в виртуализированных компонентах промежуточного ПО и сетке данных в памяти, содержащихся в каждом блоке обработки.

Компонент Virtualized-Middleware (VM) управляет обслуживанием и коммуникациями. Он содержит компоненты, которые контролируют различные аспекты синхронизации данных и обработки запросов.

В **состав VM** входят:

- Сетка обмена сообщениями
- Сетка данных

- Сетка обработки
- Менеджер развертывания.

Эти компоненты, могут быть написаны на заказ или приобретены как продукты сторонних производителей.

Виртуализированное промежуточное ПО по сути является контроллером архитектуры и управляет запросами, сеансами, репликацией данных, распределенной обработкой запросов и развертыванием блоков обработки.

Состав Виртуализированного промежуточного ПО – VM.

Сетка обмена сообщениями

Сетка обмена сообщениями, управляет входными запросами и информацией о сеансах. Когда запрос поступает в компонент виртуализированного промежуточного ПО, компонент сетки сообщений определяет, какие активные компоненты обработки доступны для получения запроса, и направляет запрос в один из этих блоков обработки. Сложность системы обмена сообщениями может варьироваться от простого алгоритма круговой выборки до более сложного алгоритма next-available, который отслеживает, какой запрос обрабатывается тем или иным процессинговым модулем.

Сетка данных

Компонент сетки данных является, пожалуй, самым важным и ключевым компонентом в этом шаблоне. Сетка данных взаимодействует с механизмом репликации данных в каждом блоке обработки для управления репликацией данных между блоками обработки, когда происходит обновление данных. Поскольку сетка сообщений может направить запрос на любой из доступных блоков обработки, очень важно, чтобы каждый блок обработки содержал абсолютно одинаковые данные в своей сетке данных в памяти. Хотя на рисунке 1 показана синхронная репликация данных между блоками обработки, в действительности она выполняется параллельно асинхронно и очень быстро, иногда синхронизация данных завершается за считанные микросекунды (одну миллионную долю секунды).

Сетка обработки

Сетка обработки, является дополнительным компонентом виртуализированного промежуточного ПО, который управляет распределенной обработкой запросов при наличии нескольких блоков обработки, каждый из которых обрабатывает свою часть приложения. Если поступает запрос,

требующий координации между типами блоков обработки (например, блок обработки заказов и блок обработки клиентов), именно сетка обработки является посредником и организует запрос между этими двумя блоками обработки.

Менеджер развертывания

Компонент deployment-manager управляет динамическим запуском и выключением блоков обработки в зависимости от условий нагрузки. Этот компонент постоянно отслеживает время отклика и нагрузку на пользователя, запускает новые блоки обработки при увеличении нагрузки и отключает блоки обработки при снижении нагрузки. Это критически важный компонент для достижения переменной масштабируемости приложения.

Система на основе SBA - сложный и дорогой в реализации паттерн. Это хороший выбор архитектуры для небольших веб-приложений с переменной нагрузкой (например, сайты социальных сетей, сайты торгов и аукционов). Однако он не очень хорошо подходит для традиционных крупномасштабных приложений реляционных баз данных с большим объемом оперативных данных.

Хотя модель архитектуры на основе пространства не требует централизованного хранилища данных, его обычно включают для выполнения первоначальной загрузки сетки данных в память и асинхронного сохранения обновлений данных, сделанных блоками обработки. Также распространенной практикой является создание отдельных разделов, изолирующих изменчивые и широко используемые транзакционные данные от неактивных данных, чтобы уменьшить объем памяти, занимаемый сеткой данных в памяти в каждом блоке обработки. транзакционные данные от неактивных данных, чтобы сократить объем памяти сетки данных в памяти в каждом блоке обработки.

Важно отметить, что хотя альтернативное название этого паттерна - облачная архитектура, блоки обработки (а также виртуализированное промежуточное ПО) не обязательно должны располагаться на облачных высокопроизводительных сервисах или PaaS (платформа как сервис). Они могут с таким же успехом размещаться на локальных серверах, что является одной из причин названия "архитектура на основе пространства".

Масштабирование

Как сохранить хорошую производительность даже в случае определенного увеличения параметров нагрузки? Ответ на этот вопрос подводит к изучению вопроса масштабируемости.

Масштабируемость (scalability) — способность системы справляться с возросшей нагрузкой.

В основном масштабирование систем выполняется по двум причинам:

1) Для того, чтобы легче было справиться со сбоями: если мы переживаем за отказ какого-либо компонента, то здесь поможет наличие такого же дополнительного компонента.

2) Для повышения производительности, что позволяет либо справиться с более высокой нагрузкой, либо снизить время отклика, либо достичь обоих результатов.

Рассмотрим **ряд наиболее распространенных технологий масштабирования**, которыми можно будет воспользоваться:

1) Наращивание мощностей

От наращивания мощностей некоторые операции могут только выиграть. Более броизводительный вычислительный комплекс способен уменьшить задержки и повысить пропускную способность, позволяя выполнять больший объем работ за меньшее время. Но такая разновидность масштабирования, которую часто называют *вертикальным масштабированием*, может быть слишком затратной: иногда один большой сервер может стоить намного больше, чем два небольших сервера сопоставимой мощности, особенно когда вы начнете получать понастоящему большие машины.

Иногда само программное обеспечение не способно освоить доступные дополнительные ресурсы. Более крупные машины зачастую предоставляют в наше распоряжение больше ядер центральных процессоров, но подчас у нас нет программных средств, позволяющих использовать такое преимущество. Еще одна проблема заключается в том, что такая разновидность масштабирования не в состоянии внести весомый вклад в устойчивость сервера, если у нас только одна машина.

2) Разделение рабочих нагрузок

Наличие единственного микросервиса на каждом хосте, безусловно, предпочтительнее модели, предусматривающей наличие на хосте сразу нескольких микросервисов. Но изначально с целью снижения стоимости оборудования или упрощения управления хостом многие принимают решение о сосуществовании нескольких микросервисов на одной физической машине.

Поскольку микросервисы запускаются в независимых процессах, обменивающихся данными по сети, задача последующего их перемещения на собственные хосты с целью повышения пропускной способности и масштабирования не представляет особой сложности. Такое перемещение может

повысить устойчивость системы, поскольку сбой одного хоста повлияет на ограниченное количество микросервисов.

3) Распределение риска

Один из способов масштабирования с целью повышения отказоустойчивости заключается в выдаче гарантий того, что вы не поместили на одном хосте сразу несколько сервисов, где сбой окажет влияние на работу сразу нескольких сервисов.

Еще одной распространенной разновидностью сокращения вероятности сбоев является гарантия того, что не все ваши сервисы запускаются на одной и той же стойке дата-центра, или того, что сервисы распределены по более чем одному дата-центру. Если вы имеете дело с основным поставщиком услуг, то важно знать о предложении и планировании им соответствующего соглашения об уровне предоставления услуг (SLA). Если для вас допустимо не более четырех часов сбоев в квартал, а хостинг-провайдер может гарантировать всего лишь не более восьми часов, то вам придется либо пересмотреть SLA, либо придумать альтернативное решение.

AWS, к примеру, имеет региональную форму распределения, которую можно рассматривать как отдельные облака. Каждый регион, в свою очередь, разбит на две и более зоны доступности (AZ). Эти зоны в AWS являются эквивалентом дата-центра. Важно, чтобы сервисы были распределены по нескольким зонам доступности, поскольку инфраструктура AWS не дает гарантий доступности отдельно взятого узла или даже всей зоны доступности. Для своих вычислительных услуг эта инфраструктура предлагает только 99,95 % безотказной работы за заданный месячный период во всем регионе, поэтому внутри отдельно взятого региона рабочую нагрузку следует распределить по нескольким доступным зонам. Некоторых такие условия не устраивают, и вместо этого они запускают свои сервисы, также распределяя их по нескольким регионам.

4) Балансировка нагрузки

Когда сервису нужна отказоустойчивость, вам понадобятся способы обхода критических мест сбоев. Для типичного микросервиса, выставляющего синхронную конечную HTTP-точку, наиболее простым способом решения этой задачи будет использование нескольких хостов с запущенными на них экземплярами микросервиса, находящимися за балансировщиком нагрузки. Потребители микросервиса не знают, связаны они с одним его экземпляром или с сотней таких экземпляров.

Существуют балансировщики нагрузки всех форм и размеров, от больших и дорогих аппаратных приспособлений до балансировщиков на основе программных средств типа mod_proxy. У всех них общие основные возможности. Они распределяют поступающие к ним вызовы между несколькими экземплярами на основе определенного алгоритма, устраняя экземпляры, утратившие работоспособность, но не теряя при этом надежды на их возвращение при восстановлении нормального режима работы.

Балансировщики нагрузки позволяют нам добавлять дополнительные экземпляры микросервисов незаметно для любых потребителей сервиса. Это дает нам более широкие возможности управления нагрузкой и в то же время уменьшает влияние сбоя на одном из хостов. Но у многих, если не у большинства микросервисов будут какие-нибудь постоянные хранилища данных, возможно, база данных, находящаяся на другой машине. При наличии нескольких экземпляров микросервисов на разных машинах, но только одного хоста с запущенным экземпляром базы данных мы обрекаем эту базу данных на роль единого источника сбоев. Схемы, позволяющие справиться с этой проблемой, будут рассмотрены чуть позже.

5) Системы на основе исполнителей

Применение балансировщиков не является единственным способом разделения нагрузки среди нескольких экземпляров сервиса и уменьшения их хрупкости. В зависимости от характера операций столь же эффективной может быть и система на основе исполнителей. Здесь вся коллекция экземпляров действует с некоторым общим отставанием в работах. Это может быть целый ряд Hadoop-процессов или, возможно, некоторое количество процессов, прослушивающих общую очередь работ. Операции такого типа хорошо подходят для формирования пакетов работ или асинхронных заданий.

Эта модель также хорошо работает при *пиковых* нагрузках, где по мере возрастания потребностей могут запускаться дополнительные экземпляры для соответствия поступающей нагрузке. Пока сама очередь работ будет сохранять устойчивость, эта модель может использовать масштабирование для повышения как пропускной способности работ, так и отказоустойчивости, поскольку становится проще справиться с влиянием отказавшего (или отсутствующего) исполнителя. Работа займет больше времени, но ничего при этом не потеряется.

Хотя в системах на основе исполнителей самим исполнителям высокая надежность и не нужна, система, содержащая предназначенную для выполнения работу. должна быть надежной. Справиться с этим можно, к примеру запустив круглосуточный брокер сообщений или такую систему, как Zookeeper.

Преимущества такого подхода заключаются в том, что при использовании для достижения этих целей существующих программных средств наиболее сложную задачу за нас выполняет кто-то другой. Но нам по-прежнему требуется знать, как настроить и обслуживать эти системы, добиваясь от них безотказной работы.

При необходимости масштабировать систему в расчете на более высокую нагрузку, то простейший способ — это купить более мощную машину - *вертикальное масштабирование* (vertical scaling, scaling up). Можно объединить много процессоров, чипов памяти и жестких дисков под управлением одной операционной системы, а быстрые соединения между ними позволят любому из процессоров обращаться к любой части памяти или диска. В подобной *архитектуре с разделяемой памятью* (shared-memory architecture) можно рассматривать все компоненты как единую машину.

Вертикальное масштабирование

Архитектуры с разделением ресурсов

Главная проблема подхода с разделяемой памятью состоит в том, что стоимость растет быстрее, чем линейно: машина с вдвое большим количеством CPU, вдвое большим количеством памяти и двойным объемом дисков стоит отнюдь не вдвое дороже. Кроме того, вследствие узких мест вдвое более мощная машина не обязательно сможет справиться с двойной нагрузкой.

Архитектура с разделяемой памятью обеспечивает ограниченную отказоустойчивость — в высокопроизводительных машинах есть возможность горячей замены компонентов (дисков, модулей памяти и даже CPU без отключения машины), — но она строго ограничена одной географической точкой.

Другой подход: *архитектура с разделяемым дисковым накопителем* (shared-disk architecture), при которой применяется несколько машин с отдельными CPU и оперативной памятью, но данные хранятся в массиве дисков, совместно используемых всеми машинами, подключенными с помощью быстродействующей сети. Такая архитектура применяется при складировании данных, но конкуренция и накладные расходы на блокировки ограничивают масштабируемость этого подхода.

Вертикальное масштабирование (Scale up)— увеличение производительности каждого компонента системы с целью повышения общей производительности. Масштабируемость в этом контексте означает возможность заменять в существующей вычислительной системе компоненты более мощными и быстрыми по мере роста требований и развития технологий. Это самый простой

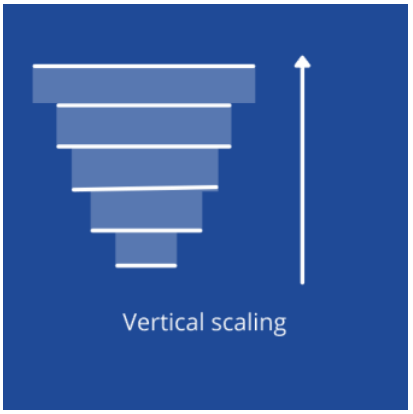
способ масштабирования, так как не требует никаких изменений в прикладных программах, работающих на таких системах.

Вертикальное масштабирование - добавление вычислительных ресурсов в какой-то уже используемый сервер. Обычно это процессоры или оперативная память.

Еще этот тип архитектуры называется - Архитектура с разделением ресурсов. Ресурсы ЭВМ делятся между выполняемыми задачами.

Вертикальное масштабирование используется, когда необходимо быстро реагировать на проблемы с производительностью, которые нельзя решить с помощью классической методики оптимизации базы данных, например изменения запросов или индексирования.

Вертикальное масштабирование подразумевает добавление большего количества ресурсов в отдельный концентратор и добавление дополнительного процессора, оперативной памяти и диска для адаптации к растущей ответственности. По сути, вертикальное масштабирование позволяет вам создать ваше нынешнее оборудование или ограничить возможности программирования, однако помните, что вы можете просто расширить его до самых дальних пределов вашего сервера.



Вертикальное масштабирование помогает справиться с пиками в рабочих нагрузках, когда текущий уровень производительности не может удовлетворить все требования. Вертикальное увеличение масштаба позволяет добавлять дополнительные ресурсы, чтобы легко адаптироваться к пиковым рабочим нагрузкам. Затем, если ресурсы больше не нужны, можно выполнить вертикальное уменьшение масштаба, чтобы вернуться к исходному состоянию и сократить затраты на облако.

Вертикальное масштабирование **выполняется в следующих случаях**:

- Рабочие нагрузки достигают определенного ограничения производительности, например ограничения, касающегося ЦП или операций ввода-вывода.
- Вам нужно быстро реагировать, чтобы устранить проблемы с производительностью, которые нельзя решить путем классической оптимизации базы данных.
- Вам требуется решение, позволяющее изменять уровни служб, чтобы адаптироваться к изменению требований к задержке.

Вертикальное масштабирование отлично подходит для бизнес критических приложений с их огромными требованиями к ресурсам. Критичные базы данных, огромные ERP системы, системы аналитики больших данных, JAVA приложения и так далее и тому подобное получат прямую выгоду от вертикального масштабирования.

Достоинства	Недостатки
• Согласованность данных. • Никаких дополнительных ИТ-ресурсов не требуется.	• Аппаратный предел. • Стоимость.

Горизонтальное масштабирование

Архитектуры без разделения ресурсов

Архитектуры без разделения ресурсов (shared-nothing architectures), известные под названием *горизонтального масштабирования* (horizontal scaling, scaling out). При этом подходе каждый компьютер или виртуальная машина, на которой работает база данных, называется *узлом* (node). Все узлы используют свои CPU, память и диски независимо друг от друга. Согласование узлов выполняется на уровне программного обеспечения с помощью обычной сети.

Для систем без разделения ресурсов не требуется никакого специального аппаратного обеспечения, так что можно применять машины, имеющие наилучшее соотношение «цена/производительность». При желании можно распределить данные по многим географическим регионам и таким образом снизить задержку для пользователей и потенциально сделать систему устойчивой к потере целого ЦОДа. Благодаря облачному развертыванию виртуальных машин вашей компании не нужно работать в масштабах Google:

даже маленькие компании могут позволить себе межрегиональную распределенную архитектуру.

Архитектуры без разделения ресурсов требуют от разработчика приложения наибольшей осторожности. В случае распределенных по нескольким узлам данных необходимо осознавать ограничения и компромиссы подобных распределенных систем — БД не может сама по себе избавить от этих особенностей.

Хотя у распределенных архитектур без разделения ресурсов есть много достоинств, обычно они приводят к усложнению приложений, а также иногда ограничивают выразительность используемых моделей данных. В некоторых случаях простая однопоточная программа может продемонстрировать намного лучшую производительность, чем кластер с сотней ядер CPU. С другой стороны, системы без разделения ресурсов могут быть высокопроизводительными.

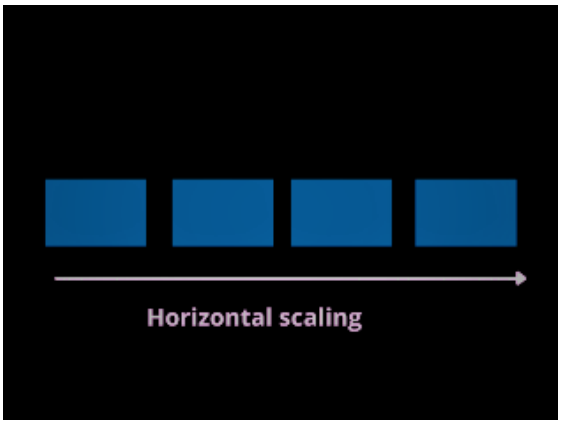
Горизонтальное масштабирование (Scale out) — разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам (или их группам), и (или) увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Масштабируемость в этом контексте означает возможность добавлять к системе новые узлы, серверы для увеличения общей производительности. Этот способ масштабирования может требовать внесения изменений в программы, чтобы программы могли в полной мере пользоваться возросшим количеством ресурсов.

Горизонтальное масштабирование - добавление новых ресурсов в инфраструктуру, например, серверов в кластер.

Еще этот тип архитектуры называется - Архитектура без разделения ресурсов. Ресурсы ЭВМ выделяются на одну задачу.

Разработчики приложений начинают применять горизонтальное масштабирование, если им не удается получить достаточно ресурсов для рабочих нагрузок даже на самых высоких уровнях производительности.

При горизонтальном масштабировании данные разбиваются на несколько баз данных (или сегментов) между серверами. Масштаб каждого сегмента можно вертикально увеличивать или уменьшать по отдельности.



Горизонтальное масштабирование подразумевает добавление большего количества машин в пул активов, в отличие от простого добавления активов путем вертикального масштабирования. Это эквивалентно масштабированию за счет добавления большего количества машин в пул или активы; однако вместо того, чтобы добавлять больше ресурсов, процессоров или оперативной памяти, вы уменьшаете размер существующих платформ.

Как шардирование (секционирование) данных повышает масштабируемость.

При вертикальном увеличении масштаба отдельной базы данных путем добавления таких ресурсов, как виртуальные машины, в конечном итоге будет достигнуто физическое ограничение оборудования. Каждая секция данных размещается на отдельном сервере, поэтому, если разделить данные на несколько сегментов, можно практически без ограничений горизонтально увеличивать масштаб системы.

Некоторые типы технологий баз данных, особенно нереляционные базы данных или базы данных NoSQL, разрабатываются с уникальными возможностями горизонтального увеличения масштаба данных путем сегментирования. Это позволяет таким базам данных управлять объемными, несвязанными, неопределенными или быстро изменяющимися данными.

Кроме того, некоторые реляционные службы баз данных (SQL), изначально предлагавшие услуги вертикального увеличения или уменьшения масштаба, начинают предоставлять интересные возможности, позволяя достичь уровня масштабируемости нереляционных баз данных. Службы гипермасштабирования, такие как Гипермасштабирование Базы данных SQL Microsoft Azure, дают пользователям возможность быстро масштабировать хранилище до 100 ТБ, обеспечивают ориентированную на облако гибкую архитектуру, позволяя увеличивать объем хранилища в соответствии с потребностями, а также включать почти моментальные операции резервного копирования и быстрые операции восстановления баз данных за несколько минут.

Горизонтальное масштабирование **выполняется в следующих случаях:**

- У вас есть географически распределенные приложения, каждое из которых должно обращаться к части данных в регионе. Каждое приложение будет обращаться только к сегменту, связанному с этим регионом, не затрагивая другие сегменты.
- У вас есть глобальный сценарий сегментирования (например, балансировка нагрузки) с большим количеством географически распределенных клиентов, которые вставляют данные в выделенные им сегменты.
- Если ограничения производительности превышаются даже на самых высоких уровнях производительности службы или если ваши данные не помещаются в одну базу данных.

Так как стоимость оборудования продолжает снижаться, а производительность расти то дешёвые, commodity (широкого потребления) сервера являются идеальным решением для горизонтального масштабирования, и могут быть собраны в большие кластера для объединения вычислительных ресурсов.

Достоинства	Недостатки
● Хорошо масштабируется. ● Экономически эффективно.	● Несогласованность данных. ● Требуются дополнительные ИТ-ресурсы.

Реагирование на нагрузку

Подходящая для определенного уровня нагрузки архитектура, вероятно, не сможет справиться с десятикратным ее увеличением. Если вы имеете дело с быстрорастущим сервисом, то, вероятно, вам придется пересматривать архитектуру при каждом возрастании нагрузки на порядок или даже еще чаще.

Системы, которые способны работать на отдельной машине, обычно проще, а высококлассные машины могут оказаться весьма недешевыми, так что при высокой рабочей нагрузке часто нельзя избежать горизонтального масштабирования. На практике хорошая архитектура обычно представляет собой прагматичную смесь этих подходов: например, может оказаться проще и

дешевле использовать несколько весьма мощных компьютеров, чем много маленьких виртуальных машин.

Некоторые системы способны *адаптироваться*, то есть умеют автоматически добавлять вычислительные ресурсы при обнаружении прироста нагрузки, в то время как другие системы необходимо масштабировать вручную (человек анализирует производительность и решает, нужно ли добавить в систему дополнительные машины). Способные к адаптации системы полезны в случае непредсказуемого характера нагрузки, но масштабируемые вручную системы проще и доставляют меньше неожиданностей при эксплуатации.

Хотя распределение сервисов без сохранения состояния по нескольким машинам особых сложностей не представляет, преобразование информационных систем с сохранением состояния из одноузловых в распределенные может повлечь значительные сложности. Поэтому до недавнего времени считалось разумным держать базу данных на одном узле (вертикальное масштабирование) до тех пор, пока стоимость масштабирования или требования по высокой доступности не заставят сделать ее распределенной.

По мере усовершенствования инструментов и абстракций для распределенных систем это мнение подвергается изменениям, по крайней мере для отдельных видов приложений. Вполне возможно, что распределенные информационные системы в будущем станут «золотым стандартом» даже для сценариев применения, в которых не идет речь об обработке больших объемов данных или трафика.

Архитектура крупномасштабных систем обычно очень сильно зависит от приложения — не существует такой вещи, как одна масштабируемая архитектура на все случаи жизни (на сленге именуемая *волшебным масштабирующим соусом*). Проблема может заключаться в количестве чтений, количестве записей, объеме хранимых данных, их сложности, требованиях к времени отклика, паттернах доступа или (зачастую) какой-либо смеси всего перечисленного, а также во многом другом.

Например, система, рассчитанная на обработку 100 000 з/с по 1 Кбайт каждый, выглядит совсем не так, как система, рассчитанная на обработку 3 з/мин. по 2 Гбайт каждый — хотя пропускная способность обеих систем в смысле объема данных одинакова.

Хорошая масштабируемая для конкретного приложения архитектура базируется на допущениях о том, какие операции будут выполняться часто, а какие — редко, то есть на параметрах нагрузки. Если эти допущения окажутся неверными, то работа архитекторов по масштабированию окажется в лучшем

случае напрасной, а в худшем — приведет к обратным результатам. На ранних стадиях обычно важнее быстрая работа существующих возможностей в опытном образце системы или непроверенном программном продукте, чем его масштабируемость под гипотетическую будущую нагрузку.

Несмотря на зависимость от конкретного приложения, масштабируемые архитектуры обычно создаются на основе универсальных блоков, организованных по хорошо известным паттернам.

Балансировщик нагрузки

Балансировщик нагрузки (Load Balancer) — сервис, помогающий серверам эффективно перемещать данные, оптимизирующий использование ресурсов доставки приложений и предотвращающий перегрузки. Он управляет потоком информации между локальным или облачным хранилищем и конечным устройством пользователя.

Этот сервис проводит непрерывные проверки работоспособности серверов, чтобы убедиться в их работоспособности. При необходимости подсистема балансировки удаляет неисправные серверы из пула.

Входящие в состав балансировщика контроллеры доставки приложений (ADC) предлагают множество дополнительных функций — шифрование, аутентификацию и межсетевой экран веб-приложений, создавая тем самым единую точку контроля для защиты, управления и мониторинга веб-сервисов.

К таким **дополнительным возможностям** относятся:

- функция разгрузки — защищает от распределенных атак типа «отказ в обслуживании» (DDoS);
- функция прогнозной аналитики — определяет узкие места трафика до того, как они возникнут;
- функция запуска новых виртуальных хранилищ данных при превышении лимитов входящего трафика.
- Обнаружение служб: Какие бэкенды доступны в системе? Каковы их адреса (например, как должен работать балансировщик нагрузки)?
- Проверка работоспособности: Какие бэкенды в настоящее время здоровы и доступны для приема запросов?
- Балансировка нагрузки: Какой алгоритм следует использовать для балансировки отдельных запросов по здоровым бэкендам?

Преимущества использования балансировщика:

- Именование абстракции: Нет необходимости в том чтобы каждый клиент знал о каждом бэкенде (service discovery). Клиент может обращаться к балансировщику нагрузки через предопределенный механизм, а затем действие разрешения имен можно делегировать в балансировщик нагрузки. Предопределенные механизмы включают встроенные библиотеки и хорошо известные DNS/IP/port и будут обсуждаться более подробно ниже.

- Отказоустойчивость: Благодаря проверке работоспособности и различным алгоритмическим методам балансировщик нагрузки может эффективно маршрутизировать вокруг неработоспособного или перегруженного бэкенда. Это означает, что администратор может спокойно, без спешки устранить проблему на нездоровом узле.

- Стоимость и производительность: Распределенные системные сети редко бывают однородными. Наиболее вероятно, что система будет охватывать несколько сетевых зон и регионов. Интеллектуальная балансировка нагрузки может максимально увеличить трафик запросов в зонах, что увеличивает производительность (меньше латентности) и снижает общую стоимость системы (меньше полосы пропускания и прокладки оптоволокна, требуемого между зонами).

Глава 6. Распределенные хранилища данных

Современный мир генерирует всё больше информации. Какая-то её часть мимолётна и утрачивается так же быстро, как и собирается. Другая должна храниться дольше, а иная и вовсе рассчитана «на века» — по крайней мере, так нам видится из настоящего. Информационные потоки оседают в дата-центрах с такой скоростью, что любой новый подход, любая технология, призванные удовлетворить этот бесконечный «спрос», стремительно устаревают.

В крупных организациях системы хранения данных занимают значительную долю стоимости ИТ-инфраструктуры (по оценкам специалистов – до 25%). Эта цифра может существенно вырасти. Причины – рост объема данных и увеличение потребности в емкостях систем хранения данных (СХД), в том числе из-за законов, которые обязывают эти данные хранить. В то же время компании активно стараются экономить ИТ-бюджеты, что вынуждает их находиться в постоянном поиске наиболее выгодных технологических решений, которые бы позволили сократить эти расходы не в ущерб качеству сервиса. Это же относится к хранению и обработке данных.

Распределенное хранилище данных

Распределенное хранилище данных - это компьютерная сеть, в которой информация хранится более чем на одном узле, часто реплицируемым образом. Обычно он специально используется для обозначения либо распределенной базы данных, в которой пользователи хранят информацию на *нескольких узлах*, либо компьютерной сети, в которой пользователи хранят информацию на *нескольких узлах одноранговой сети*.

Распределенная база данных - это набор логически связанных между собой разделяемых данных (и их описаний), которые физически распределены в некоторой компьютерной сети. Он может храниться на нескольких компьютерах, расположенных в одном и том же физическом месте (например, в центре обработки данных); или, возможно, рассредоточен по сети взаимосвязанных компьютеров.

В отличие от параллельных систем, в которых процессоры тесно связаны и составляют единую систему баз данных, распределенная система баз данных состоит из слабо связанных узлов, которые не имеют общих физических компонентов.

Системные администраторы могут распределять коллекции данных (например, в базе данных) по нескольким физическим местоположениям. Распределенная база данных может размещаться на организованных сетевых

серверах или децентрализованных независимых компьютерах в Интернете, в корпоративных интранетах или внешних сетях или в других сетях организации. Поскольку распределенные базы данных хранят данные на нескольких компьютерах, распределенные базы данных могут повысить производительность на рабочих местах конечных пользователей, позволяя обрабатывать транзакции на многих машинах, а не ограничиваться одной.

Распределенные базы данных обычно являются нереляционными базами данных, которые обеспечивают быстрый доступ к данным на большом количестве узлов. Некоторые распределенные базы данных предоставляют широкие возможности для запросов, в то время как другие ограничены семантикой хранилища ключевых значений.

Примерами ограниченных распределенных баз данных являются от Google, которая намного больше, чем распределенная файловая система или одноранговая сеть, Динамическая система Amazon и хранилище Microsoft Azure.

Поскольку возможность произвольного запроса не так важна, как доступность, разработчики распределенных хранилищ данных увеличили последние за счет согласованности. Но высокоскоростной доступ для чтения/записи приводит к снижению согласованности, поскольку невозможно гарантировать согласованность и доступность в разделенной сети, как указано в теореме CAP.

Распределенная обработка

Очень важно понимать различия между распределенными СУБД и средствами распределенной обработки данных.

Распределенная обработка - обработка с использованием централизованной базы данных, доступ к которой может осуществляться с различных компьютеров сети.

Ключевым моментом в определении распределенной СУБД является утверждение, что система работает с данными, физически распределенными в сети. Если данные хранятся централизованно, то даже в том случае, когда доступ к ним обеспечивается для любого пользователя по сети, эта система просто поддерживает распределенную обработку, но не может рассматриваться как распределенная СУБД.

Распределенные БД (DDB) и Распределённые СУБД

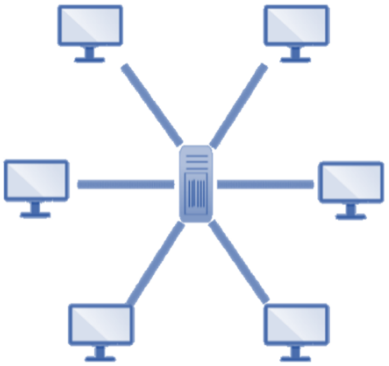
Единое хранилище информации (не произвольная коллекция документов), индивидуально сберегаемой на разных узлах виртуальной конструкции,

являющейся дифференцированной файловой системой, называется распределенная база данных (*Distributed Database* DDB или РБД).

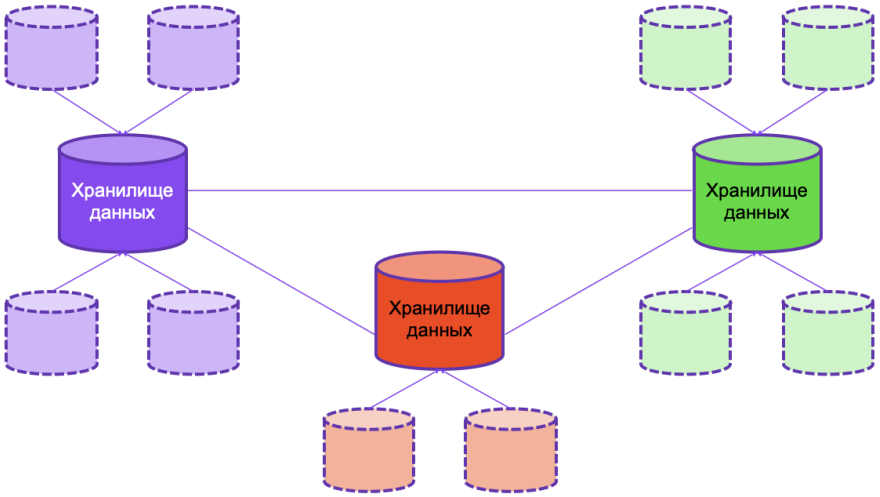
Распределенная база данных - набор логически связанных между собой совокупностей разделяемых данных (и их описаний), которые физически распределены в некоторой компьютерной сети.

Информацию можно квалифицировать, как DDB лишь в том случае, если она взаимосвязана реляционной конструкцией с доступом, который предоставляет высокоуровневый единый интерфейс. Показатель уровня реплицированности у подобных хранилищ данных может быть разнообразным: от отсутствия какой-либо возможности копировать данные до всецелого информационного дубляжа абсолютно во всех дифференцированных прототипах – например, технологии блокчейн.

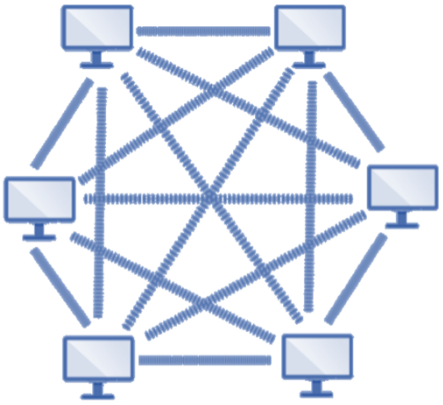
Централизованное хранилище (рисунок 6) данных может применяться для таких сред, где средоточием данных компании по-прежнему является мэйнфрейм. В то же время нецелесообразно было бы сохранять централизованную архитектуру информационных приложений в ситуациях, когда операционные приложения (и соответствующие данные) базируются на распределенных вычислительных системах.



Архитектура централизованного хранилища данных



Архитектура распределенного хранилища данных



Децентрализованное хранилище

Механизм прозрачности

Дифференцирование информационного массива по огромному количеству нод (узлов), таким образом, чтобы для пользователя эта операция прошла незамеченной, называется термином «прозрачность».

В свою очередь механизм распределения информации по вычислительным устройствам, включенным в общую сеть, является главной функцией для достижения децентрализации данных относительно непосредственно самой среды хранилища.

Это достигается, благодаря нескольким **типам прозрачности**:

- системная (на уровне технологий);
- репликации;

- фрагментации;
- доступа (осуществляется, как в централизованное хранилище).

В идеале, полная прозрачность подразумевает одинаковый язык запроса, как в случае с DDB, так с централизованной распределенной базой данных.

Распределенная СУБД (РСУБД) - это программный комплекс, предназначенный для управления распределенными базами данных и позволяющий сделать распределенность информации прозрачной для конечного пользователя.

Система управления распределенными базами данных (СУРБД) состоит из единой логической базы данных, разделенной на некоторое количество фрагментов. Каждый фрагмент базы данных сохраняется на одном или нескольких компьютерах, которые соединены между собой линиями связи, и каждый из которых работает под управлением отдельной СУБД. Любой из сервисов способен независимо обрабатывать запросы пользователей, требующие доступа к локально сохраняемым данным (что создает определенную степень локальной автономии), а также способен обрабатывать данные, сохраняемые на других компьютерах сети.

Пользователи взаимодействуют с распределенной базой данных через приложения. Приложения могут быть классифицированы следующим образом: приложения, которые не требуют доступа к данным на других сайтах (локальные приложения), и приложения, которые требуют подобного доступа (глобальные приложения).

В распределенном хранилище данных управление метаданными может усложниться, хотя это и необязательно. Распределенное хранилище данных, построенное на основе распределенной СУБД (РСУБД), разумеется, будет использовать метаданные РСУБД для компонентов хранилища данных. В других случаях, например в «доморощенных» распределенных хранилищах данных, ответственность за определение метаданных и управление ими возлагается на проектировщиков и специалистов по реализации. Как и в средах централизованных хранилищ данных, это может также осуществляться средствами репозиториев.

Распределенные хранилища файлов

Дифференцированные файловые банки делятся на **несколько видов**:

- облачные хранилища;
- торренты.

Назвать децентрализованной распределенную базу данных облачного хранилища нельзя, так как всегда в подобной конструкции присутствует оператор, который арендует необходимое оборудование, либо обходится собственными мощностями.

Именно частичная централизация архитектуры является основным минусом облачных хранилищ.

Однако требуется именно такая DDB, которая была бы устойчивой к дифференциации, а также доступной – ведь пользователь заинтересован получать нужный ответ (пускай неверный) в максимально сжатые сроки. Такая требовательность существенно ограничивает выбор до группы хранилищ информации NoSQL.

К достоинствам этой категории БД относится высокая скорость обработки информации, линейное масштабирование с объемом хранилища, устойчивость к недоступности некоторой части копий, продуманные реализации. Минус один, но какой – слабая защита от проблемы «византийских генералов».

С торрентами ситуация противоположная. Хранение файлов – прерогатива обычных пользователей, которые, таким образом, вносят свою лепту в развитие архитектуры, предлагая сервис по сбережению данных. Взамен появляется возможность скачивать интересующие документы у других участников виртуальной системы.

Это не исключает факт появления игроков с мощным оборудованием (серверами), поэтому, как не крути, все равно остается риск централизации цифровой конструкции. Но даже не в этом заключается основополагающий недостаток торрентов. Их главный минус – отсутствие стабильных и, что немаловажно, ощутимых привилегий (бонусов) для участников системы.

Преимущества и недостатки РСУБД

Системы с распределенными базами данных имеют **преимущества** перед традиционными централизованными системами баз данных:

1) Отражение глобальной структуры организации.

Крупные организации, как правило, имеют множество отделений, которые могут находиться в разных концах страны и даже за ее пределами. Вполне логично будет предположить, что используемая компанией база данных должна быть распределена между ее отдельными офисами. В подобной базе данных персонал отделения сможет выполнять необходимые ему локальные запросы. Руководству компании может потребоваться выполнять глобальные запросы,

предусматривающие получение доступа к данным, сохраняемым во всех существующих отделениях компании.

2) Разделимость и локальная автономность.

Географическая распределенность организации может быть отражена в распределении ее данных, причем пользователи одного сайта смогут получать доступ к данным, сохраняемым на других сайтах. Данные могут быть помещены на тот сайт, на котором зарегистрированы пользователи, которые их чаще всего используют. В результате заинтересованные пользователи получают локальный контроль над требуемыми ими данными и могут устанавливать или регулировать локальные ограничения на их использование. Администратор глобальной базы данных (АБД) отвечает за систему в целом. Как правило, часть этой ответственности делегируется на локальный уровень, благодаря чему АБД локального уровня получает возможность управлять локальной СУБД.

3) Повышение доступности данных.

В централизованных СУБД отказ центрального компьютера вызывает прекращение функционирования всей СУБД. Однако отказ одного из сайтов СУРБД или линии связи между сайтами сделает недоступными лишь некоторые сайты, тогда как вся система в целом сохранит свою работоспособность. Распределенные СУБД проектируются таким образом, чтобы обеспечивать продолжение функционирования системы, несмотря на подобные отказы. Если выходит из строя один из узлов, система сможет перенаправить запросы к отказавшему узлу в адрес другого сайта.

4) Повышение надежности.

Если организована репликация данных, в результате чего данные и их копии будут размещены более чем на одном сайте, отказ отдельного узла или соединительной связи между узлами не приведет к недоступности данных в системе.

5) Повышение производительности.

Если данные размещены на самом нагруженном сайте, который унаследовал от систем-предшественников высокий уровень параллельности обработки, то развертывание распределенной СУБД может способствовать повышению скорости доступа к базе данных (по сравнению с доступом к удаленной централизованной СУБД). Более того, поскольку каждый сайт работает только с частью базы данных, уровень использования центрального процессора и служб ввода/вывода может оказаться ниже, чем в случае централизованной СУБД.

Недостатки

1) Повышение сложности системы.

Распределенные СУБД, способные скрыть от конечных пользователей распределенную природу используемых ими данных и обеспечить необходимый уровень производительности, надежности и доступности, безусловно, являются более сложными программными комплексами, чем централизованные СУБД. Тот факт, что данные могут подвергаться копированию, также создает дополнительную предпосылку усложнения программного обеспечения распределенной СУБД.

2) Увеличение стоимости эксплуатации

Увеличение сложности означает и увеличение затрат на приобретение и сопровождение распределенной СУБД. Потребуется дополнительное оборудование, необходимого для установки сетевых соединений между узлами. Следует ожидать и увеличения расходов на оплату каналов связи и оплату труда персонала.

3) Проблемы защиты данных

В централизованных системах доступ к данным легко контролируется. Однако в распределенных системах потребуется организовать контроль доступа не только к копируемым данным, расположенных на нескольких производственных площадках, но и защиту самих сетевых соединений.

4) Усложнение контроля за целостностью данных

Реализация ограничений поддержки целостности обычно требует доступа к большому количеству данных, используемых при выполнении проверок, хотя и не требует выполнения операций обновления. В распределенных СУБД повышенная стоимость передачи и обработки данных может препятствовать организации эффективной защиты от нарушений целостности данных.

5) Усложнение разработки

Разработка распределенных баз данных, помимо обычных трудностей, связанных с процессом проектирования централизованных баз данных, требует принятия решения о фрагментации данных, распределении фрагментов по отдельным узлам и репликации данных.

Примеры Распределенных реляционных (SQL) баз данных

SQL базы данных могут быть развёрнуты в облаке как в виде единичных инстансов на виртуальных машинах. Ниже в таблице приведены SQL СУБД поддерживающие развёртывание в облачной инфраструктуре.

Cloud
• Amazon Relational Database Service (MySQL) • Microsoft SQL Azure (MS SQL) • GenieDB • Heroku PostgreSQL as a Service (распределенная и выделенная база данных) • Clustrix Database as a Service • Xeround Cloud Database — MySQL front-end • EnterpriseDB Postgres Plus Cloud Database • GaianDB • ClearDB ACID-compliant MySQL • VK Cloud Solutions — Базы данных: MySQL, PostgreSQL • Yandex Cloud PosgreSQL

Примеры Распределенных нереляционных (noSQL) баз данных

Название	Лицензия	Высокая доступность	Где используется
Apache Accumulo	AL2		
Aerospike	AGPL		
Apache Cassandra	AL2	да	
Apache Ignite	AL2		
Bigtable	Proprietary		Google

Название	Лицензия	Высокая доступность	Где используется
Couchbase	AL2		LinkedIn, PayPal, eBay
CrateDB	AL2	да	
Apache Druid	AL2		Netflix, Yahoo
Dynamo	Proprietary		Amazon
Hazelcast	AL2, Proprietary		
HBase	AL2	да	
Hypertable	GPL 2		Baidu
MongoDB	SSPL		Mail VK group
Riak	AL2	да	
Redis	BSD License	да	
Scylla	AGPL		
ClickHouse	AL2	да	Yandex
Greenplum	AL2	да	

Data Warehouse

Бизнес стал активно интересоваться корпоративными хранилищами еще в конце прошлого века. Их внедряли для увеличения скорости реагирования на изменения, мониторинга показателей эффективности и автоматизации процессов. Разные приложения отвечали за разные процессы: одни использовались для финансовых операций, другие — для координации цепочек поставок, третьи помогали анализировать показатели продаж.

Компаниям требовалось решение, которое бы позволило анализировать информационную картину целиком, а не данные из разных систем по отдельности.

Для решения этой проблемы был создан особый инструмент — корпоративное хранилище данных, или Data Warehouse. Фактически DWH — это предметно-ориентированная база данных, которая консолидирует важную бизнес-информацию и позволяет в автоматическом режиме подготавливать консолидированные отчеты.

Пример DWH

Data Warehouse — это единое корпоративное хранилище архивных данных из разных источников (систем, департаментов и прочее). Цель Data Warehouse — обеспечить пользователя (компанию и ее ключевых лиц) возможностью принимать верные решения в ключе управления бизнесом на основе целостной информационной картины.

Озеро данных - Data Lake

Data Lake (Озеро данных) – это метод хранения данных системой или репозиторием в натуральном (RAW) формате, который предполагает одновременное хранение данных в различных схемах и форматах. Обычно используется blob-объект (binary large object) или файл. Идея озера данных в том чтобы иметь логически определенное, единое хранилище всех данных в организации (enterprise data) начиная от сырых, необработанных исходных данных (RAW data) до предварительно обработанных (transformed) данных, которые используются для различных задач: отчеты, визуализация, аналитика и машинное обучение.

Data Lake включает структурированные данные из реляционных баз данных (строки и колонки), полуструктурированные данные (CSV, лог файлы, XML, JSON), неструктурированные данные (почтовые сообщения, документы, pdf) и даже бинарные данные (видео, аудио, графические файлы).

Data Lake кроме методов хранения и описания данных, предполагает определение источников и методов пополнения данных. При этом используются следующие термины:

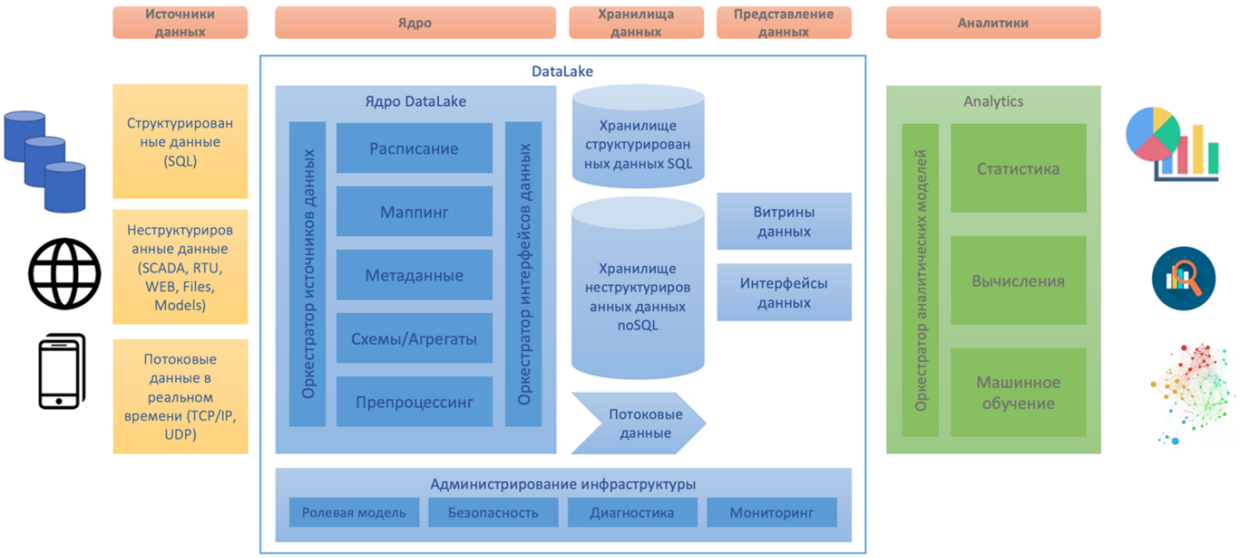
- источники – sources;
- настройки каналов – pipelines;
- регулярность обновлений – schedulers;
- владельцы – custodians;
- время хранения – retention time;
- метаданные – другие «данные о данных».

Data Lake может использовать единый репозиторий в качестве хранилища данных (HDFS, EDW, IMDG, Cloud и т.д.) либо использовать модульную концепцию источников хранения данных для разных требований по безопасности, скорости, доступности при соблюдении условий хранения данных: неизменяемые RAW данные, согласованное время хранения (retention time), доступность.

С практической точки зрения, можно охарактеризовать озеро данных тремя атрибутами:

1. Собирать и хранить все что угодно — озеро данных содержит все данные, как сырые необработанные данные за любой период времени, так и обработанных/очищенные данные.
2. Глубокий анализ — озеро данных позволяет пользователям исследовать и анализировать данные.
3. Гибкий доступ — озеро данных обеспечивает гибкий доступ для различных данных и различных сценариев.

Пример архитектуры Data Lake представлен на рисунке:



Архитектура Data Lake

Масштабирование баз данных

Масштабирование сервисов без сохранения состояния производится довольно просто. А что делать, если мы сохраняем данные в базе данных? Нам нужно знать, как выполнять масштабирование и в таком случае. Различные типы баз данных требуют разных форм масштабирования, и понимание того, какая из этих форм подойдет наилучшим образом именно для вашего случая, гарантирует выбор нужной технологии баз данных с самого начала.

Вертикальное и горизонтальное масштабирование

Вертикальное масштабирование

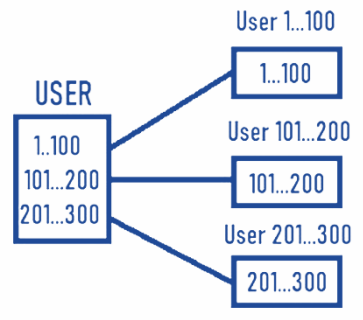
Вертикальное масштабирование — scaling up — увеличение количества доступных для ПО ресурсов за счет увеличения мощности применяемых с серверов.

Разделение данных в СУБД называют – фрагментирование, сегментирование, партиционирование или шардинг.

Фрагментирование данных при вертикальном масштабировании называется - «вертикальный шардинг» или просто «Партицирование» - это разбитие таблицы базы данных на несколько таблиц, по какому либо принципу.

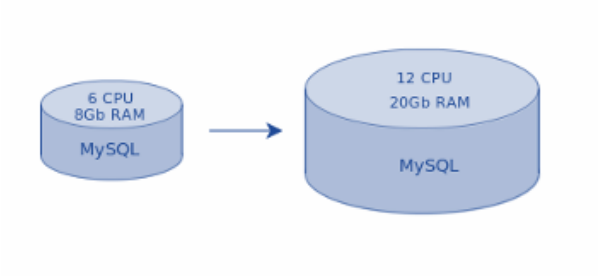
Партицирование происходит на одном инстансе базы данных, где находится большая таблица, которая разделяется на более мелкие части.

Например, у вас база данных пользователей, которую надо разбить (рисунок). Это дает: прирост в 3-4 раза, просто в исполнении и нужен один инстанс.



Возможности для масштабирования для серверов баз данных определяются применяемыми программными решениями. Чаще всего это реляционные базы данных (MySQL, Postgresql) или NoSQL (MongoDB, Cassandra и др).

Например MySQL является одной из популярных СУБД и, как и любая из них, требует для работы под нагрузкой много серверных ресурсов. Масштабирование возможно, в основном, вверх.

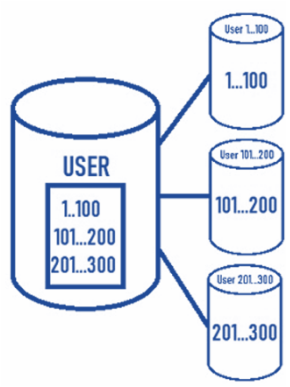


Таким образом с MySQL нужен будет сервер с большим количеством CPU и оперативной памяти, такие сервера имеют значительную стоимость.

Горизонтальное масштабирование

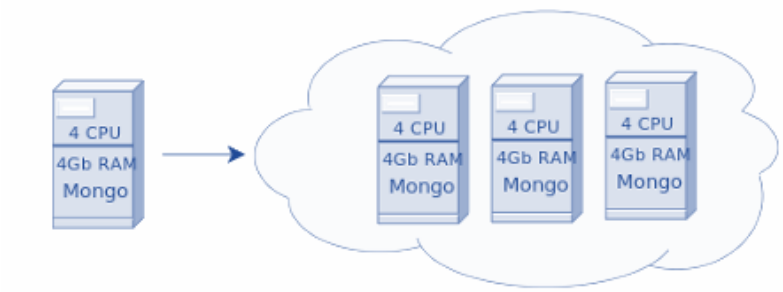
Горизонтальное масштабирование — scaling out — увеличение количества узлов, объединенных в кластер серверов при нехватке CPU, памяти или дискового пространства.

Фрагментирование данных при горизонтальном масштабировании называется - «горизонтальный шардинг» - таблицы лежат в разных базах на других инстансах (это виртуальная машина, которая запускается и работает в облаке). Что это дает: сложнее вертикального варианта масштабирования и используются разные сервера.



Главное отличие горизонтального масштабирования от вертикального в том, что горизонтальное масштабирование будет разносить данные по разным инстансам.

Например с MongoDB можно добавить еще один средний сервер и полученное решение будет стабильно работать давая дополнительно отказоустойчивость. NoSQL масштабируется легко. Например MongoDB будет значительно выгоднее материально масштабироваться горизонтально. При этом не потребует трудозатратных настроек и поддержки получившегося решения. Шардинг при этом осуществляется автоматически. Хотя для его настройки требуется вносить изменения в код сервиса и репликация, которая может быть сложной в поддержке.



Горизонтальное масштабирование для серверов баз данных при больших нагрузках значительно дешевле.

Приведенный пример с реляционными базами данных и NoSQL является ситуацией, которая имеет место чаще всего. Масштабируются также фронтэнд и бэкенд сервера.

Репликация (replication)

Сам термин пришел в информатику из биологии. Репликация предполагает механизм создания полной копии. Рассматриваемый процесс подразумевает

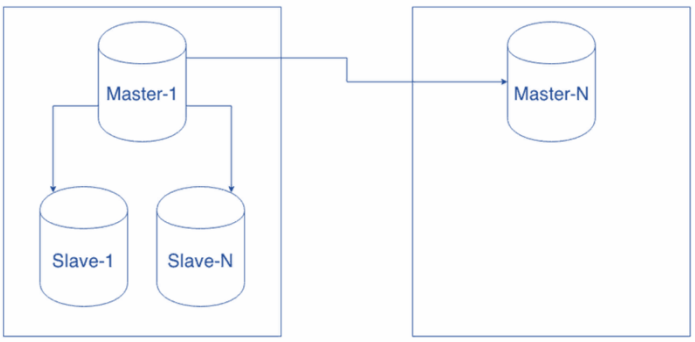
периодическую синхронизацию отдельного объекта со всеми копиями, которых, к слову, может быть неограниченное количество. Другими словами, изменения, сделанные в одной реплике, распространятся на все остальные.

Репликация (от лат. replico -повторяю) — это синхронное или асинхронное копирование данных между несколькими серверами с главного сервера БД на одном или нескольких зависимых серверах. Ведущие сервера называют мастерами (master), а ведомые сервера — слэйвами (slave). Мастера используются для изменения данных, а слэйвы — для считывания.

В классической схеме репликации обычно один мастер и несколько слэйвов, так как в большей части веб-проектов операций чтения на несколько порядков больше, чем операций записи. Однако в более сложной схеме репликации может быть и несколько мастеров.

Узлы, в которых хранятся копии БД, называются - репликами.

Репликация (replication) также означает хранение копий одних и тех же данных на нескольких машинах, соединенных с помощью сети.



Пример репликации БД Master

Главная задача — повышение стрессоустойчивости системы. В случае необходимости созданные реплики исходной системы могут стать базой для полного перезапуска как на старой физической платформе, так и за ее пределами. При этом наибольшую защищенность обеспечивает только тот вариант репликации, который подразумевает дублирование не только содержащейся в системе информации, но и структуры объекта.

Репликация представляет собой набор технологий копирования и распространения данных и объектов баз данных между базами данных, а также синхронизации баз данных для поддержания согласованности. Используя репликацию, можно распространять данные в различные расположения, а также

удаленным или мобильным пользователям по локальным или глобальным сетям через коммутируемое соединение, по беспроводным соединениям и через Интернет.

Репликация особенно востребована в распределенных системах, таких как микросервисные, при этом репликация очень хорошо сочетает с подходом Database-per-service.

Паттерн database-per-service предполагает, что у каждого сервиса своя база. Сервис может обращаться к данным другого сервиса только через API (в широком смысле), без прямого подключения к его базе.

Паттерн database-per-service позволяет командам соответствующих сервисов выбирать базы, как им нравится. Кто-то умеет в MongoDB, кто-то верит в PostgreSQL, кому-то достаточно Redis (риск потери данных при выключении для этого сервиса приемлем), а кто-то вообще хранит данные в CSV-файлах на диске (а почему бы, собственно, и нет?).

Стоит понимать, что сама по себе репликация данных не имеет ценности, и является лишь инструментом решения следующих задач:

- Повышение производительности чтения данных. С помощью репликации мы сможем поддерживать несколько копий сервера, и распределять между ними нагрузку.
- Повышение отказоустойчивости. Репликация позволяет избавиться от единственной точки отказа, которой является одиночный сервер БД. В случае аварии на основном сервере, есть возможность быстро переключить нагрузку на резервный.
- Распространение данных. В современную эпоху глобализации ваше приложение может обслуживать пользователей со всего мира, и мы хотим, чтобы жители и Сиднея, и Хельсинки имели минимальную задержку доступа к нему.
- Распределение нагрузки. В случае, если БД обслуживает запросы разных типов (быстрые и легкие, медленные и тяжелые), может иметь смысл развести эти запросы по разным серверам, для увеличения эффективности работы каждого типа.
- Тестирование новых конфигураций. С помощью репликации есть возможность проведения тестирования новых версий сервера БД, изменения параметров конфигурации, и даже изменения типов хранилища данных.

- Резервное копирование. С помощью репликации есть возможность делать механизмы резервного копирования более гибкими и вносить меньше негативных эффектов в работающую систему.
- Высокая доступность. Сохранение работоспособности системы в целом даже в случае отказа одной из машин (или нескольких машин, или даже целого ЦОДа) и повышения, таким образом, доступности.
- Работа в офлайн-режиме. Возможность продолжения работы приложения в случае прерывания соединения с сетью.
- Снижение задержек передачи. Данные размещаются географически близко к пользователям, чтобы те могли работать с ними быстрее (сокращения, таким образом, задержек);
- Масштабирование. Возможность обрабатывать бОльшие объемы операций чтения, чем способна обработать одна машина, с помощью выполнения операций чтения на репликах. Для горизонтального масштабирования количества машин, обслуживающих запросы на чтение (и повышения, таким образом, пропускной способности по чтению).

Если реплицируемые данные не меняются с течением времени, то репликация не представляет сложности: просто нужно однократно скопировать их на каждый узел и все.

Основные сложности репликации заключаются в том, что делать с изменениями реплицированных данных.

Репликации разделяют **два отдельных класса:**

- Однонаправленная. Дублирование основной БД в резервные реплики. Данные изменяются только в основной БД, а в другую БД данные только копируются и не подвергаются другим изменениям.
- Мультинаправленная. Синхронизация происходит между двумя или более самостоятельных копий одной БД. Данные могут изменяться и вводиться на всех БД.

Репликации бывают:

- Синхронные,
- Асинхронные,

- Полусинхронные.

Существует **три базовых алгоритма репликации** изменений между узлами:

- Репликацию с одним ведущим узлом master (single-leader),
- С несколькими ведущими узлами master-master (multi-leader),
- Без ведущего узла (master/leader-less).

Варианты реализации репликаций:

- Физическая. Передаётся информация о физическом изменении страниц базы данных.
- Логическая. Передаётся информация об измененни записей базы данных.
- Передача запросов. Передаётся информация о выполненных запросах.

Практически все распределенные базы данных используют один из этих подходов. У каждого из них есть свои плюсы и минусы.

Еще необходимо отвечать на вопрос - что делать с решением конфликтов и со сбойными репликами. Зачастую в базах данных есть для этого алгоритмы и конфигурационные настройки, и хотя конкретные детали зависят от базы, общие принципы одинаковы для множества различных реализаций.

Репликация с одним ведущими и ведомыми узлами (master-slave)

При наличии множества реплик неизбежно возникает вопрос: как обеспечить присутствие всех данных во всех репликах?

Каждая операция записи в базу должна учитываться каждой репликой, иначе нельзя гарантировать, что реплики содержат одни и те же данные. Наиболее распространенное решение - это репликация с ведущим узлом (leader-based replication). Известна также под названиями «репликации типа «активный/пассивный» (active/passive (primary) replication) или «репликации типа «главный-подчиненный» (master-slave replication). Показана на рисунке.

Репликация с ведущим узлом главный — подчиненный/master-slave

Схема работы:

1) Одна из реплик назначается ведущим (leader/master/primary) узлом. Клиенты, желающие записать данные в базу, должны отправить свои запросы ведущему узлу, который сначала записывает новые данные в свое локальное хранилище.

2) Другие реплики называются ведомыми (followers/ read replicas), подчиненные узлы/slaves), вспомогательные узлы/secondaries/горячий резерв (hot standbys) узлами. Всякий раз, когда ведущий узел записывает в свое хранилище новые данные, он также отправляет информацию об изменениях данных всем ведомым узлам в качестве части журнала репликации (replication log) или потока изменений (change stream). Все ведомые узлы получают журнал от ведущего и обновляют соответствующим образом свою локальную копию БД, применяя все операции записи в порядке их обработки ведущим узлом.

3) Когда клиенту требуется прочитать данные из базы, он может выполнить запрос или к ведущему узлу, или к любому из ведомых. Однако запросы на запись разрешено отправлять только ведущему (ведомые с точки зрения клиента предназначены только для чтения).

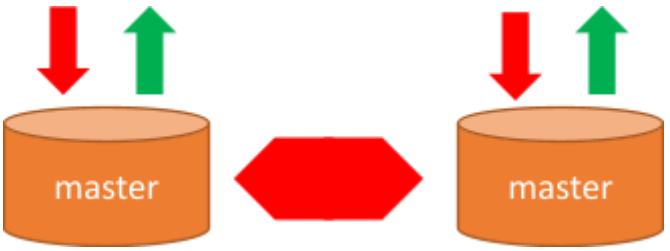
Преимущества	**Недостатки**
● Простота реализации	● Высокая нагрузка на только один ведущий узел, через который должны проходить все операции записи; ● Низкая надёжность при прерывании соединения к БД.

Репликация с несколькими ведущими узлами мульти-мастер (multi-master)

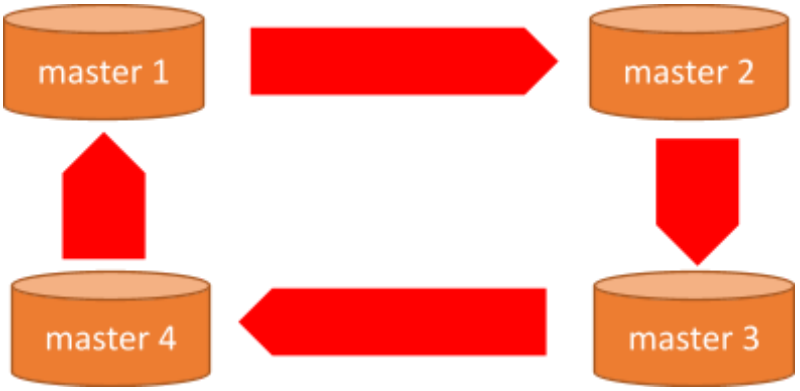
У репликации с ведущим узлом есть один большой недостаток: только один ведущий узел, через который должны проходить все операции записи. Если подключиться к узлу невозможно, например, из-за разрыва сети между вами и ведущим узлом, то нельзя выполнить и запись в базу данных.

Напрашивающееся расширение модели репликации с ведущим узлом — разрешение приема запросов на запись более чем одному узлу. Репликация будет выполняться так же, как и ранее: каждый узел, обрабатывающий операцию записи, должен перенаправлять информацию о данных изменениях всем остальным узлам. Это называется схемой репликации с несколькими ведущими узлами (multi-leader replication), или репликацией типа «главный — главный» (master — master), или репликацией типа «активный/активный» (active/active replication). При такой схеме каждый из ведущих узлов одновременно является ведомым для других ведущих.

Репликация Multi-Master – это круговая репликация, когда вы можете писать на любом сервере, а данные будут реплицироваться на все остальные.



Репликация multi-master



Общая схема репликация multi-master со множеством узлов

В режиме мульти-мастера любой из узлов кластера доступен для чтения или записи данных. Каждый ведущий узел скачала выполняет запрос у себя, затем синхронизирует его на других ведущих узлах.

Сравнение схем с одним и несколькими ведущими узлами:

- Производительность.
  a) При схеме с одним ведущим узлом каждая операция записи должна проходить через Интернет в ЦОД, содержащий ведущий узел. Это может существенно задержать операцию записи и вообще пойдет вразрез с идеей нескольких ЦОДов.

  b) При схеме с несколькими ведущими узлами все операции записи будут обрабатываться в локальных ЦОДах и реплицироваться асинхронно в остальные ЦОДы. Таким образом, сетевые задержки между ЦОДами становятся незаметными для пользователей, а значит, субъективная производительность возрастет.

- Устойчивость к перебоям в обслуживании ЦОДов.

  a) При схеме с одним ведущим узлом в случае отказа ЦОДа, в котором находится ведущий узел, восстановление после сбоя сделает ведомый узел в другом ЦОДе ведущим.

  b) При схеме с несколькими ведущими узлами каждый ЦОД может работать независимо от остальных и репликация наверстывает упущенное после возобновления работы отказавшего ЦОДа.

- Устойчивость к проблемам с сетью.

  a) Трафик между ЦОДами обычно проходит через общедоступный Интернет — менее надежный, чем локальная сеть внутри ЦОДа. Схема с одним ведущим узлом очень чувствительна к сбоям работы этого соединения между ЦОДами, поскольку операции записи выполняются через него синхронно.

  b) Схема асинхронной репликации с несколькими ведущими узлами обычно более устойчива к проблемам с сетью: временные сбои работы сети не мешают обработке операций записи.

У репликации с несколькими ведущими узлами есть свои преимущества. Однако она имеет и серьезный недостаток: одни и те же данные могут

одновременно модифицироваться в двух различных ЦОДах, и такие конфликты записи необходимо разрешать.

4)

Преимущества	**Недостатки**
● Производительность, ● Надёжность.	● Конфликты данных, ● Сложность реализации.

Шардирование

Архитектуры без разделения ресурсов

Архитектуры без разделения ресурсов (shared-nothing architectures), известные под названием горизонтального масштабирования (horizontal scaling, scaling out). При этом подходе каждый компьютер или виртуальная машина, на которой работает база данных, называется узлом (node). Все узлы используют свои CPU, память и диски независимо друг от друга. Согласование узлов выполняется на уровне программного обеспечения с помощью обычной сети.

Для систем без разделения ресурсов не требуется никакого специального аппаратного обеспечения, так что можно применять машины, имеющие наилучшее соотношение «цена/производительность». При желании можно распределить данные по многим географическим регионам и таким образом снизить задержку для пользователей и потенциально сделать систему устойчивой к потере целого ЦОДа. Благодаря облачному развертыванию виртуальных машин вашей компании не нужно работать в масштабах Google: даже маленькие компании могут позволить себе межрегиональную распределенную архитектуру.

В случае распределенных по нескольким узлам данных необходимо осознавать ограничения и компромиссы подобных распределенных систем — БД не может сама по себе избавить вас от этих нюансов.

Шардирование/Секционирование/Партиционирование - разбиение большой базы данных на небольшие подмножества, называемые секциями (partitions), в результате чего разным узлам можно поставить в соответствие различные секции/шарды (это называется «шардинг»). Шардирование

(partitioning) представляет собой способ умышленного разбиения большого набора данных на меньшие.

Путаница в терминологии

То, что называется в:

- MongoDB – секцией (partition);
- Elasticsearch и SolrCloud – «шард» (shard);
- HBase — «регион» (region);
- Bigtable — «сегмент» (tablet);
- Cassandra и Riak — «виртуальный узел» (vnode);
- Couchbase — «виртуальный участок» (vBucket).

Однако «секционирование» (partitioning) — наиболее часто употребляемый термин.

Шардинг — это прием, который позволяет распределять данные между разными физическими серверами. Процесс шардинга предполагает разнесения данных между отдельными шардами на основе некого ключа шардинга. Связанные одинаковым значением ключа шардинга сущности группируются в набор данных по заданному ключу, а этот набор хранится в пределах одного физического шарда. Это существенно облегчает обработку данных.

Цель шардирования заключается в распределении нагрузки по данным и запросам равномерно по нескольким машинам, а также в том, чтобы избежать горячих точек (узлов с непропорционально высокой нагрузкой). Это требует выбора подходящей для набора данных схемы шардирования и перебалансировки шард при добавлении или удалении узлов из кластера.

Шардированные БД появились в 1980-х годах. Это были такие программные продукты, как Teradata и Tandem NonStop SQL. Недавно технология была открыта заново базами данных NoSQL и складами данных на основе Hadoop. Одни из систем спроектированы в расчете на нагрузку в виде обработки транзакций, а другие предназначены для аналитики: это влияет на настройку системы, но основы шардирования не отличаются для обоих видов нагрузки.

Горизонтальный шардинг — это разделение одной таблицы на разные сервера. Это необходимо использовать для огромных таблиц, которые не умещаются на одном сервере. Разделение таблицы на шарды(секции) делается по такому алгоритму:

- На нескольких серверах создается одна и та же таблица (только структура, без данных).
- В приложении выбирается условие, по которому будет определяться нужное соединение (например, четные на один сервер, а нечетные — на другой).
- Перед каждым обращением к таблице происходит выбор нужного соединения.

Ключ шарда — какой-то параметр, который позволит определять, на каком именно сервере лежат те или иные данные:

- ID поля таблицы
- Хеш-таблица с соответствиями «пользователь=сервер». Тогда, при определении сервера, нужно будет выбрать сервер из этой таблицы. В этом случае узкое место — это большая таблица соответствия, которую нужно хранить в одном месте.
- Литерал – формирование имени сервера с помощью числового (буквенного) преобразования. Например, можно вычислять номер сервера, как остаток от деления на определенное число (количество серверов, между которыми Вы делите таблицу). В этом случае узкое место — это проблема добавления новых серверов — Вам придется делать перераспределение данных между новым количеством серверов.

Глава 7. Паттерны микросервисной архитектуры

Паттерны, относящиеся к сервисным архитектурам

1. Согласованность данных CAP теорема
2. Паттерны в архитектурах с брокерами сообщений в Event-driven architecture (EDA)
3. Паттерны автомасшабирования, балансировки и резервирования в Space-Based Architecture (SBA)
4. Паттерны реплицирования и шардирования данных
5. Паттерны проектирования, декомпозиции и рефакторинга для перехода на сервисы
6. Паттерны связывания и коммуникаций
7. Паттерны управления данными
8. Паттерны поддержки целостности данных и согласованности транзакций
9. Паттерны повышения отказоустойчивости
10. Паттерны построения пользовательского интерфейса
11. Паттерны обнаружения сервисов
12. Паттерны развёртывания сервисов
13. Паттерны мониторинга
14. Паттерны тестирования
15. Двенадцать факторов Web-приложения

Стили взаимодействия

Стили взаимодействия — это способы описания взаимодействия между клиентами и сервисами, не привязанные к конкретным технологиям.

Прежде чем выбирать механизм межпроцессного взаимодействия (inter-process communication - IPC) для API сервиса, полезно будет подумать о стиле взаимодействия между сервисом и его клиентами. Это поможет сосредоточиться на требованиях и не увязнуть в деталях конкретной технологии

IPC. К тому же выбор стиля взаимодействия влияет на уровень доступности приложения.

Существует много разных стилей взаимодействия между клиентом и сервисом. Как показано в таблице, их можно разделить на два уровня:

Таблица

Взаимодействие	Один к одному	Один ко многим
Синхронное	Запрос/ответ	-
Асинхронное	Асинхронный запрос/ответ, однонаправленные уведомления	Издатель/подписчик, издатель/асинхронные ответы

1) Первый уровень определяет выбор между отношениями «один к одному» и «один ко многим»:
   - *«один к одному»* — каждый клиентский запрос обрабатывается ровно одним сервисом;
   - *«один ко многим»* — каждый запрос обрабатывается несколькими сервисами.
2) Второй уровень определяет выбор между синхронным и асинхронным взаимодействием:
   - *синхронное* — клиент рассчитывает на своевременный ответ от сервиса и может даже заблокироваться на время ожидания;
   - *асинхронное* — клиент не блокируется, а ответ, если таковой придет, может быть отправлен не сразу.

Виды взаимодействия **«один к одному»:**

- *Запрос/ответ* — клиент отправляет сервису запрос и ждет ответа. Он рассчитывает на то, что ответ придет своевременно, и может даже заблокироваться на время ожидания. Этот стиль взаимодействия обычно приводит к жесткой связанности сервисов.
- *Асинхронный запрос/ответ* — клиент отправляет запрос, а сервис отвечает асинхронно. Клиент не блокируется на время ожидания, поскольку сервис может долго не отвечать.

- *Однонаправленные уведомления* — клиент отправляет сервису запрос, не ожидая (и не получая) ничего в ответ.

Виды взаимодействия **«один ко многим»**:

- *Издатель/подписчик* — клиент публикует сообщение с уведомлением, которое потребляется любым количеством заинтересованных сервисов.
- *Издатель/асинхронные ответы* — клиент публикует сообщение с запросом и ждет определенное время ответа от заинтересованных сервисов.

Сервисы обычно используют сочетание этих стилей взаимодействия.

CAP-теорема

Любую система можно описать при помощи CAP-теоремы (рисунок 8) – это краеугольный камень проектирования распределённых систем.

Звучит она следующим образом. Существует три базиса создания распределённых систем. Это целостность, доступность и устойчивость к разделению (к сетевым проблемам).

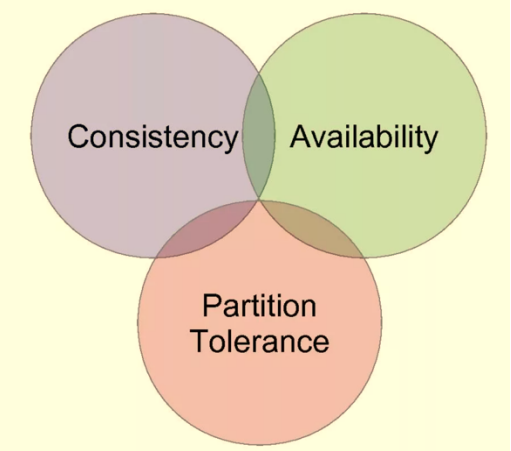


Схема CAP

Согласованность

Согласованность означает, что все клиенты обладают доступом к одним и тем же данным в одно и то же время вне зависимости от того, к какому узлу они подключены. Чтобы это стало возможным, каждый раз при записи данных на одном узле необходимо обеспечить их мгновенную пересылку или репликацию между всеми остальными узлами в системе, после чего запись будет считаться успешной.

Доступность

Доступность означает, что обработка всех запросов данных клиентов гарантируется даже в том случае, если отдельные узлы недоступны. Другое определение: все активные узлы в распределенной системе возвращают допустимый ответ на любой запрос без каких-либо исключений.

Устойчивость к разделению

Разделение — это нарушение связи в пределах распределенной системы — потерянное или временно приостановленное соединение между двумя узлами. Устойчивость к разделению означает, что кластер должен продолжать работать, несмотря на любое число нарушений связи между узлами системы.

Паттерны обнаружения сервисов в микросервисной архитектуре

Эта группа шаблонов описывает методы, которые могут использовать клиентские приложения для определения местонахождения нужных им сервисов. Это особенно важно в микросервисных приложениях, так как они работают в виртуализированных и контейнерных средах, где количество экземпляров сервисов и их расположение изменяются динамически.

Ключевым компонентом обнаружения микросервисов выступает реестр сервисов (Service Registry) — база данных с информацией о расположении сервисных экземпляров. Когда экземпляры запускаются и останавливаются, информация в реестре обновляется. Но взаимодействовать с реестром сервисов можно двумя путями, которые и легли в основу описанных ниже шаблонов.

Обнаружение сервисов на уровне приложения

Один из способов реализации обнаружения сервисов заключается в том, что сервисы приложения и их клиенты взаимодействуют с реестром сервисов. Для вызова сервиса клиент сначала обращается к реестру, чтобы получить список его экземпляров, а затем шлет запрос одному из них.

Клиенты обращаются к реестру, чтобы найти сетевое местоположение доступных экземпляров сервиса.

Данный подход к обнаружению сервисов сочетает в себе два шаблона:

1) Первый — это саморегистрация (Self registration). Экземпляр сервиса обращается к API реестра, чтобы зарегистрировать свое сетевое местоположение. Он также может предоставить URL-адрес для проверки работоспособности. Этот URL-адрес является конечной точкой API, которую реестр периодически запрашивает, чтобы убедиться в том, что

сервис работает в нормальном режиме и доступен для обработки запросов. Реестр может требовать, чтобы экземпляр сервиса периодически вызывал API «пульс», который предотвращает истечение срока действия его регистрации.

2) Вторым шаблоном - является обнаружение на клиентской стороне (Client-side discovery). Когда клиент хочет обратиться к сервису, он обращается к реестру, чтобы получить список его экземпляров. Для улучшения производительности клиент может кэшировать экземпляры сервиса. После этого он использует алгоритм балансирования нагрузки, циклический или случайный, чтобы выбрать конкретный экземпляр и отправить ему запрос.

«Обнаружение сервисов на стороне клиента» (Client-Side Service Discovery)

В этом случае сервисы и их клиенты напрямую взаимодействуют с реестром. Последовательность шагов следующая:

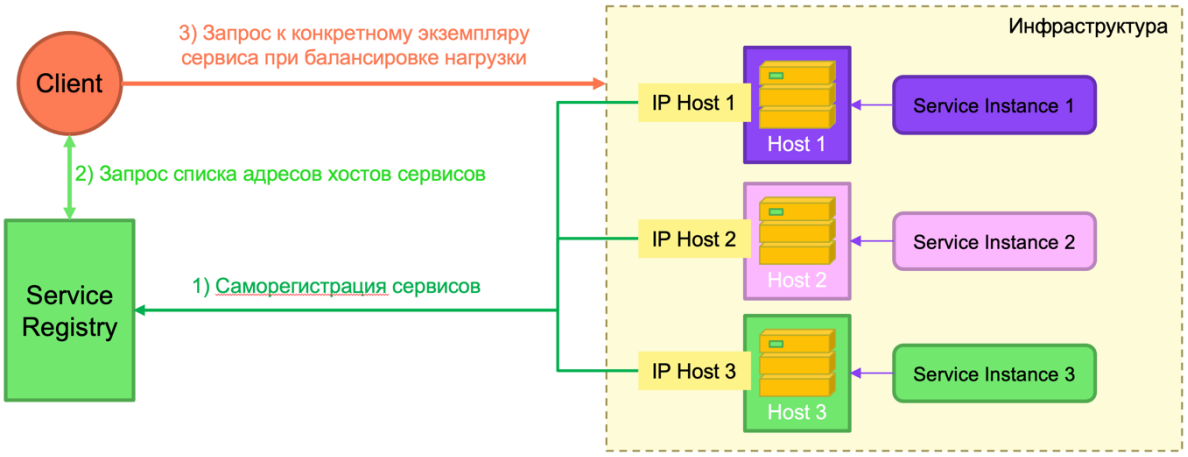
1. Экземпляр сервиса обращается к API реестра, чтобы зарегистрировать свое сетевое местоположение. Он также может предоставить URL-адрес для проверки своей работоспособности (Health Check), который будет использоваться для продления срока его регистрации в реестре.

2. Клиент самостоятельно обращается к реестру сервисов, чтобы получить список экземпляров сервисов. Для улучшения производительности клиент может кэшировать экземпляры сервиса.

3. Клиент использует алгоритм балансировки нагрузки, циклический или случайный, чтобы выбрать конкретный экземпляр сервиса и отправить ему запрос.

Ключевое преимущество обнаружения сервисов на стороне клиента — его независимость от используемой платформы развертывания. Например, если часть ваших сервисов развернута на K8s, а остальные работают в устаревшей среде, то обнаружение на уровне приложения будет лучшим вариантом, так как серверное решение на базе Kubernetes не будет совместимо со всеми сервисами.

К недостаткам подхода можно отнести необходимость использования различных клиентских библиотек для каждого языка программирования, а иногда и фреймворка. Кроме этого, на вашу команду ложится дополнительная нагрузка по настройке и обслуживанию реестра сервисов.



Паттерн Client-Side Service Discovery

Обнаружение сервисов, предоставляемое платформой

Многие современные платформы развертывания, такие как Docker и Kubernetes, имеют встроенные реестр и механизм обнаружения сервисов. Платформа развертывания выдает каждому сервису DNS-имя, виртуальный IP-адрес (VIP) и привязанное к нему доменное имя. Клиент делает запрос к DNS-имени/VIP, а платформа развертывания автоматически направляет его к одному из доступных экземпляров сервиса. В итоге регистрация и обнаружение сервисов, а также маршрутизация запросов выполняются самой платформой.

Экземпляры сервисов заносятся в реестр регистратором. У каждого сервиса есть сетевое местоположение, DNS-имя или виртуальный IP-адрес. Клиент выполняет запрос к сетевому местоположению сервиса. Маршрутизатор обращается к реестру и распределяет запросы между всеми доступными сервисами

«Обнаружение сервисов на стороне сервера» (Server-Side Service Discovery)

В этом случае за регистрацию, обнаружение сервисов и маршрутизацию запросов отвечает инфраструктура развертывания. Последовательность шагов следующая:

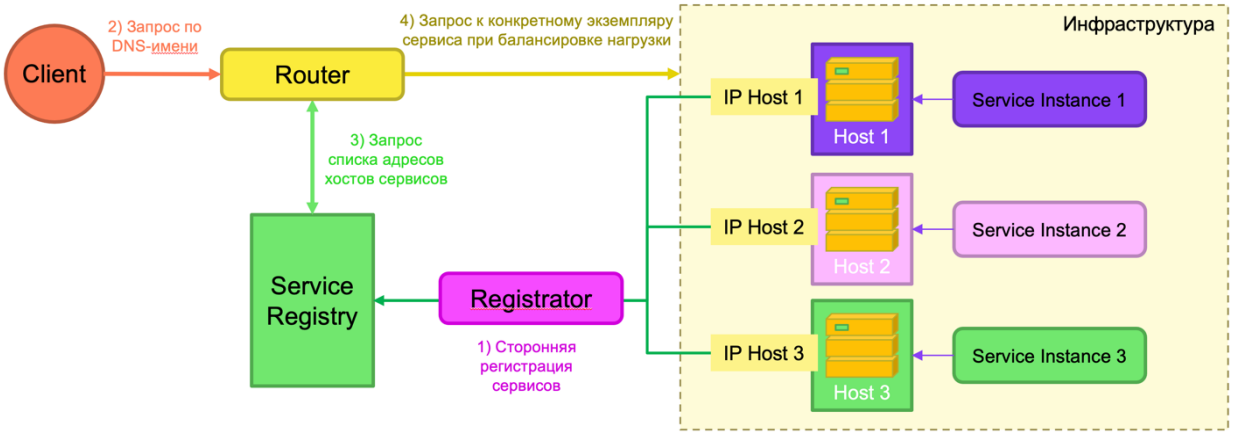
1. Регистратор, который обычно является частью платформы развертывания, прописывает все экземпляры сервисов в реестре сервисов. По каждому экземпляру сохраняется DNS-имя и виртуальный IP-адрес (VIP).

2. Вместо того чтобы обращаться к реестру напрямую, клиент делает запрос по DNS-имени сервиса. Запрос поступает в маршрутизатор, являющийся частью платформы развертывания.
3. Маршрутизатор обращается к реестру сервисов для получения сетевого расположения экземпляров нужного сервиса.
4. Маршрутизатор применяет балансировку нагрузки, чтобы выбрать конкретный экземпляр сервиса и отправить ему запрос.

Все современные платформы развертывания, включая Docker, Kubernetes и другие, как правило, имеют встроенный реестр и механизмы обнаружения сервисов.

Основное преимущество паттерна состоит в том, что всеми аспектами обнаружения сервисов занимается сама платформа. Дополнительный код на стороне клиента или сервисов не требуется. Благодаря этому достигается независимость от используемых в приложении языков программирования и фреймворков.

Недостатком паттерна является невозможность его применения к сервисам, которые развернуты вне основной платформы, реализующей механизмы обнаружения. Несмотря на это ограничение, рекомендуется использовать обнаружение сервисов на стороне сервера всюду, где это осуществимо.



Паттерн Server-Side Service Discovery

Архитектура развёртывания

Архитектуры развёртывания существенно разнятся, но в целом, ярусы начинаются с develpment (DEV) и заканчиваются production (PROD). Распространённой 4-х ярусной архитектурой является каскад ярусов deployment, testing, model, production (DEV, TEST, MODL, PROD) с деплоем софта на каждом ярусе по очереди.

Другое распространённое окружение это Quality Control (QC), для приёмочного тестирования; песочница или экспериментальное окружение (EXP), для экспериментов не предназначенных для передачи в продакшен; и Disaster Recovery ('аварийное восстановление'), для предоставления возможности немедленного отката в случае проблемы с продакшеном. Другой распространённой архитектурой является: deployment, testing, acceptance and production (DTAP).

Такая разбивка в частности подходит для серверных программ, когда сервера работают в удаленных дата-центрах; для кода который работает на конечных устройствах пользователя, например приложений (apps) или клиентов, последний ярус обозначают как окружение пользователя (USER) или локальное окружение (LOCAL).

Диаграмма развертывания

Диаграмма развертывания обеспечивает визуализацию элементов и компонентов программы, существующих лишь на этапе ее исполнения (runtime). В них представляются только компоненты-экземпляры программы, являющиеся исполняемыми файлами или динамическими библиотеками. Те компоненты, которые не используются на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. Она является единой для системы в целом, поскольку должна отражать особенности ее реализации. Разработка диаграммы развертывания, как правило, является последним этапом спецификации модели программной системы.

При разработке диаграммы развертывания преследуют следующие **цели**:

- определить распределение компонентов системы по ее физическим узлам;

- показать физические связи между всеми узлами реализации системы на этапе ее исполнения;
- выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Эти диаграммы разрабатываются совместно системными аналитиками, сетевыми инженерами, DevOps инженерами и системотехниками.

Элементами диаграмм являются:

a) Узлы

b) Связи между узлами

c) Компоненты приложения

Узел (node) представляет собой некоторый физически существующий элемент системы, обладающий определенным вычислительным ресурсом. В качестве такого ресурса может рассматриваться электронная, магнитооптическая память или процессор. Узлы бывают **двух типов**:

1) Устройство (*device* - физическое оборудование: компьютер или устройство, связанное с системой).

2) Среда выполнения (*execution environment* - программное обеспечение, которое само может включать другое программное обеспечение, например операционную систему или процесс-контейнер).

Узлы могут содержать артефакты (*artifacts*), которые являются физическим олицетворением программного обеспечения. Обычно это файлы.

Графически на диаграмме развертывания узел изображается в форме *трехмерного куба*. Узел имеет собственное имя, которое указывается внутри его графического символа. Сами узлы могут представляться как в качестве объектов (типов), так и в качестве экземпляров.

В первом случае имя узла записывается без подчеркивания и начинается с заглавной буквы. Во втором - имя узла-экземпляра записывается в виде

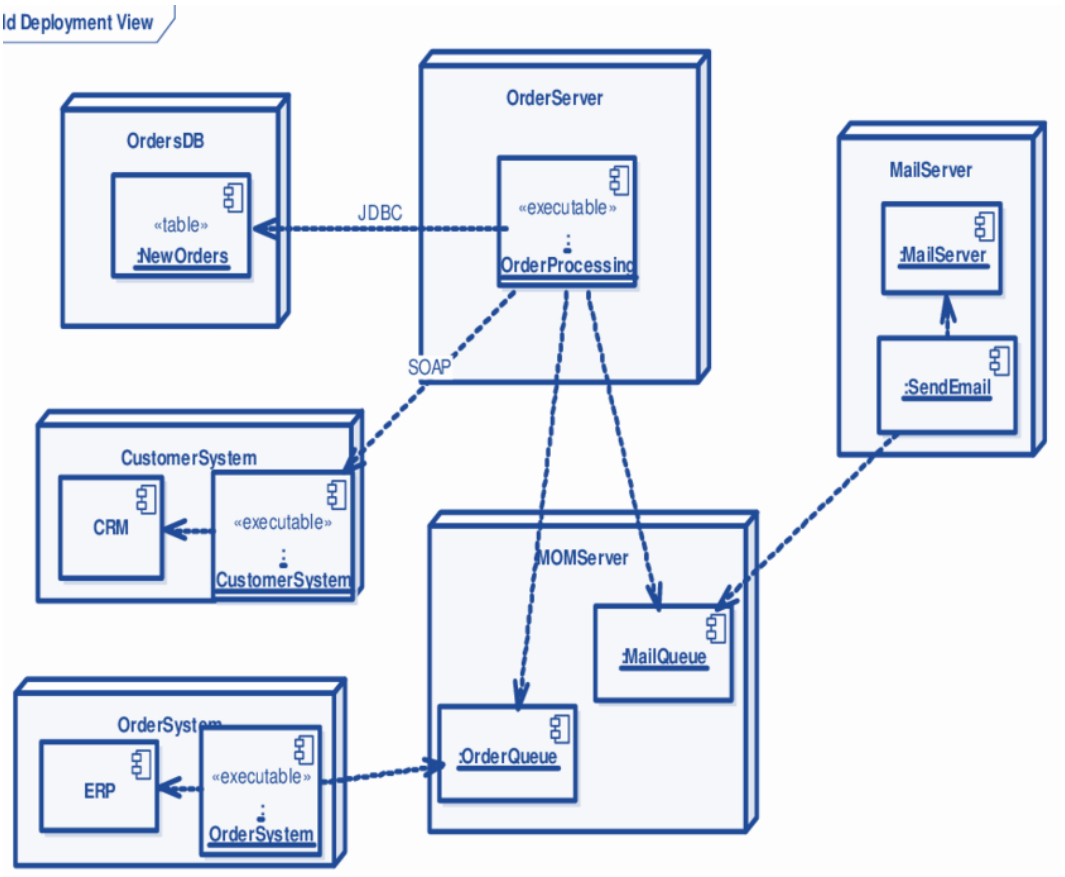
<имя узла ':' имя типа узла>.

Имя типа узла указывает на некоторую разновидность узлов, присутствующих в модели системы.

Так же, как и на диаграмме компонентов, изображения узлов могут расширяться, чтобы включить некоторую дополнительную информацию нем. Если эта информация относится к имени узла, то она записывается под ним.

Если необходимо явно указать компоненты, которые размещаются на отдельном узле, то это можно сделать двумя способами. Первый позволяет разделить графический символ узла на две секции горизонтальной линией. В верхней секции записывают имя узла, а в нижней размещенные на этом узле компоненты. Второй способ разрешает показывать на диаграмме развертывания узлы с вложенными изображениями компонентов. При этом нужно учитывать, что в качестве вложенных могут выступать только исполняемые компоненты.

Пример Диаграммы развёртывания представлен на рисунке.



Пример Диаграммы развёртывания

Использование контейнеров для развёртывания сервисов

Контейнеризация – это форма виртуализации операционной системы, при которой приложения запускаются в изолированных пользовательских пространствах, называемых контейнерами, и все они используют одну и ту же общую операционную систему (ОС).

По сути контейнер представляет собой полностью упакованную и портативную вычислительную среду:

Все, что необходимо приложению для запуска – библиотеки, конфигурационные файлы и зависимости – инкапсулируется и изолируется в контейнере.

Сам контейнер абстрагируется от хостовой ОС, имея лишь ограниченный доступ к базовым ресурсам – подобно легковесной виртуальной машине.

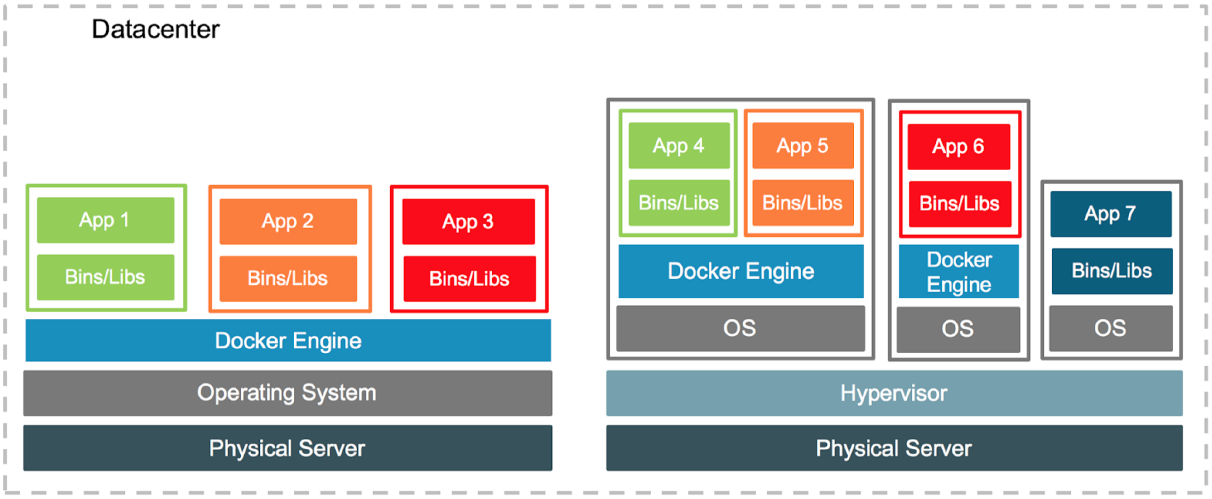
В результате контейнеризованное приложение может быть запущено на различных типах инфраструктуры: на пустых машинах, внутри ВМ, в облаке, без необходимости рефакторинга для каждой среды. При запуске не нужно настраивать отдельную гостевую ОС для каждого приложения, так как все они используют ядро одной операционной системы. Благодаря такой высокой эффективности, контейнеризация обычно используется для упаковки множества отдельных микросервисов, из которых состоят современные приложения.

Как работает контейнеризация

Каждый контейнер представляет собой исполняемый пакет программного обеспечения, работающий поверх операционной системы хоста. Хост может одновременно поддерживать сотни и даже тысячи контейнеров. Все контейнеры запускают минимальные, изолированные от ресурсов процессы, к которым другие не имеют доступ.

Контейнеризованное приложение состоит из нескольких слоев (см. рисунок ниже):

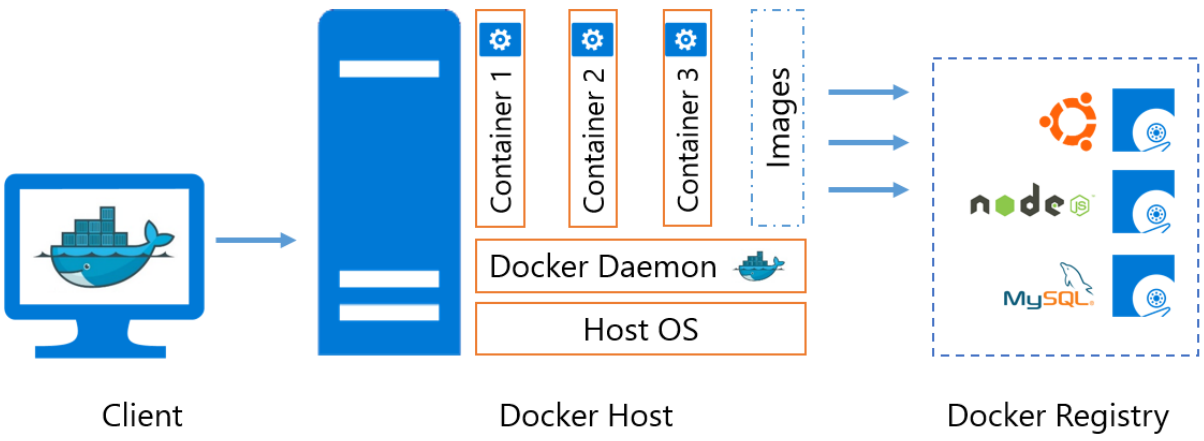
- В нижней части находится аппаратное обеспечение инфраструктуры: процессор, дисковое хранилище и сетевые интерфейсы.
- Уровнем выше расположены операционная система хоста и ее ядро. Последнее выступает «мостом» между программным обеспечением операционной системы и аппаратным обеспечением базовой системы.
- Движок контейнера и его гостевая ОС, характерные для используемой технологии контейнеризации, располагаются над операционной системой хоста.
- В самом верху находятся файлы и библиотеки (bins/libs) для каждого приложения и самих приложений, работающие в своих изолированных пользовательских пространствах (контейнерах).
- И далее - Приложения.



Варианты виртуализации и контейнерезации приложений

Докер — это платформа для разработки, доставки и эксплуатации приложений. Docker нужен для более быстрого выкладывания ваших приложений. С помощью docker можно отделить приложение от инфраструктуры и обращаться с инфраструктурой как управляемым приложением. Docker помогает выкладывать код быстрее, быстрее тестировать, быстрее выкладывать приложения и уменьшить время между написанием кода и запуска кода. Это делается с помощью легковесной платформы контейнерной виртуализации, используя процессы и утилиты, которые помогают управлять и выкладывать приложения.

Общая архитектура контейнеризации в Docker показана на рисунке.



Архитектура контейнеризации в Docker

В своем ядре docker позволяет запускать практически любое приложение, безопасно изолированное в контейнере. Безопасная изоляция позволяет вам запускать на одном хосте много контейнеров одновременно. Легковесная природа контейнера, который запускается без дополнительной нагрузки гипервизора, позволяет вам добиваться больше от вашего железа.

Платформа и средства контейнерной виртуализации могут быть полезны **в следующих случаях**:

- упаковывание вашего приложения (и так же используемых компонент) в docker контейнеры;
- раздача и доставка этих контейнеров вашим командам для разработки и тестирования;
- выкладывания этих контейнеров на ваши продакшены, как в дата центры так и в облака.

Docker состоит из **двух главных компонент:**

- Docker: платформа виртуализации с открытым кодом;
- Docker Hub: платформа-как-сервис для распространения и управления docker контейнерами.

Кроме Docker, на практике, часто можно увидеть **такие решения как**:

- Openshift
- Kubernetes
- Rancher
- Различные cloud решения (часто базируются на Kubernetes, например, Mail Cloud)

Контейнеризация и виртуализация – в чем разница

Контейнеризация происходит на уровне ОС, при этом все контейнеры делят между собой одно ядро. В отличие от виртуализации с участием виртуальных машин (ВМ): ВМ запускается поверх гипервизора, который представляет собой специализированное оборудование, программное

обеспечение или прошивку для работы ВМ на хостовой машине, например, сервере или ноутбуке.

С помощью гипервизора каждой ВМ назначаются не только необходимые bins/libs, но и виртуализированный аппаратный стек, включающий в себя процессоры, хранилища и сетевые адаптеры.

Для выполнения всех этих задач каждая ВМ полагается на полноценную гостевую ОС. Сам гипервизор может быть запущен из машинной ОС хоста или как bare-metal приложение.

Как и контейнеризация, традиционная виртуализация позволяет полностью изолировать приложения, чтобы они работали независимо друг от друга, используя реальные ресурсы из базовой инфраструктуры.

Но **разница** между ними значительна:

- Серьезные накладные расходы, так как все ВМ нуждаются в собственных гостевых ОС и виртуализированных ядрах, плюс потребность в дополнительном уровне (гипервизоре) между ними и хостом.
- Гипервизор также может вызвать дополнительные проблемы с производительностью, особенно когда он работает под управлением ОС хоста, например, на Ubuntu.

Из-за высоких общих накладных расходов на ресурсы, хост-машина, которая могла бы работать с десятью или более контейнерами, с трудом справляется с поддержкой одной ВМ.

Тем не менее, запуск нескольких ВМ на относительно мощном оборудовании все еще является распространенной практикой при разработке и развертывании приложений.

Цифровые рабочие среды обычно поддерживают как виртуализацию, так и контейнеризацию для достижения общей цели – сделать приложения как можно более доступными и масштабируемыми.

Преимущества контейнеризации

Контейнерные приложения легко доставляются пользователям в цифровой среде. В частности, контейнеризация приложения на основе микросервисов, имеет больше преимуществ, начиная от гибкости при разработке ПО и заканчивая более удобным контролем расходов.

- Гибкая, ориентированная на DevOps разработка программного обеспечения

По сравнению с ВМ, контейнеры проще настраивать, независимо от того, использует команда UNIX-подобную ОС или Windows. Инструменты для разработчиков универсальны и просты в использовании, что позволяет быстро разрабатывать, упаковывать и внедрять контейнеризированные приложения в различных операционных системах. Инженеры и команды DevOps могут применять технологии контейнеризации для ускорения рабочих процессов.

- Сокращение накладных расходов и экономия затрат по сравнению с ВМ

Контейнер не требует полноценной гостевой операционной системы или гипервизора. Это позволяет не только сократить время загрузки, но и уменьшить объем памяти и, как правило, повысить производительность. Также сокращаются расходы на сервер и лицензирование, которые в противном случае были бы направлены на поддержку развертывания нескольких ВМ. Таким образом, контейнеры позволяют повысить эффективность и рентабельность сервера.

- Портативность во всех рабочих пространствах

Каждый контейнер абстрагируется от хостовой операционной системы и будет работать одинаково в любом месте. Соответственно, его можно записать для одной хостовой среды, а затем портировать и развернуть в другой, при условии, что новый хост поддерживает соответствующие контейнерные технологии и операционные системы. Контейнеры для Linux составляют большую часть всех развернутых контейнеров и могут быть портированы на различные ОС на базе Linux, независимо от того, находятся ли они on-premise или в облаке. На Windows контейнеры для Linux можно легко запускать в виртуальной машине Linux VM или через изоляцию Hyper-V.

- Простое управление с помощью оркестровки

Оркестровка контейнеров с помощью такого решения, как Kubernetes, делает управление контейнерными приложениями и сервисами более практичным. Используя Kubernetes, можно автоматизировать развертывание и откат, организовать систему хранения данных, выполнить балансировку нагрузки и перезагрузить вышедшие из строя контейнеры. Kubernetes совместим со многими контейнерными движками, в том числе Docker и OCI.

Контейнеризация, микросервисы и оркестровка

Микросервисы, входящие в состав приложения, могут быть упакованы в контейнеры и управляться на масштабируемой облачной инфраструктуре. Основные преимущества контейнеризации микросервисов включают минимальные накладные расходы, независимое масштабирование и простое управление с помощью Kubernetes.

Kubernetes (K8s) – платформа контейнерной оркестрации с открытым исходным кодом, разработанная Google. Предназначена для развертывания, масштабирования и управления контейнерными приложениями.

И хотя это не единственное решение, Kubernetes стала общепринятым стандартом контейнерной оркестрации.

Функционал демонстрирует, на что должен быть способен современный инструмент оркестрации:

- экспонировать контейнеры по имени DNS или IP-адресу.
- управлять балансировкой нагрузки и распределением трафика для контейнеров.
- автоматически монтировать локальные и облачные хранилища.
- выделять определенные ресурсы процессора и оперативной памяти для контейнеров, а затем помещать их на узлы.
- заменять или уничтожать проблемные контейнеры без ущерба для производительности приложений.

Kubernetes стал популярен благодаря пробам работоспособности (health probe). На практике это означает, что вы деплоите контейнер в pod, а Kubernetes следит за работоспособностью процесса. Обычно такая модель недостаточно хороша. Процесс может выполняться, но при этом не быть работоспособным.

Здесь пригодятся проверки readiness и liveness. Kubernetes выполняет readiness-проверку, чтобы решить, когда приложение будет готово принимать трафик во время запуска. Liveness-проба непрерывно проверяет работоспособность сервиса. До Kubernetes это было не очень популярно, зато сегодня почти все языки, фреймворки и рантаймы проводят проверки работоспособности для быстрого запуска конечного устройства.

Также Kubernetes предложил управление жизненным циклом приложения, то есть нам больше не нужно самим контролировать запуск и остановку сервиса. Kubernetes запускает приложение, останавливает его, переносит по разным нодам. Чтобы вся эта система работала, нужно правильно настроить события при запуске и остановке.

Еще одна причина популярности Kubernetes — декларативные развёртывания. Нам больше не нужно запускать сервисы и проверять логи, чтобы узнать, запущены ли они. Нам не нужно апгрейдить инстансы вручную. За все отвечает декларативное развёртывание в Kubernetes. В зависимости от

выбранной стратегии оно останавливает старые инстансы и запускает новые. А если что-то пошло не так, само делает откат.

Еще одно преимущество — объявление требований для ресурсов. При создании сервиса мы упаковываем его в контейнер и говорим платформе, сколько ЦП и памяти ему понадобится. На основе этой информации Kubernetes подбирает самую подходящую ноду для ваших рабочих нагрузок. Раньше приходилось вручную размещать инстанс на ноде, опираясь на наши критерии. Теперь мы сообщаем Kubernetes наши предпочтения, и он все решает за нас.

В Kubernetes можно управлять конфигурациями на разных языках. Нам не нужно проверять конфигурацию в рантайме приложения — Kubernetes проследит, чтобы настройки оказались на той же ноде, что и рабочая нагрузка. Настройки для приложения подключаются как файлы в томе или переменные окружения.

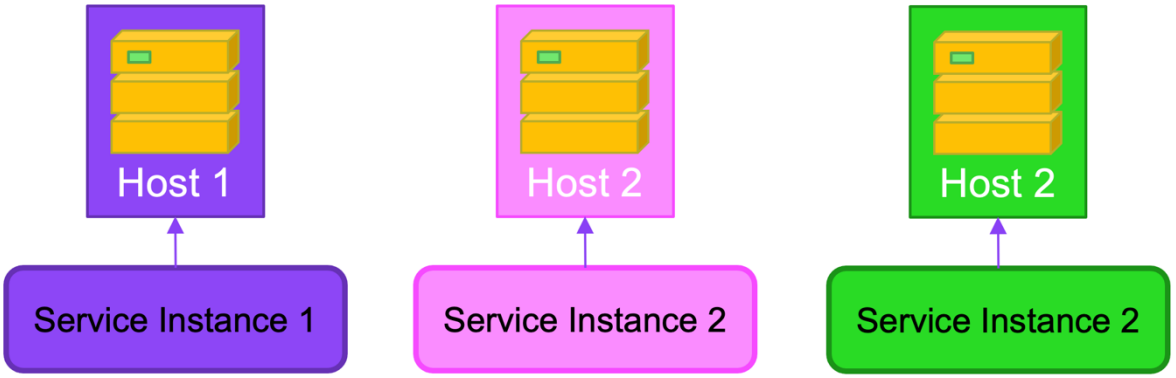
Паттерны развертывания микросервисов

Рассмотрим возможные варианты развертывания разработанных микросервисов.

«Экземпляр сервиса на хост» (Service Instance Per Host)

При переходе на микросервисную архитектуру рекомендуется проводить развертывание каждого экземпляра сервиса на собственном хосте (виртуальном или физическом). Паттерн Service Instance Per Host позволяет изолировать экземпляры сервисов друга от друга, избежать конфликтов версий и требований к ресурсам, максимально использовать ресурсы хоста, а также легче и быстрее проводить повторные развертывания. К недостаткам паттерна можно отнести потенциально менее эффективное использование ресурсов по сравнению с развертыванием нескольких экземпляров на хост.

Иногда выделяют разновидности описанного шаблона, наиболее часто используемые на практике: «Экземпляр сервиса на виртуальную машину» (Service Instance Per VM) и «Экземпляр сервиса на контейнер» (Service Instance Per Container). При их использовании каждый экземпляр сервиса упаковывается и разворачивается в виде отдельной виртуальной машины либо контейнера соответственно.

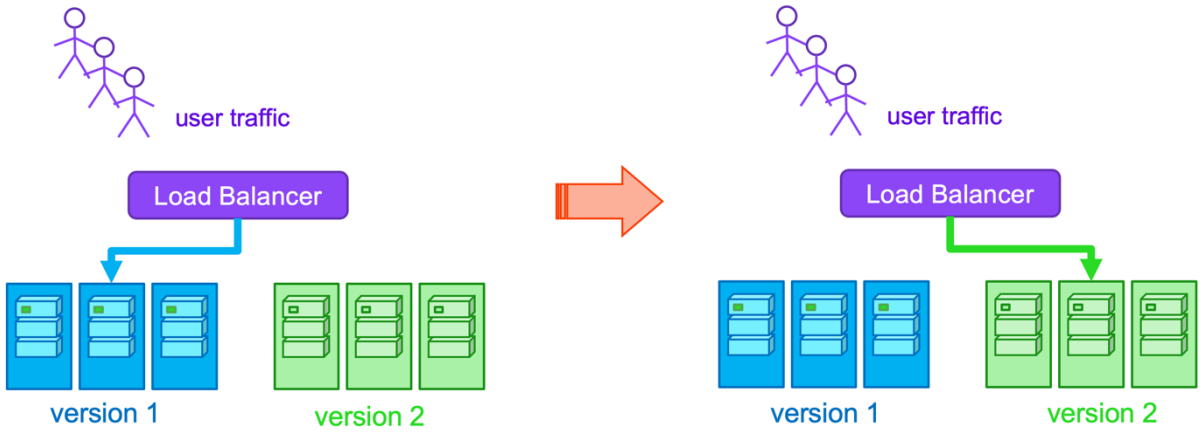


Паттерн Service Instance Per Host

«Сине-зеленое развертывание» (Blue-Green Deployment)

Паттерн позволяет выполнить развертывание новых версий сервисов максимально незаметно для пользователей, сократив время простоя до минимума. Это достигается за счет запуска двух идентичных производственных сред — условно синего и зеленого цвета. Предположим, что синий — это существующий активный экземпляр, а зеленый — это новая версия приложения, развернутая параллельно с ним.

В любой момент времени только одна из сред является активной, и именно она обслуживает весь производственный трафик. После успешного развертывания новой версии — с прохождением всех тестов и так далее — трафик переключается на нее. В случае ошибок всегда можно вернуться к предыдущей версии.

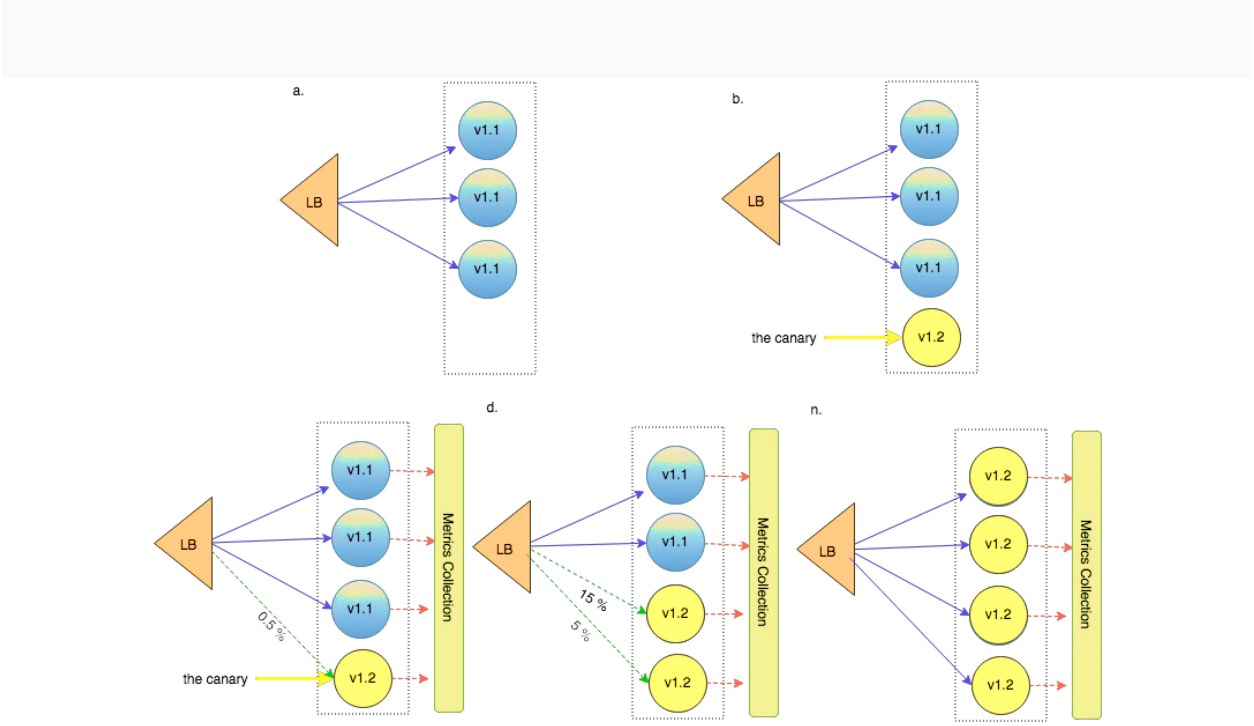


Паттерн Blue-Green Deployment

«Канареечные развертывания» (Canary Deployment)

Канареечное развертывание позволяет проверять потенциальные проблемы и соответствие ключевым показателям, прежде чем влиять на всю продакшн систему/базу пользователей. Мы «тестируем в продакшене», выполняя развертывание непосредственно в продакшн окружении, но только для небольшой группы пользователей.

Вы можете выбрать маршрутизацию на основе процента использования или в зависимости от региона/местоположения пользователя, типа клиента и параметров биллинга. Даже при развертывании в небольшом подмножестве важно тщательно отслеживать производительность приложения и измерять ошибки — эти показатели определяют порог качества. Если приложение ведет себя так, как ожидалось, мы начинаем переводить больше трафика на экземпляры новой версии.



Балансировщик нагрузки (LB) постепенно выпускает новую версию в продакшн

Преимущества:

- Наблюдаемость;

- Возможность тестирования на продакшен трафике. Протестировать в среде разработки так же, как в производственной, сложно;
- Возможность задеплоить версию для небольшой группы пользователей и получить реальную обратную связь перед более крупным деплоем;
- Быстро получить отказ. Поскольку мы развертываем прямо в продакшен, мы можем быстро получить отказ работы, то есть немедленно вернуться на рабочую версию, если что-то сломается. При этом сбой затронет только часть пользователей, а не все сообщество.

Что такое observability

Под observability (наблюдаемость) подразумевается показатель того, насколько эффективно можно определить внутреннее состояние системы по ее выходным данным (телеметрии). Ничего нового в этом нет — мы пытаемся разобраться в работе приложений по логам с начала времен, но сейчас все осложняется облаками, контейнерами, микросервисами, многоязычными средами и т. д.

В ИТ и облачных вычислениях наблюдаемость - это возможность измерения текущего состояния системы на основе генерируемых ею данных, таких как журналы, метрики и трассировки.

Наблюдаемость опирается на телеметрию, полученную с помощью приборов, которые поступают от конечных точек и служб в ваших многооблачных вычислительных средах. В этих современных средах каждый компонент аппаратной, программной и облачной инфраструктуры, а также каждый контейнер, инструмент с открытым исходным кодом и микросервис генерируют записи о каждом действии. Цель наблюдаемости - понять, что происходит во всех этих средах и между технологиями, чтобы вы могли обнаружить и устранить проблемы для поддержания эффективности и надежности ваших систем и удовлетворенности ваших клиентов.

Организации обычно реализуют наблюдаемость, используя комбинацию методов инструментария, включая инструменты инструментария с открытым исходным кодом, такие как OpenTelemetry или Prometheus.

Поскольку облачные сервисы опираются на уникальную распределенную и динамичную архитектуру, наблюдаемость также иногда может относиться к конкретным программным инструментам и практикам, которые используются компаниями для интерпретации данных о производительности облака. Хотя некоторые люди могут думать о наблюдаемости как о слове для сложного мониторинга производительности приложений (APM), есть несколько ключевых

различий, которые следует иметь в виду при сравнении наблюдаемости и мониторинга.

В чем разница между мониторингом и наблюдаемостью?

Является ли наблюдаемость на самом деле мониторингом под другим названием? Если коротко, то нет. Хотя наблюдаемость и мониторинг связаны и могут дополнять друг друга, на самом деле это разные понятия.

В сценарии мониторинга вы обычно предварительно настраиваете приборные панели, которые предназначены для предупреждения вас о проблемах производительности, которые вы ожидаете увидеть позже. Однако эти панели основаны на ключевом предположении, что вы можете предсказать, с какими проблемами вы столкнетесь до их возникновения.

Облачные среды не очень хорошо подходят для такого типа мониторинга, потому что они динамичны и сложны, что означает, что у вас нет возможности знать заранее, какие проблемы могут возникнуть.

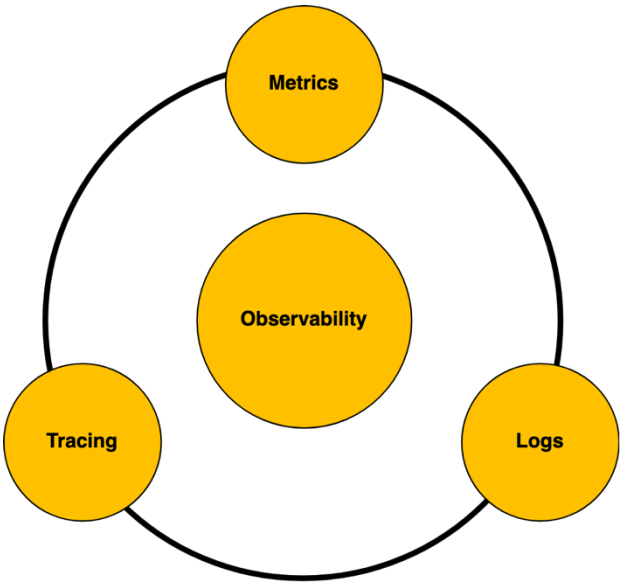
В сценарии наблюдаемости, когда среда полностью инструментирована для предоставления полных данных наблюдаемости, вы можете гибко исследовать происходящее и быстро найти первопричину проблем, которые вы, возможно, не могли предвидеть.

Столпы observability

Observability - это инструментарий систем для сбора полезных данных, которые подскажут вам, когда и почему системы ведут себя определенным образом. При возникновении неисправности в системе необходимо сначала изучить данные телеметрии, чтобы лучше понять причину ошибки.

Следующие данные телеметрии, которые принято называть столпами observability, являются ключевыми для достижения наблюдаемости в распределенных системах:

- Логи
- Метрики
- Трассировки



Основы observability

Логи

Логи - это структурированные и неструктурированные строки текста, которые система выдает при выполнении определенных кодов. В целом, логи можно рассматривать как запись события, произошедшего в приложении. Логи помогают выявить непредсказуемое и эмерджентное поведение компонентов архитектуры микросервисов.

Их легко генерировать и обрабатывать. Большинство прикладных фреймворков, библиотек и языков поддерживают ведение логов. Практически каждый компонент распределенной системы генерирует логи действий и событий в любой момент времени.

Файлы логов содержат исчерпывающую информацию о системе, например, о неисправности и конкретном времени, когда она произошла. Анализируя логи, вы можете устранить неполадки в коде и определить, где и почему произошла ошибка. Логи также полезны для устранения инцидентов безопасности в балансировщиках нагрузки, кэшах и базах данных.

Метрики

Метрики - это числовое представление данных, которые можно использовать для определения общего поведения службы или компонента с течением времени. Метрики включают в себя набор атрибутов (например, имя, значение, метка и метка времени), которые передают информацию о качестве службы или компонента.

В отличие от логов событий, в котором регистрируются конкретные события, метрики - это измеренное значение, полученное в результате работы системы. Метрики действительно экономят время, поскольку их можно легко соотнести между компонентами инфраструктуры, чтобы получить целостное представление о состоянии и производительности системы. Кроме того, они упрощают составление запросов и позволяют дольше хранить данные.

Например, можно собирать показатели времени работы системы, времени отклика, количества запросов в секунду, а также количества вычислительной мощности или памяти, используемой приложением. Обычно SRE и операционные инженеры используют метрики для запуска оповещений, когда значение системы превышает определенный порог.

Например, допустим, вы хотите отслеживать количество запросов в секунду в службе HTTP. Вы заметили резкий всплеск трафика и хотите знать, что происходит в вашей системе. Метрики обеспечивают более глубокую видимость и понимание, которые помогают понять причину всплеска. Всплеск может быть вызван неправильной конфигурацией службы, вредоносным поведением или проблемами в других частях вашей системы. Помимо обеспечения видимости, вы также можете использовать информацию для обнаружения и определения серьезности проблем.

Трассировки

Хотя логи и метрики могут быть адекватными для понимания поведения и производительности отдельных систем, они редко предоставляют полезную информацию для понимания времени жизни запроса в распределенной системе. Чтобы просмотреть и понять весь жизненный цикл запроса или действия в нескольких системах, вам понадобится другая техника наблюдаемости, называемая трассировкой.

Трассировка представляет собой весь путь запроса или действия, проходящий через все узлы распределенной системы. Трассировка позволяет составлять профиль и наблюдать за системами, особенно за контейнерными приложениями, бессерверными архитектурами или архитектурой микросервисов. Анализируя данные трассировки, вы и ваша команда можете измерить общее состояние системы, определить узкие места, быстрее выявлять и устранять проблемы, а также определить приоритетные области для оптимизации и улучшений.

Трассировки являются важным компонентом наблюдаемости, поскольку они обеспечивают контекст для других компонентов наблюдаемости. Например, вы можете проанализировать трассировку, чтобы определить наиболее ценные

метрики, основанные на том, чего вы пытаетесь достичь, или журналы, относящиеся к проблеме, которую вы пытаетесь устранить.

Трассировка лучше подходит для отладки и мониторинга сложных приложений, которые нетривиально борются за ресурсы (например, мьютекс, диск или сеть).

Трассировка дает быстрые ответы на следующие вопросы в распределенных программных средах:

- В каких службах есть неэффективный или проблемный код, который должен быть приоритетным для оптимизации?
- Как обстоят дела со здоровьем и производительностью сервисов, составляющих распределенную архитектуру?
- Каковы узкие места в производительности, которые могут повлиять на общий опыт конечного пользователя?

Хотя логи, трассировки и метрики служат каждая своей уникальной цели, все они работают вместе, чтобы помочь лучше понять производительность и поведение распределенных систем.

Глава 8. Двенадцать факторов приложения

I. Кодовая база

Одна кодовая база, отслеживаемая в системе контроля версий, – множество развёртываний

II. Зависимости

Явно объявляйте и изолируйте зависимости

III. Конфигурация

Сохраняйте конфигурацию в среде выполнения

IV. Сторонние службы (Backing Services)

Считайте сторонние службы (backing services) подключаемыми ресурсами

V. Сборка, релиз, выполнение

Строго разделяйте стадии сборки и выполнения

VI. Процессы

Запускайте приложение как один или несколько процессов не сохраняющих внутреннее состояние (stateless)

VII. Привязка портов (Port binding)

Экспортируйте сервисы через привязку портов

VIII. Параллелизм

Масштабируйте приложение с помощью процессов

IX. Утилизируемость (Disposability)

Максимизируйте надёжность с помощью быстрого запуска и корректного завершения работы

X. Паритет разработки/работы приложения

Держите окружения разработки, промежуточного развёртывания (staging) и рабочего развёртывания (production) максимально похожими

XI. Журналирование (Logs)

Рассматривайте журнал как поток событий

XII. Задачи администрирования

Выполняйте задачи администрирования/управления с помощью разовых процессов

Глоссарий

Архитектура программного обеспечения — совокупность важнейших решений об организации программной системы. Понятия «архитектура» и «дизайн» программного обеспечения зачастую тождественны.

Распределенные вычислительные системы — это физические компьютерные, а также программные системы, реализующие тем или иным способом параллельную обработку данных на многих вычислительных узлах.

Сервис-ориентированная архитектура (*COA*, англ. *service-oriented architecture-SOA*) — модульный подход к разработке программного обеспечения, базирующийся на обеспечении удаленного по стандартизированным протоколам использования распределенных, слабо связанных легко заменяемых компонентов (сервисов) со стандартизированными интерфейсами.

Персентиль (или перцентиль или процентиль) — методика измерения в статистике, которая показывает процент значений измеряемой метрики, который находится ниже значения персентиля. Например, если говорить о времени ответа системы, 99й персентиль на отметке 100 миллисекунд говорит о том, что 99% измеряемых запросов выполнились за 100 миллисекунд и менее.

Согласованность в конечном счёте (англ. *eventual consistency*) — одна из моделей согласованности, используемая в распределённых системах для достижения высокой доступности, в рамках которой гарантируется, что в отсутствии изменений данных, через какой-то промежуток времени после последнего обновления («в конечном счёте») все запросы будут возвращать последнее обновлённое значение.

Конечный автомат — математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных.

Дополнительные материалы

1. Статья "Service Oriented Architecture (SOA)"
2. Статья "Architectural Patterns and Styles"
3. Статья "Understanding Service-Oriented Architecture"
4. The Clean Architecture // URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (дата обращения: 06.06.2021).
5. Мартин, Р. Чистый код: создание, анализ и рефакторинг. Москва : Юпитер, 2018. – 464 с.
6. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения. – Москва : Юпитер, 2018. – 352 с.
7. Фаулер М. Архитектура корпоративных приложений.: Пер с англ. – М.: Издательский дом «Вильямс», 2006. – 544 с.

Используемые источники

Книги к прочтению

1. Thomas Erl “Service-Oriented Architecture. Analysis and Design for Services and Microservices”. 2nd edition (December 12, 2016), - 416 pages.
2. Mark Richards, Neal Ford "Fundamentals of Software Architecture. An Engineering Approach". 1st Edition (March 3, 2020), - 419 pages.
3. Chris Richardson “Microservices Patterns”. 1st edition (November 19, 2018), - 520 pages.
4. Иван Портянкин «Программирование Cloud Native. Микросервисы, Docker и Kubernetes» Архитектура микросервисов. Leanpub book 2022, - 172 с.

Онлайн источники

5. Перечень интеграционных паттернов Enterprise Integration Patterns - Messaging Patterns Overview
6. Tarantool wiki - Tarantool — Википедия
7. Полный технический обзор приложения NET StockTrader 6.1 Sample Application
8. SOA
9. Документация на Tarantool - Tarantool – Documentation
10. CAP теорема - Что такое CAP-теорема, как она связана с Big Data и NoSQL-СУБД
11. Приложение .NET StockTrader 6.1 Sample Application
12. Архитектура микросервисов – A pattern language for microservices
13. Создание предметно ориентированных сервисов
14. Взаимодействие между микросервисами - Взаимодействие в архитектуре микрослужб | Microsoft Learn
15. Асинхронное взаимодействие на базе сообщений - Асинхронное взаимодействие на основе сообщений | Microsoft Learn
16. Docker Containers and VMs Together - Docker

17. Сравнение шаблона шлюза API с прямым взаимодействием клиента и микрослужбы - Сравнение шаблона шлюза API с прямым взаимодействием клиента и микрослужбы | Microsoft Learn
18. Слои, Луковицы, Гексогоны, Порты и Адаптеры — всё это об одном - Слои, Луковицы, Гексогоны, Порты и Адаптеры — всё это об одном / Хабр
19. OData
20. Webhook
21. Декомпозиция на микросервисы - 26 основных паттернов микросервисной разработки
22. Эволюция распределённых систем в Kubernetes. Эволюция распределённых систем в Kubernetes / Хабр
23. Архитектура микросервисов – A pattern language for microservices
24. Заблуждения о распределенных вычислениях - Fallacies of distributed computing - Wikipedia
25. Проблемы Long Pooling - RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP
26. Kafka - Apache Kafka: основы технологии / Хабр
27. CQRS - Шаблон CQRS - Azure Architecture Center | Microsoft Learn
28. Event Driven Messaging - Event-Driven Messaging - SOA Patterns
29. Схема маршрутизации шлюза - Схема маршрутизации шлюза - Azure Architecture Center | Microsoft Learn
30. Схема агрегирования на шлюзе - Схема агрегирования на шлюзе - Azure Architecture Center | Microsoft Learn
31. Схема разгрузки шлюза - Схема разгрузки шлюза - Azure Architecture Center | Microsoft Learn
32. Сага - Оркестрируемая сага или как построить бизнес-транзакции в сервисах с паттерном database per service / Хабр
33. От монолита к микросервисам - Переход от монолита к микросервисам: история и практика
34. Проблемы распределенных транзакций - Проблематика распределенных транзакций в контексте микросервисной архитектуры / Хабр

35. Пример перехода от монолита к микросервисам
36. Взаимодействие в архитектуре микрослужб - Взаимодействие в архитектуре микрослужб | Microsoft Learn
37. API Gateway - Сравнение шаблона шлюза API с прямым взаимодействием клиента и микрослужбы | Microsoft Learn
38. Микросервисы и монолиты - Как построить микросервисную инфраструктуру
39. Крис Ричардсон «Микросервисы. Паттерны разработки и рефакторинга».
40. Интеграционные парттерны, Pub-Sub - Enterprise Integration Patterns - Publish-Subscribe Channel
41. ESB - Централизованная шина vs Service Mesh: как митап превратить в баттл / Хабр
42. ESB - Путешествие в мир сервисных корпоративных шин на IBM WebSphere ESB
43. Kafka - Apache Kafka: обзор / Хабр
44. Асинхронное взаимодействие - Асинхронное взаимодействие на основе сообщений | Microsoft Learn
45. Сага - Паттерн: Сага / Хабр
46. Паттерны разработки микросервисов - 26 основных паттернов микросервисной разработки
47. EDA
48. Brokerless
49. EDA A Guide to Enterprise Event-Driven Architecture
50. MSA Pattern: Microservice Architecture
51. DLQ Что такое Dead Letter Queue (DLQ)? | Yandex Cloud - Документация
52. Patterns broker Модели обмена сообщениями - Cloud Design Patterns | Microsoft Learn
53. Взаимодействие Интеграция: синхронное, асинхронное и реактивное взаимодействие, консистентность и транзакции / Хабр

54. Надёжность MSA Микросервисы: проблемы, которые мы не замечаем / Хабр
55. AsyncAPI 2.2.0 | AsyncAPI Initiative for event-driven APIs
56. Идемподеннтность Идемпотентный метод - Глоссарий | MDN
57. Что такое Идемпотентность?
58. Отказоустойчивость Отказоустойчивое взаимодействие с внешними сервисами
59. Типы репликаций Типы репликации - SQL Server | Microsoft Learn
60. Современные распределенные хранилища данных: обзор технологий и перспективы
61. РСБД
62. Распределенное хранение данных: от облака до блокчейна | ForkLog
63. SDS Программно-определяемые СХД: сравниваем 7 решений / Хабр
64. Миграции Версионная миграция структуры базы данных: основные подходы
65. Данильчик В. В. Основные подходы к миграции схем баз данных
66. Ceph Знакомство с хранилищем Ceph в картинках
67. Транзакции Обработка распределенных транзакций в микросервисной архитектуре
68. MapReduce MapReduce
69. Паттерны обмена данными
70. Hadoop Apache Hadoop
71. BlockChain
72. Блокчейн против базы данных
73. Backup Виды резервного копирования
74. Snapshot 3 Snapshot Concepts & Architecture
75. GTID 17.1.3.1 GTID Format and Storage
76. Параллельная репликация

77. Индексирование БД 15) Индексирование в базах данных - CoderLessons.com

78. Хеширование 12) Хеширование в СУБД - CoderLessons.com

79. Хэш-алгоритмы / Хабр

80. Координатор Очень большой Postgres / Хабр

81. 2PC 3PC Понимание: двухфазная фиксация транзакции и трехфазная фиксация (2PC, 3PC) - Русские Блоги

82. Стратегии согласования Distributed transaction patterns for microservices compared

83. PostgreSQL

84. Кластер PostgreSQL внутри Kubernetes: что нужно знать для успешного внедрения

85. DBA 2. «Администрирование PostgreSQL 9.5. Расширенный курс». Потоковая репликация. Тема №11

86. Zab

87. Cosul Consensus Protocol | Raft | Consul | HashiCorp Developer

88. Пакетная обработка Пакетная обработка - Azure Architecture Center | Microsoft Learn

89. Streaming Обработка в режиме реального времени - Azure Architecture Center | Microsoft Learn