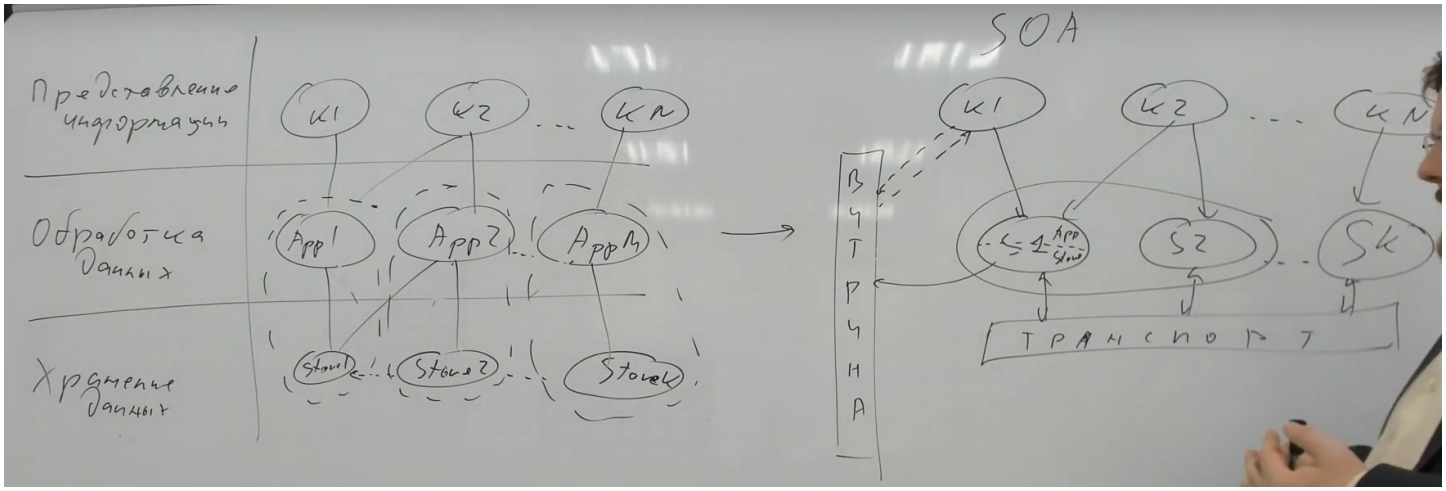
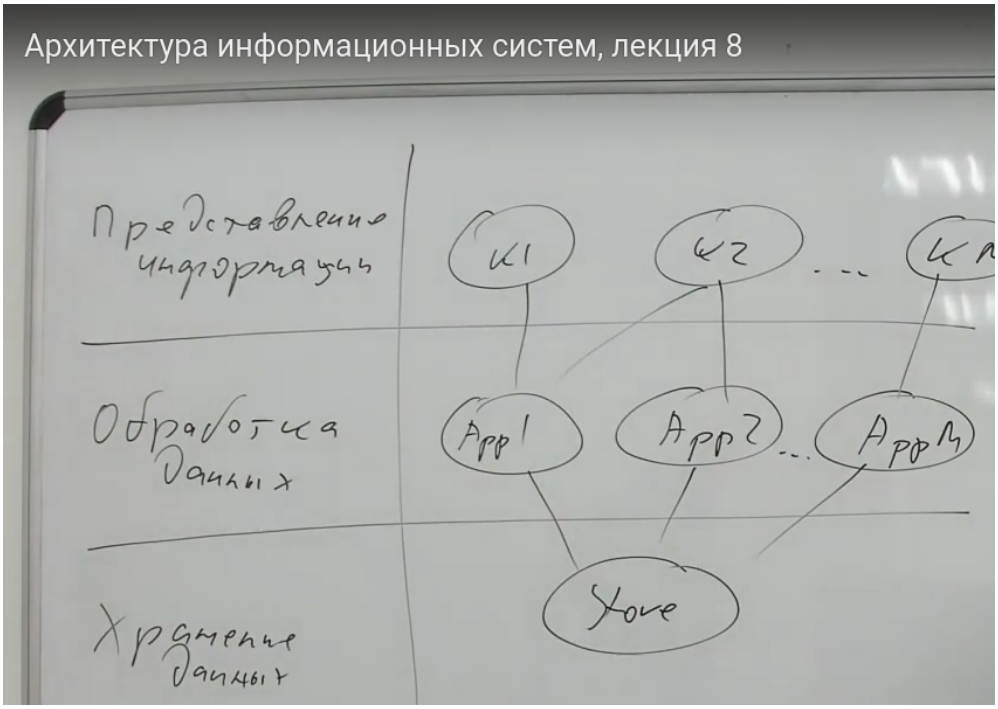
Перейти к описанию и содержанию конспекта.

Ссылка на лекцию





В классическом клиент-серверном подходе выделяется 3 слоя:

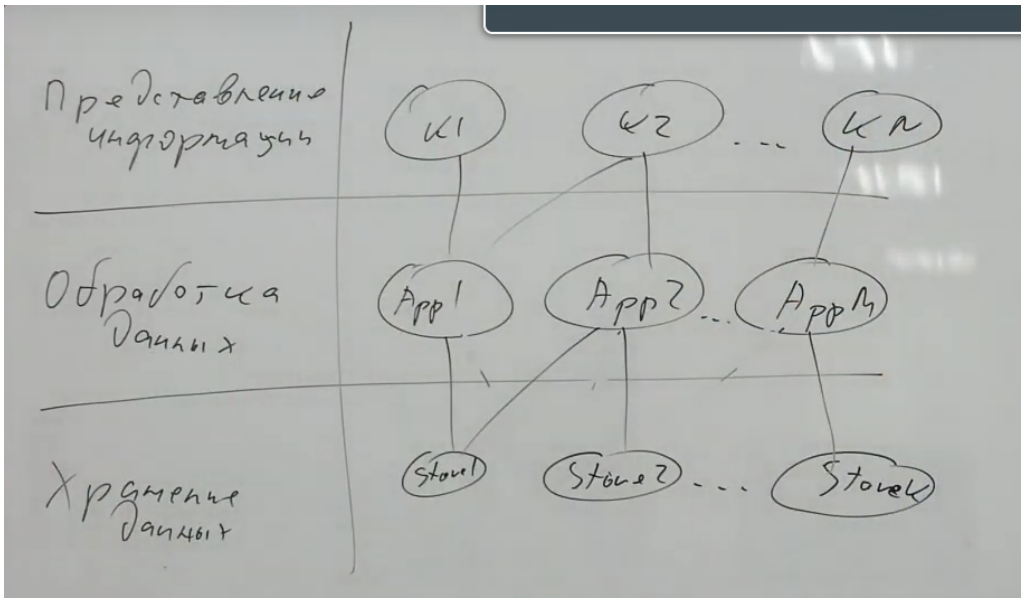
— представление инфы (множество клиентов K.n)

-обработка инфы (множество обработчиков App.m)

-хранение данных (множество хранилищ Storage)

Мы можем искать удачное решение по распределенной обработки,но узким местом остается единое хранилище. Если это хранилище будет работать транзационно, т. е. оно будет блокировать какие-то объекты данных на периоды выполнения каких-то запросов, то в конечном итоге мы будем ограничены в кол-ве приложений на уровне обр.данных и ограничены в интенсивности запросов от них производительностью хранилища(аппаратные средства в какой-то момент дают естественные ограничения).

Распределенные слой хранения



Когда возникает распределенный слой хранения данных, то появляется вопрос целостности данных.

Направления рассуждений по целостности данных:

—*сколько копий данных*? (одна или больше) Если одна копия, то повышение производительности возможно за счет одного фактора — приближать потребителей к хранимым данным (задачи приблизить данные к потребителю данных).

Если есть объекты данных относительно не связаны между собой, т. е. вероятность того, что вызов одного объекта попадет в транзакцию другого объекта. Например, таблицы которые работают в единой транзакцией и подпадают под совокупные блокировки или есть таблицы, которые не связаны в рамках одной транзакцией. Тогда становится логичным разместить объекты по экземплярам хранилища и считать, что они существуют в единственном экземпляре.

Все наши фантазии часто разбиваются о CAP-теорему

—*Что делать? Перемещение объектов данных или распределенные транзакции?*

Перемещение подразумевает, что транзация будет происходит в пределах одного узла. Если нужен объект на другом узле, то нужно обеспечить его перемещение на нужный узел и только когда все объекты собрались вместе, тогда запустится нужная транзакция (собрать все ресурсы, потом запустить транзакцию)

Если сложно разделить независимые совокупности объектов данных, то может возникать эффект трешинга (некий объект начинает часто мигрировать — эта миграция дает накладные расходы, гранулированность данных-узлы будут ожидать сбора всех данных, а значит запросы встают в очередь — время отклика увеличивается)

Распределенная транзакция (2-х этапная) — где находятся нужные объекты данных и их блокировать, а после уже делается нужная транзакция. Далее снимает блокировку или откатываем транзакцию. Здесь уходим от накладных расходов на перемещение, но попадаем в риск длительных блокировок. Начинается поиск компромиссов: большая или маленькие данные, индексация данных и т.д.

Вариант, что данные существуют в одном экземпляре эффективно работает только с рядом условий:

данные можно относительно разделить на относительно независимые блоки, где взаимосвязь между ними по запросам выше, чем связь между блоками + нагрузка должна более-менее равномерно распределяться

—*Дублирование данных*. Дублирование только на чтение или на чтение и запись

Дублирование только на чтение — если я обращаюсь к какому-то объекту данных, а этого объекта не в хранилище, к которому идет обращение, то запрос на копию этих данных на хранилище к которому идет запрос (эта копия там и останется). А пределе могут быть копии на всех узлах. Дальше попадаем в ограниченность самого объема данных хранимых на узле, поэтому нужны разные алгоритмы вытестенения (по частоте обращений, по кол-ву и т.д.).

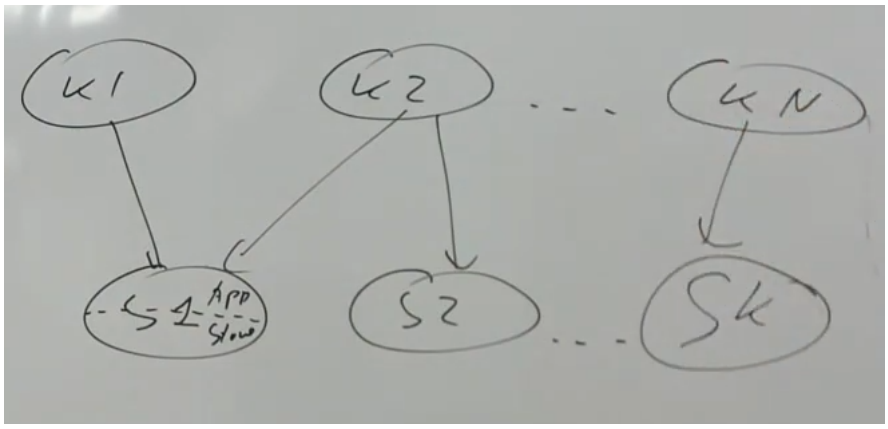
Если данные поменяли на одном узле, то на других узлах данные неактуальные (данные надо удалить). Пока сообщение об удалении будет идти, встанет в очередь на исполнение + уровень ОС — так или иначе это всё задержки и есть вероятность получить неконсистентные данные и сломать всё. И тогда появляется концепция транзакционного удаления — если я хочу внести изменения в данные, то я прошу всех удалить все данные и только когда получаю подтверждение, что данные везде удалены, тогда вношу изменения воставшийся у меня фрагмент. Риски: например, узел недоступен (задержки), даже если веерный опрос состояния. В таком случае запись становится медленной. Но от коллизий не избавляемся: например 2 узла решили внести изменения в один объект — тогда эти узлы пошлют сообщение удалить копию — блокировка, т. к. никто не может удалить копию или надо придумывать алгоритмы консенсуса. Другой вариант — вносить изменения в каждом узле свои, а постфактум разбираться с консистентностью данных. Достигать консистентности за время, но мгновенной консистентности нет. Такое возможно только есть есть правильные изменения в функциональной и информационной архитектуре.

Промежуточный итог: если на уровне обработки можно найти удовлетворительное решение, то на уровне хранения сталкиваемся с существенными трудностями.

Подумать — в каких случаях видели на уровне здравого смысла, что решение было бы полегче тогда, когда удавалось разделить данные на блоки, сильно связанных внутри блока и слабо связанных с точки зрения взаимодействия между блоков (тогда удобно со стороны одной копии или множества копий). Это всё привело к тому, что идея различных клиент-серверных решений с распределенными слоями трансформировалось в SOA (сервис-ориентированную архитектуру)

Сервис-ориентированная архитектура

Идея объединение приложений с их хранилищами и между собой разрешим им как-то взаимодействовать



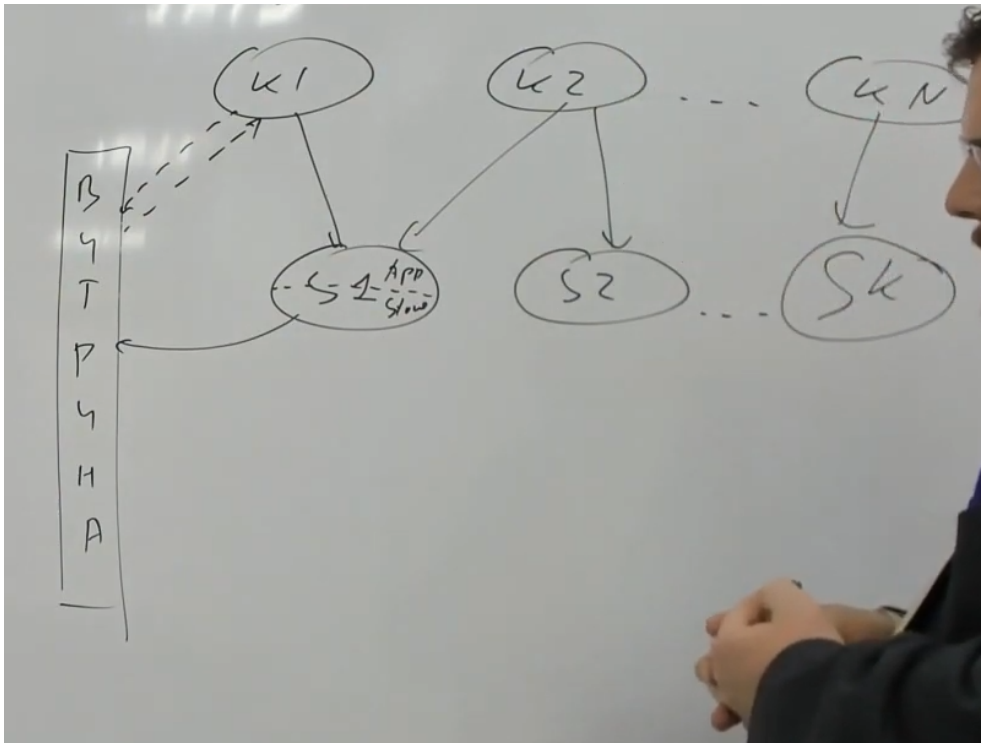
Получаем:

-клиенты

-сервисы (сервисы разделяются по функционалу для того чтобы сделать относительно независимыми их хранилища, чтобы вероятность того, что нам потребуются данные из хранилищ разных сервисов была меньше, чем вероятность того, что потребуются данные в хранилище одного сервисе). Можно делать масштабирование сервиса одного типа. Внутри сервиса останется та же многоуровневая модель (приложение и хранилище), но это всё абстрагировано интерфейсом этого сервиса

Как клиенты узнают, что существует сервис с которым можно работать?

Появляется необходимость компонента — витрина (там может быть разное : ip-адреса и др.)

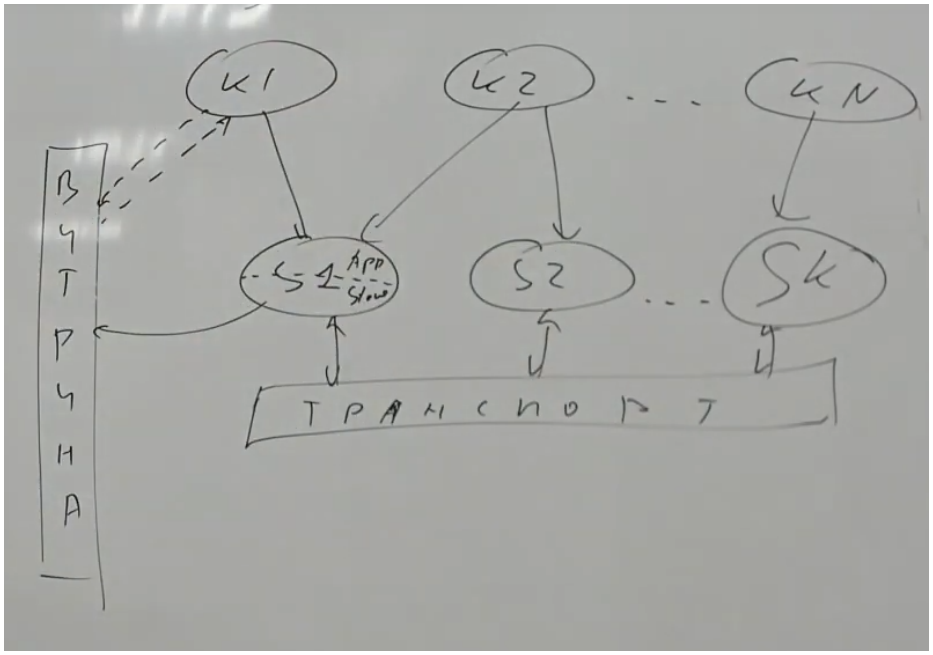


В такой модели сложно балансировать нагрузку (по какой логике будут забирать адреса сервисов, если их несколько и они могу выполнить один и тот же запрос)

Узкое место — витрина. Если откажет витрина, то откажет весь сервис

Иногда может быть ситуация, когда нужно действовать несколько сервисов — тогда возникает задача интероперабельности сервисов (взаимодействие сервисов между собой).

При решении этой задачи возникает транспорт



интероперабельности сервисов обычно разделяют на два уровня

— техническая интероперабельности (используем стандартизованный протокол СОА — передавать xml, которые можно опубликовать даже на витрине) Её

недостаточно — например, проблема синхронизации времени между сервисами (правильная интерпретация чисел, которые пришли от сервисов).

— сементическая интероперабельности (правильная интерпретация чисел, которые пришли от сервисов)

В модели СОА понятия клиента и сервера уже не являются стабильными, они могут меняться местами.

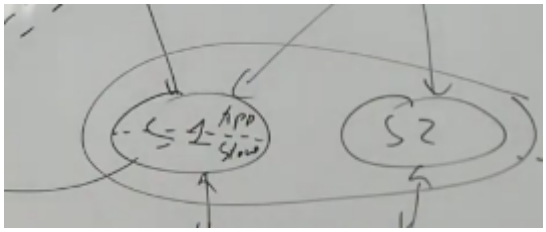
Если сервисы не могут взаимодействовать между собой, то интероперабельность нарушена и большинство операций выполнить нельзя. Решения такой ситуации — решения на распределенных инфраструктурах. Суть высокопроизводительные сервисы, которые обеспечивают множество очередей (не системы массового обслуживания,а способ локального транспорта совмещенного с витриной). Суть:

Сервис появляется и подписывается на очереди, публикую свой интерфейси начинает слушать если ему что-то придет, адресованное ему. Клиент также обращается в нужную очередь (запросы в очередь идут, а сервисы разбирают очереди). Очереди можно дублировать (если сервис с очередью не работает, то обращение идет в другую очередь)

Но есть проблема: чтобы сделать модель СОА эффективной, то сервисы часто делают всё меньше и меньше, т. к. маленький сервис удобно сохранять независимость, много экземпляров делать, управлять производительностью (подняли или убрали сервисы).

Чем меньше гранулируем сервисы, тем гибче управление производительностью системы.

Для одного маленького сервиса жирно выделять сервер, поэтому несколько сервисов должны оказаться на одном аппаратном узле.



Нужно быстро поднимать сервисы и желательно автоматически. т. к. в ручном режиме это может быт долго, а если еще через пару дней, то может быть вообще неактуально.

Поэтому естественным развитием пошла *модель контейнерезации* (она была откликом на концепцию облачных решениях).

Идея облака: множество аппаратных ресурсов разделяли между множеством из потребителей и при этом не были привязаны к одному экземпляру ОС (если один экземпляр, то будет сложно иметь несколько экземпляров субд, серверов приложений и т. д. + совместное использование ресурсов — будут блокировки,

конфликты и т. п. И не сможем эффективно использовать решение) Поэтому разумно сделать связяку ОС с хранилищами и сама ОС нативно сделает изоляцию компонента (инкапсуляция сделаем) в виртуальную машину. Эти концепции облаков часто строятся на полной виртуализации.

Сервис — раздел где сущ.виртуальная машина с ОС, но это эффективно для крупных сервисов. Создание ОС — много ресурсов жрет + есть планировщик ОС и ниже планировщик на уровне гипервизора (они не общаются между собой)

Для маленького сервиса заряжать большую ОС — жирно, поэтому здесь как решение возникает контейнерезация. Контейнерезация пришла из виртуализации уровня ядра ОС. Суть:

ядро оставим одно (снизив накладные расходы), а пространство для другого (имен процессо и т. д.) сделаем независимыми нужной степени друг от друга, виртуализированными. Дальше если так можно сделать, то можно передавать контейнер, где всё собрано и сконфигурировано и легко из библиотек поднимать соответствующие сервисы. И тогда получается модель которая дает приемлемые накладные расходы, а с другой может обеспечить реализацию микросервисной архитектуры.

Но нужно понимать, что идеального решения нет!

Разные решения по архитектуре существуют, т. к. каждая из этих технологий имеет свои ограничения и находит свою нишу применения.

Контейнерезация это супер, но мы попадаем в ограниченность ядер ОС

Если делать независимую контейнерезацию, то надо создавать слои эмуляции чтобы обеспечивать нужный уровень абстракции. Т.е. контейнер способный работать с одним ядром должен мочь работать в другой ОС с другим ядром. Такой слой абстракции сразу несет в себе проблемы надежности и безопасности.

Для каждого решения нужен свой анализ сценария, свой анализ рисков.

Архитектура — это концептуальное решение,а выбор технологий это след.уровень.

Предыдущая лекция