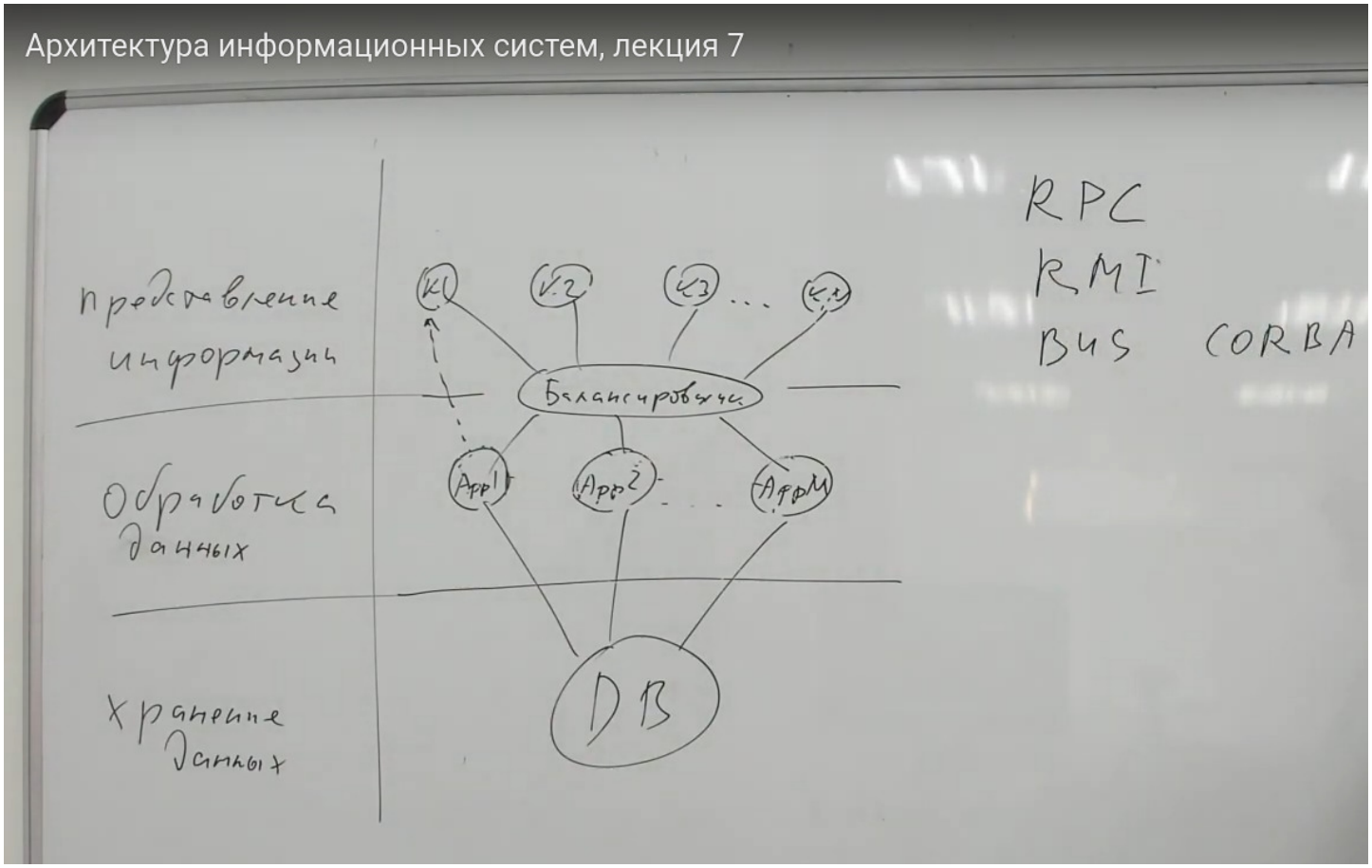
Перейти к описанию и содержанию конспекта.

Ссылка на лекцию




Была информационная и функциональная архитектура в пред.лекциях. Здесь рассматривается техническая архитектура. Тех.арх. Можно разделить на 3 части (point of view):

— системная архитектура,

— программная архитектура

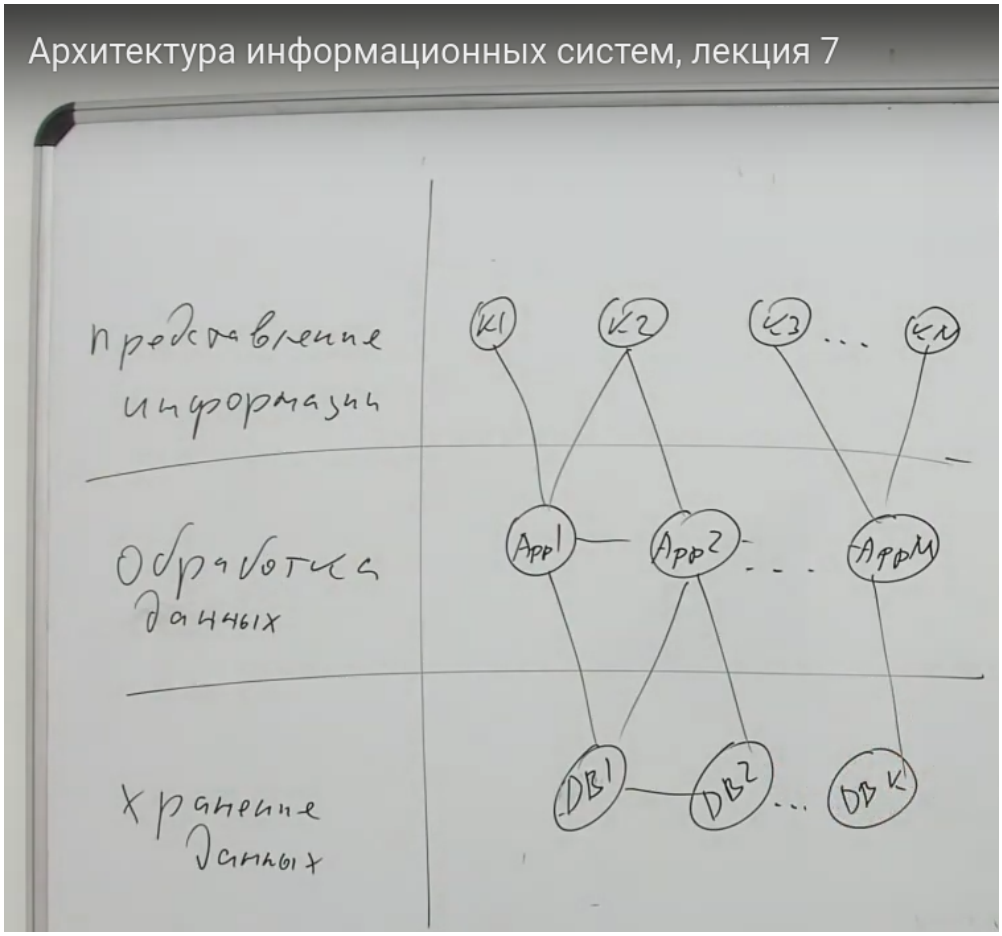
— архитектура данных

Программная архитектура и архитектура данных очень сильно зависти от архитектуры данных

ИС, в большинстве случаев, может эффективно работать в многопользовательском режиме, а значит необходимо разделять слой представления данных от слоев обработки и хранения данных. Дальше пошло развитие по абстрагированию этих слоев. Как следствие, начала развиваться распределенная архитектура ИС.

**Эволюционное развитие распределенной архитектуры ИС**

(см. рисунок далее)




Это условная схема, т. к. возможны разные варианты реализации (узлы обработки независимы или зависимы друг от друга, один экземпляр хранения или несколько, мб клиент привязан к конкретному клиенту и т.д.)

Цели создания распределенной архитектуры:

1. *повысить производительность* за счет параллельной обработки данных (т. к. один узел имеет технологический предел)

Т.к. есть предел, то можно попытаться посчитать и найти равновесную точку между расширением выч.ресурсов узла и момента когда увеличение ресурсов мало будет приносить улучшения в производительности узла.

Иногда можно строить системы (сети) на основе дешевых компонентов и если один из компонентов сети выйдет из строя, то легче купить еще один узел такой, чем как-то восстанавливать, т. к. этот компонент дешевый + низкоквалифицированный персонал на уровне вытащил-вставил.

Или наоборот когда сложные, дорогие системы, дорогой штат системных инженеров, которые и будут это всё обеспечивать.

Так или иначе есть ограничение по производительности узла, поэтому приходим к тому, что нужно распределять обработку между узлами.

2. *снизить телекоммуникационные расходы* за счет приближения обрабатываемых узлов к потребителю.

Говорят, что географическое расстояние перестает играть роль, но это не так, если большое кол-во запросов идет, а не один какой-то запрос.

Ситуация. Много клиентов, которые в разных точках находятся и обращаются к одному монолитному серверу. Один идет по ненагруженой сети, другой по загруженной, нестабильной сети и здесь сеть может быть узким местом.

ИС работают в рамках сессии. Т.е. открывается сессия и начинается обмен данными, пакетами данных (+проверки, подтверждения, протоклы и т.д.). Пакеты собираются в массив для дальнейшей обработки. Т.е. серевер начинает загружаться при взаимодействии с клиентов, где сеть имеет нестабильное состояние (проверка целостности, подтверждения и т. д.). Таким образом сервер может помешать тем клиентам, которые проблем с сетью не имеют.

Условная ситуация: один клиент запустил транзакцию, но из-за проблем с телекоммуникацией эта транзакция может обрабатываться долго, поэтому другие клиенты ждут пока та транзакция закончится.

Поэтому участок между пользователем и узлом обработки данных был более-менее одинаков в плане возможности производительности для большинства клиентов.

Связей между обработкой и хранилищем часто меньше, чем между клиентом и узлом обработки. + можно более эффективно управлять сетями между обработкой и хранилищем (мониторинг, реагирование и т. д.) Сетями к пользователям управлять не можем, т. к. они чужие(солжные маршруты, vpn и т. д.)

Т.е. слой обработки данных лучше максимально возможно географически придвинуть к пользователю с целью минимизации хопов до узлов обработки. Пример:CDN (т. е. статика отдается ближе, а динамически изменяемый контент дальше от клиента — можно сказать грубо — кэширование). Статичность данных только на определенном отрезке времени, т. к. есть вопрос актуальности данных (обновляется статика из-за развития технологий, изменений в коде и т.д.). Т.е. данные консистентны только в какой-то период времени.

3. *повысить надежность за счет дублирования* узлов и хранилищ данных

Под надежность понимается гарантия доступности функционала нашей системы в любой момент времени. Т.е. потенциал отказа не превышал какую-то величину или период доступности не превышал какое-то время и т. д.

Если есть множество серверов приложений и клиенты могут обращаться к разным серверам приложений, то если выйдет один сервер приложений, то пользователи просто перейдут к следующему доступному. (но нагрузка возрастет на оставшиеся, но система будет доступна, процессы не остановятся. Мб снизится время откликов).

Такой же подход касается и данных.

Узел распределенной обработки можно держать в горячем резерве, т. к. узлы в плане обработки мало меняются.

Распределенность данных сложнее, т. к. важна консистентность данных. Если выйдет один узел из строя, а потом другой узел вводят на место сбоя, то в этом новом узле должны быть те данные, которые были в момент сбоя. Т.е. нужна постоянная репликация, т. к. если не репликации и данные потеряны, то невозможно будет их восстановить, потому что они собраны за какое-то время и специально обработаны.

4. *повышение масштабирования за счет абстрагирования узлов*. В случае монолита возможны проблемы очередей, когда сложная задача съедает много ресурсов, а простые забивают очередь и ресурсы все могут быть задействованы, в результате чего идет влияние на бизнес-процессы.

Возможное решение: компонентов обработки может быть несколько и эти компоненты могут быть распределены по уникальной функциональности (один переиндексирование, другой на формирование данных и т.д.). Такое решение позволяет масштабировать систему (добавлять или убираться нужные компоненты обработки по функционалу)

Сложности распределенной архитектуры построении:

-триада (производительность, надежность, функциональность)+нефункци.требования

-целостность, доступность и устойчивость к разделению данных (CAP-теорема или теорема Брюера)

-оптимизировать использование ресурсов при равномерной нагрузке (разные запросы по трудоемкости, но трудоемкость сложно оценить и будут всё равно вероятностные характеристики)

-синхронизация событий и проблемы взаимоблокировок, тупиков (deadlock, кольцо запросов к нескольким данным). Разные узлы обрабатывают данные в по таймерам, а время на каждом узле течет по своему (рассуждения лампорта о невозможности глобального таймера

-стандартизация интерфейсов. Хорошо бы унифицировать интерфейсы, но это накладные расходы. Любое абстрагирование интерфейсов — это нагрузка на надежность и безопасность.

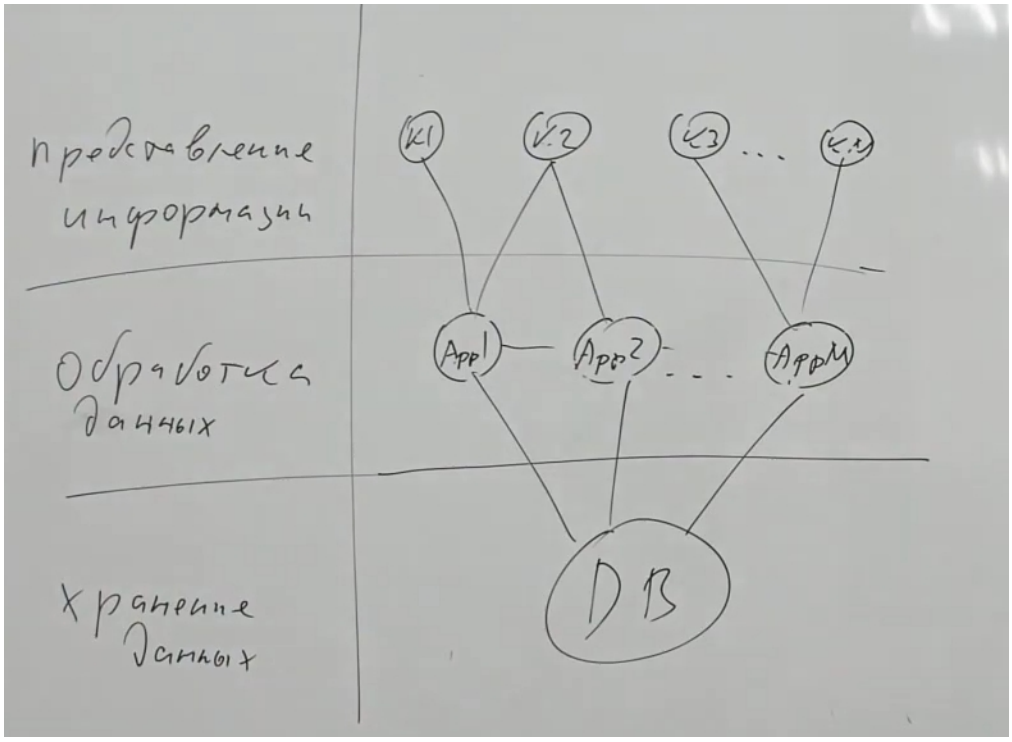
-мониторинг и диагностика. Когда монолит, то легко мониторить и диагностировать. Если распределенная, то это сложный мониторинг, диагностика и консолидация данных по ним (кто свободен, кто занят, кому нельзя посылать данные и т.д.). Противоречие для принятия решений нужна актуальная информация о всех узлах (т. к. всё постоянно меняется) и расход вычислительных ресурсов на эту актуальную информацию. Т.е. получение актуальной информации может быть нивелировано расходами, которые мы понесли получая инфу.

**Промежуточный итог:**

Обозначили цели и возникающие сложности для достижения этих целей

Разделяем на решения в рамках клиент-серверной архитектуры и сервис-ориентированные архитектуры.

**С точки зрения клиент-серверной архитектуры**




Т. е. **клиент-серверная архитектура с распределенным слоем приложений**

Ситуация. *Любой клиент может обращаться к любому приложению*.

Но это не значит, что нагрузка будет равномерной, т. к. может быть, что много клиентов будут обращаться к одному узлу чаще,а к другим реже.

В случае такой ситуации можно по лимитам запросов управлять нагрузкой (обращаться к следующему узлу пока кто-то не согласится принять запрос)

Или же узел может принять запрос и попросить выполнить этот запрос соседа, но тогда нужно сделать горизонтальные связи чтобы узлы могли обмениваться между собой информацией.

Вопрос делегации задачи. Как делегировать? Если:

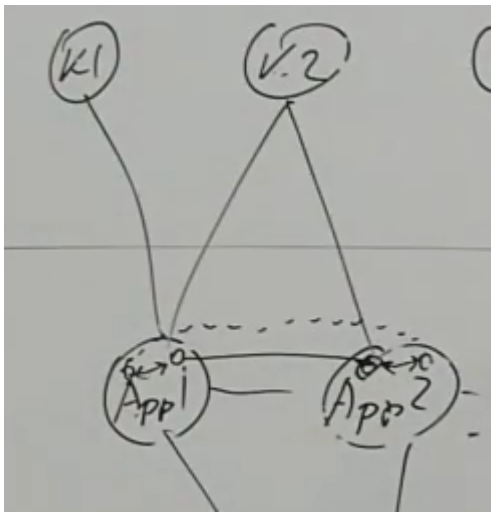
-ограничения ОС есть (своя память, адресное пространство, изолированный процесс и т.д.)

-как реализована программная архитектура приложения и можно ли обрабатывать данные с делегированием

Е*сли это не ООП вариант,то часто используют подход RPC* (remote procedure call).

Как это работает?

Есть процедура, которая выполняется на одном приложении. Она должна вызвать выполнение процедуры, находящейся на другом приложении. Делает это через комбинацию заглушек. Заглушка получив вызов, оборачивает этот вызов в нужные параметры и передает другой заглушке на другой узел и эта заглушка передает на обработку процедуру. Обратно уже вернется результат обработки.




Метод даёт обойти изолированность процессов на узлах, но этот метод сопряжен с накладными расходами (две доп процедуры, телеком расходы, надежность, безопасность и т. д.) Возможно непредсказуемая процедура передачи данных, выход за пределы ОС и т. д. Иначе нужно обеспечивать безопасность (допрасходы на шифрование, дешифрование и т.д.)

*Если ООП решение, то возможна серилизация объектов и их передача.*

Есть объект и методы его обработки. Мы может экземпляр класса соответствующий объект упаковать, сжать и передать его на другой узел и как-то заставить его выполнять нужные методы над данными, но в контексте другого узла. Это проще может быть в плане безопасности и надежности, но нужно чтобы контекста совпали. Т.к. объект может содержать большой объем данных и надо как-то это всё правильно обработать, то такой метод не всегда можно применять, а только тогда когда могут быть хорошие условия (массивы данных в объектах могут быть небольшими, а методы будут требовать серьезных вычислительных ресурсов для обработки этих данных).

Можно попробовать сделать еще один вариант RMI (Remote Method Invocation — удаленный вызов метода) называется в java, но технология в том или ином видеть есть и в других стеках использоваться.

Здесь объекты остаются у себя, где они созданы (создали и продолжает существовать на своем узле), но мы может метод этого объекта может вызвать с другого узла. Но тогда возникают вопросы:

-как обеспечить публичность этих методов (витрину чтобы знать где что есть и есть ли вообще)

-вопросы синхронизации

В сути взаимодействии лежит аналог заглушек как в RPC,только объекты - заглушки:

Объекты считают, что вызывают метод в своем контекте на самом деле этот объект заглушка привел к тому чтобы использовать или создать скелетона объекта на другой стороне и уже от его имени уже произошла обработка нужного объекта. После уже произошел callback. Т.е. в любом случае на одном узле надо данные как-то упаковать и передать,а на другом узле эти данные должны быть распакованы и обработаны и назад упаковать-передать-обработать.

Если эти объекты умеют доступ к одному хранилищу, то супер (обеспечиваем транзакционность и поехали), тогда при вызове удаленного метода мы должны передать небольшой объем данных с точки зрения упакованных параметров. Если нет, то будем передавать большие массивы данных и тогда может быть, что не будет никакого выйгрыша

Всё же эти механизмы приводят к дублированию передаваемых данных, т. к. вынуждены передавать большое кол-во данных, т. к. не знаем кто и куда будет обращаться

*Третий вариант — BUS (шина)*

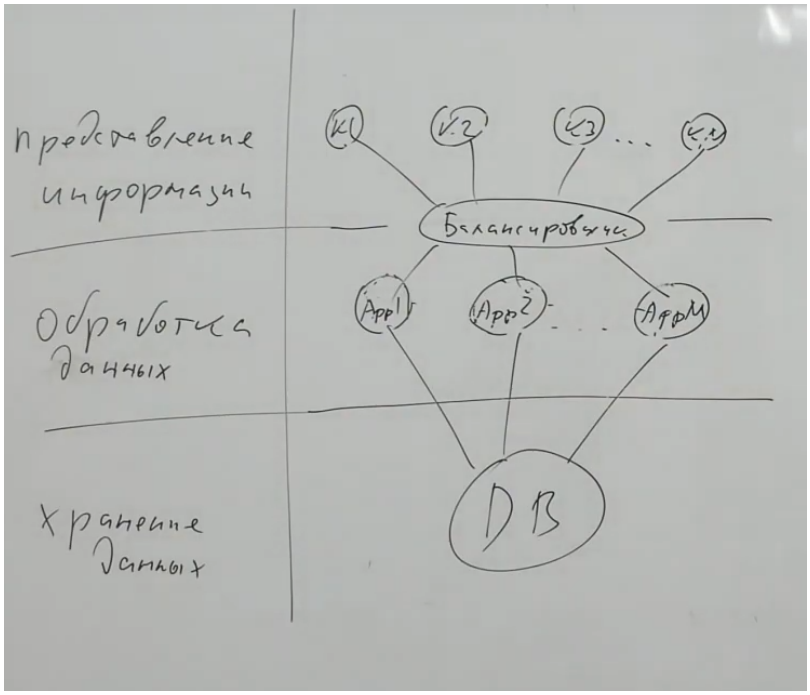
Пример — corba (Common Object Request Broker Architecture)

Идея — каждое приложение взаимодействует с брокером с какой-то универсальной шиной, универсальным интерфейсам. Это надо для чего?

Есть какой-то унифицированный язык описания интерфейсов, который позволяет любому приложению взаимодействовать с другим интерфейсом. Есть брокеры ресурсов и модель подписки чтобы слушать сообщения. Слушаем-обрабатываем запрос если можем и отдает по транспорту ответ.

Рассматривали варианты когда мы считаем, что любой узел к любому приложению независимо от того где это приложение находится. Как видим, такой подход требует решения многих сложных задач балансировки во взаимодействии между узлами.

**Балансировщик**




В случае когда узлы приложения не знают друг о друге ничего, тогда между ними нужно поставить некого посредника — балансировщик

Балансировщик является единой точкой входа. К нему обращаются все клиенты, а баланс решает кому что передать по запросам. Но тогда баланс — узкое мест. Можно рассматривать варианты, когда балансировщики резервируются, а на клиентах настроено простое правило:

пытаемся обратить к основному балансу, если нет через несколько попыток, то к другому.

Тогда надо держать в горячем резерве еще один или несколько балансов, т. к. надо понимать кто главные в моменте времени (т. к. управлять распределением будучи самим распределенным — усложнение значит, поэтому главный один, а другой жрет ресурсы в горячем резерве только чтобы в нужный момент заменить главного чтобы минимизировать риски недоступности системы в случае отказа основного баланса)

Балансы делятся на:

*синхронный балансировщик:* держит сессию у себя. К нему пришел запрос-выбрал кто выполнит-узел обработал-баланс отдал клиенту результат. Нагрузка на вход и на выход. В вебе это разная нагружка. Запрос(вход) — небольшая нагрузка, ответ(выход) -большая нагрузка (код на java, и т. д.). На входи мб и выдержим множество подключений,а на выход с одного сетевого интерфейса упираемся в проблему отдачи данных.

*асинхронный:* клиент обращается к балансировщику — тот выбирает сервер приложений который будет сопровождать в рамках сессии или запроса и передает ему не только запрос и информацию о том кому и как отдать ответ — тогда уже приложение выполнив запрос отдает результат клиенту. Минус такого балансира с пассивной оценкой загрузки узлов в том, что балансир не может отслеживать

результат ответов по запросам, а значит не знает насколько сильно нагружен тот или иной узел, а балансир продолжает слать запросы. Тогда балансир должен переходит к активной оценки узлов — опрашивать надо узлы, а значит будет доп нагрузка будет на узлы (актуальности инфы, нагрузка, накладные расходы и т. д.на узлы)

Можно *использовать мультиагентных* подходов, когда мы на каждое приложение вешаем агента, который следит и информирует о состоянии узлов по пороговым значениям.

Агент говорит загружен узел — тогда балансир перестаете нагружать узел. Если пиковая нагрузка, то балансир может поменять порог. Если снизилась нагрузка, то снизили порог

таким подходом можно уйти от постоянных запросов о состоянии, но требует постоянно работающего агента и сложно технологической разработки логики всего взаимодействия всех объектов и других аспектов надежности, безопасности и т.д.

*Все эти рассмотрения трехзвенной архитектуры исходили из того, что вопрос целостности данных не волнует, т. к. хранилище данных централизовано. Поэтому решив вопросы обработки, рано или поздно упремся в хранилище данных всёравно (один интерфейс, нагрузка растет и т. д.)* Нужно решать ситуацию с хранилищем данных.

Предыдущая лекция

Следующая лекция