[**Архитектура программного обеспечения**](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F)

**1.Определения**

**1.1 Архитектура** **программного** **обеспечения** (англ. software **architecture**) — совокупность важнейших решений об организации **программной** системы. **Архитектура** включает: выбор структурных элементов и их интерфейсов, с помощью которых составлена система...

Область компьютерных наук с момента своего образования столкнулась с проблемами, связанными со сложностью программных систем. Ранее проблемы сложности решались разработчиками путём правильного выбора структур данных, разработки алгоритмов и применения концепции разграничения полномочий. Хотя термин «архитектура программного обеспечения» является относительно новым для индустрии разработки ПО, **фундаментальные принципы этой области неупорядоченно применялись пионерами разработки ПО начиная с середины 1980-х**. Первые попытки осознать и объяснить программную архитектуру системы были полны неточностей и страдали от недостатка организованности**, часто это была просто диаграмма из блоков, соединенных линиями.**

В 1990-е годы наблюдается попытка определить и систематизировать основные аспекты данной дисциплины. Первоначальный набор шаблонов проектирования, стилей проектирования, передового опыта (best practices), языков описания и формальная логика были разработаны в течение этого времени[[6]](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F#cite_note-6).

**Основополагающей идеей дисциплины программной архитектуры является идея снижения сложности системы путём абстракции и разграничения полномочий.**

На сегодняшний день до сих пор нет согласия в отношении чёткого определения термина «архитектура программного обеспечения».

Являясь в настоящий момент своего развития **дисциплиной без четких правил о «правильном» пути создания системы, проектирование архитектуры ПО все ещё является смесью науки и искусства.**

Аспект «искусства» заключается в том, что любая коммерческая система подразумевает наличие применения или миссии. С точки зрения пользователя программной архитектуры, программная архитектура дает направление для движения и решения задач, связанных со специальностью каждого такого пользователя, например, заинтересованного лица, разработчика ПО, группы поддержки ПО, специалиста по сопровождению ПО, специалиста по развертыванию ПО, тестера, а также конечных пользователей. В этом смысле архитектура программного обеспечения на самом деле объединяет различные точки зрения на систему. Тот факт, что эти несколько различных точек зрения могут быть объединены в архитектуре программного обеспечения, является аргументом в защиту необходимости и целесообразности создания архитектуры ПО ещё **до этапа разработки** ПО[[7]](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F#cite_note-7)[[8]](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F#cite_note-8)[[9]](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F#cite_note-9).

**1.2 История**

Начало архитектуре программного обеспечения как концепции было положено в научно-исследовательской работе Эдсгера Дейкстры в 1968 году и Дэвида Парнаса в начале 1970-х. Эти ученые подчеркнули, что структура системы ПО имеет важное значение, и что построение правильной структуры — критически важно. Популярность изучения этой области возросла с начала 1990-х годов вместе с научно-исследовательской работой по исследованию архитектурных стилей (шаблонов), языков описания архитектуры, документирования архитектуры, и формальных методов.

В развитии архитектуры программного обеспечения как дисциплины играют важную роль научно-исследовательские учреждения. Мэри Шоу и Дэвид Гэрлан из университета Carnegie Mellon написали книгу под названием «Архитектура программного обеспечения: перспективы новой дисциплины в 1996 году», в которой выдвинули концепции архитектуры программного обеспечения, такие как компоненты, соединители (connectors), стили и так далее. В калифорнийском университете институт Ирвайна по исследованию ПО в первую очередь исследует архитектурные стили, языки описания архитектуры и динамические архитектуры.

Первым стандартом программной архитектуры является стандарт IEEE 1471: ANSI / IEEE 1471—2000: Рекомендации по описанию преимущественно программных систем. Он был принят в 2007 году, под названием ISO ISO / IEC 42010:2007.

**1.3 Языки описания архитектуры**

Языки описания архитектуры (ADLS) используются для описания архитектуры программного обеспечения. Различными организациями было разработано несколько различных ADLS, в том числе AADL (стандарт SAE), Wright (разработан в университете Carnegie Mellon), Acme (разработан в университете Carnegie Mellon), xADL (разработан в UCI), Darwin (разработан в Imperial College в Лондоне), DAOP-ADL (разработан в Университете Малаги), а также ByADL (Университет L’Aquila, Италия). Общими элементами для всех этих языков являются понятия компонента, коннектора и конфигурации. Также, помимо специализированных языков, для описания архитектуры часто используется унифицированный язык моделирования UML.

**1.4 Виды (views)**

Архитектура ПО обычно содержит несколько видов, которые аналогичны различным типам чертежей в строительстве зданий. В онтологии, установленной ANSI / IEEE 1471—2000, виды являются экземплярами точки зрения, где точка зрения существует для описания архитектуры с точки зрения заданного множества заинтересованных лиц.

**Архитектурный вид состоит из 2 компонентов**:

* Элементы
* Отношения между элементами

Архитектурные виды можно поделить на 3 основных типа[[10]](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F#cite_note-:0-10):

1. Модульные виды (англ. *module views*) — показывают систему как структуру из различных программных блоков.
2. Компоненты-и-коннекторы (англ. *component-and-connector views*) — показывают систему как структуру из параллельно запущенных элементов (компонентов) и способов их взаимодействия (коннекторов).
3. Размещение (англ. *allocation views*) — показывает размещение элементов системы во внешних средах.

**Примеры модульных видов:**

* Декомпозиция (англ. *decomposition view*) — состоит из модулей в контексте отношения «является подмодулем»
* Использование (англ. *uses view*) — состоит из модулей в контексте отношения «использует» (т.е. один модуль использует сервисы другого модуля)
* Вид уровней (англ. *layered view*) — показывает структуру, в которой связанные по функциональности модули объединены в группы (уровни)
* Вид классов/обобщений (англ. c*lass/generalization view*) — состоит из классов, связанные через отношения «наследуется от» и «является экземпляром»

**Примеры видов компонентов-и-коннекторов**:

* Процессный вид (англ. *process view*) — состоит из процессов, соединённых операциями коммуникации, синхронизации и/или исключения
* Параллельный вид (англ. *concurrency view*) — состоит из компонентов и коннекторов, где коннекторы представляют собой «логические потоки»
* Вид обмена данными (англ. *shared-data (repository) view*) — состоит из компонентов и коннекторов, которые создают, сохраняют и получают постоянные данные
* Вид клиент-сервер (англ. *client-server view*) — состоит из взаимодействующих клиентов и серверов, а также коннекторов между ними (например, протоколов и общих сообщений)

**Примеры видов размещения:**

* Развертывание (англ. *deployment view*) — состоит из программных элементов, их размещения на физических носителях и коммуникационных элементов
* Внедрение (англ. *implementation view*) — состоит из программных элементов и их соответствия файловым структурам в различных средах (разработческой, интеграционной и т.д.)
* Распределение работы (англ. *work assignment view*) — состоит из модулей и описания того, кто ответственен за внедрение каждого из них

Хотя было разработано несколько языков для описания архитектуры программного обеспечения, в настоящий момент нет согласия по поводу того, какой набор видов должен быть принят в качестве эталона. В качестве стандарта «для моделирования программных систем (и не только)» был создан язык UML.

**Для удовлетворения проектируемой системы различным атрибутам качества применяются различные архитектурные шаблоны (паттерны). Каждый шаблон имеет свои задачи и свои недостатки.**

Примеры архитектурных шаблонов:

* **Многоуровневый шаблон** (Layered pattern). Система разбивается на уровни, которые на диаграмме изображаются один над другим. Каждый уровень может вызывать только уровень на 1 ниже него. Таким образом разработку каждого уровня можно вести относительно независимо, что повышает модифицируемость системы. Недостатками данного подхода являются усложнение системы и снижение производительности.
* **Шаблон посредника** (Broker pattern). Когда в системе присутствует большое количество модулей, их прямое взаимодействие друг с другом становится слишком сложным. Для решения проблемы вводится посредник (например, шина данных), по которой модули общаются друг с другом. Таким образом, повышается функциональная совместимость модулей системы. Все недостатки вытекают из наличия посредника: он понижает производительность, его недоступность может сделать недоступной всю систему, он может стать объектом атак и узким местом системы.
* **Шаблон «Модель-Представление-Контроллер» (Model-View-Controller pattern)**. Т.к. требования к интерфейсу меняются чаще всего, то возникает потребность часто его модифицировать, при этом сохраняя корректное взаимодействие с данными (чтение, сохранение). Для этого в шаблоне Model-View-Controller (MVC) интерфейс отделён от данных. Это позволяет менять интерфейсы, равно как и создавать их разные варианты. В MVC система разделена на:
  + Модель, хранящую данные
  + Представление, отображающее часть данных и взаимодействующее с пользователем
  + Контроллер, являющийся посредником между видами и моделью

Однако, концепция MVC имеет и свои недостатки. В частности, из-за усложнения взаимодействия падает скорость работы системы.

* **Клиент-серверный шаблон (Client-Server pattern).** Если есть ограниченное число ресурсов, к которым требуется ограниченный правами доступ большого числа потребителей, то удобно реализовать клиент-серверную архитектуру. Такой подход повышает масштабируемость и доступность системы. Но при этом сервер может стать узким местом системы, при его недоступности становится недоступна вся система.

**1.5 Базовые фреймворки для архитектуры ПО**

Существуют следующие фреймворки ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *software architecture frameworks*), относящиеся к области архитектуры ПО:

* 4+1
* RM-ODP (Reference Model of Open Distributed Processing)
* Service-Oriented Modeling Framework (SOMF)

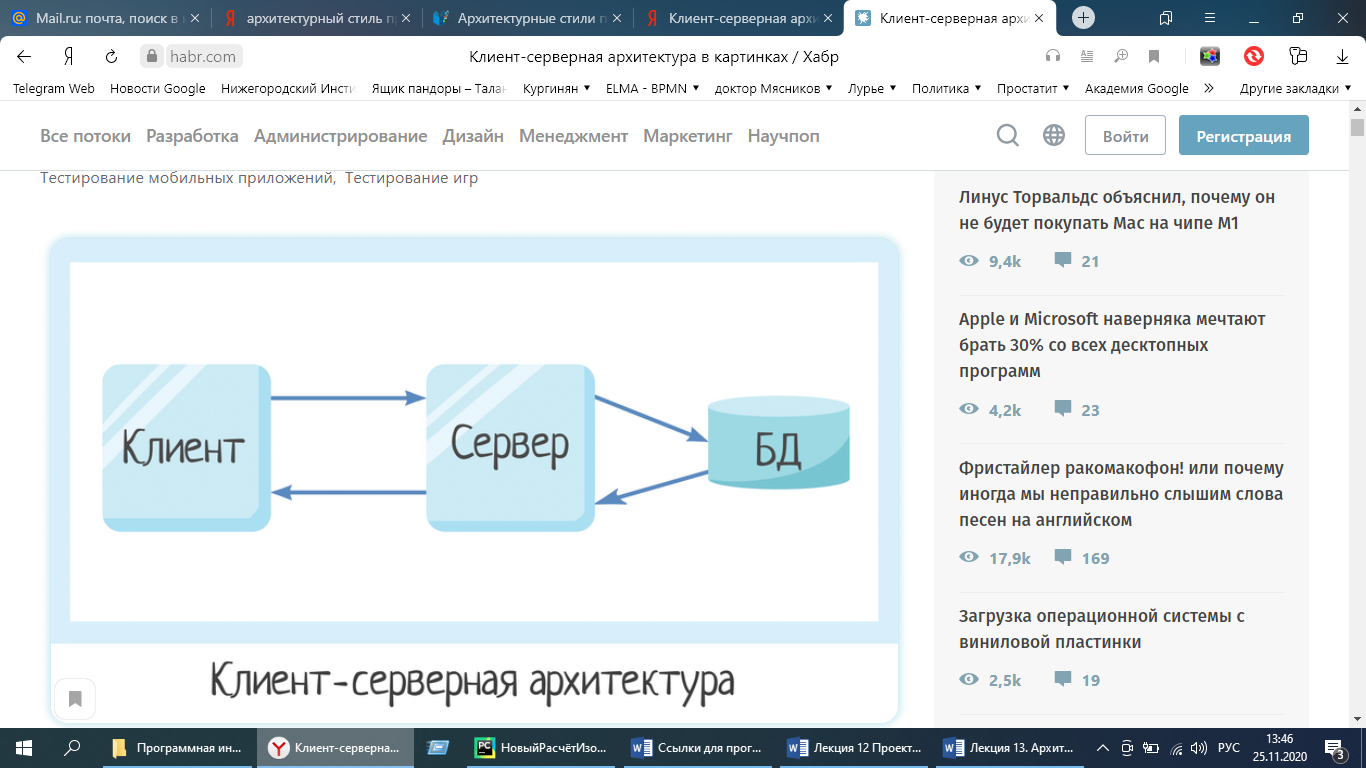
**2 Архитектурные стили проектирования**

Архитектурный стиль (парадигма) может рассматриваться **как обобщенный шаблон для проектирования архитектуры ПО**, опирающийся на некий набор принципов и обеспечивающий абстрактную базу для определенного семейства систем.

**2.1 Типовые архитектурные стили**

**2.1.1 Клиент – серверная архитектура**

Система разделяется на клиентскую и серверную части, где клиент посылает запросы к серверу. Во многих случаях в роли сервера выступает сервер БД, а логика приложения представлена процедурами



<https://www.youtube.com/watch?v=wLHuviTWnuY&feature=youtu.be>

Плюсы архитектуры

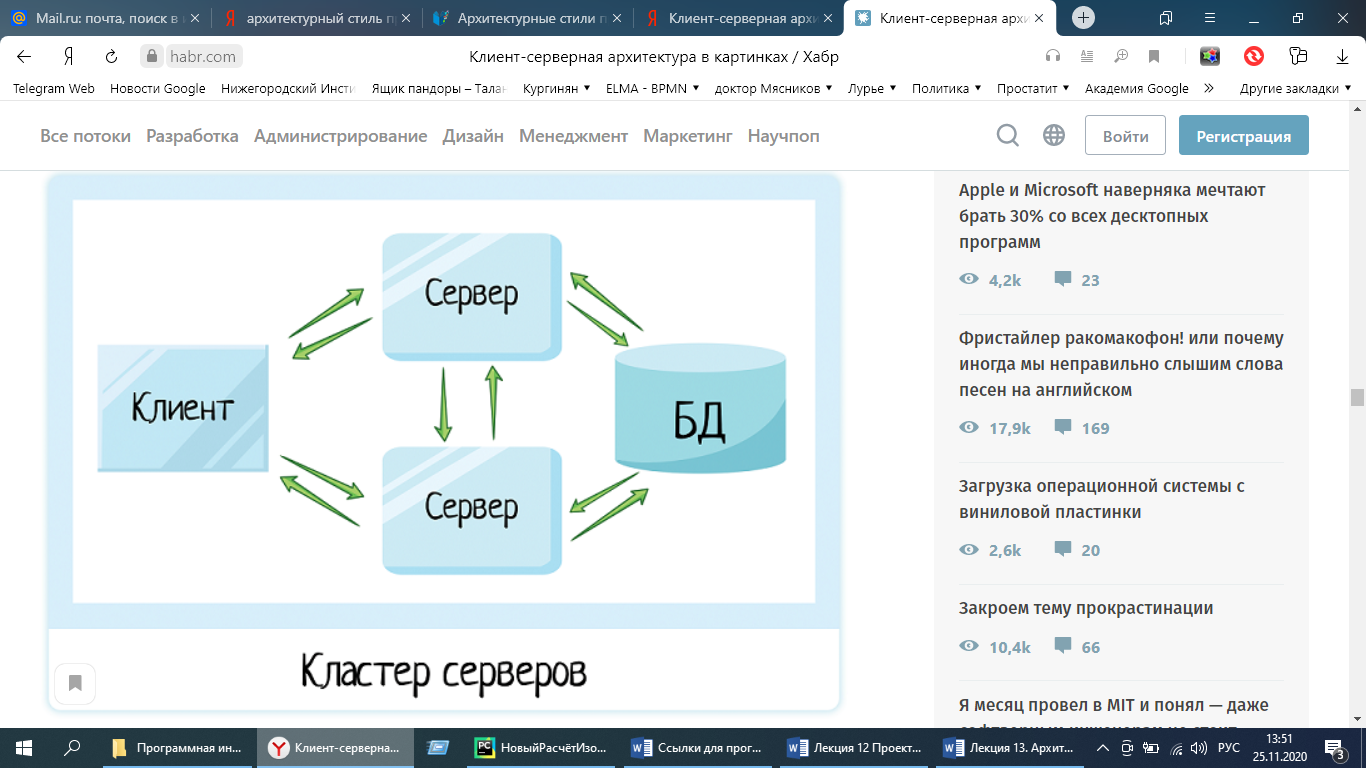
Резюмируем плюсы архитектуры:

1. **Мощный сервер дешевле 100+ мощных клиентских машин** — если мы хотим, чтобы приложение не тормозило, нужна хорошая машина. Она у вас будет одна. Или несколько, если нагрузка большая, но явно меньше, чем количество клиентов.
2. **Нет дублирования кода** — основной код хранится на сервере, клиент отвечает только за «нарисовать красивенько» и простенькие проверки на полях «тут число, тут строка не длиннее 100 символов».
3. **Персональные данные в безопасности** — простой пользователь не видит лишнего. Он не знает ваше ключевое слово, паспортные данные и количество денег на счете.

Минусы архитектуры

**Упало одно звено — все отдыхают.**  
  
Если упал сервер или отвалилась база, то есть испортилось 1 звено — всё, все в ступоре, все отдыхают. Сотни, тысячи, да хоть миллионы клиентов если есть — никто не может работать. Все операционистки грустно смотрят на окно «Простите, что-то пошло не так» и разводят руками перед клиентом.

Именно поэтому в бизнес-критичном ПО архитектуру усложняют и даже дублируют. Банк с тысячами операционистов не может позволить себе простой. **Поэтому они используют кластер серверов — один упал, остальные работают**.



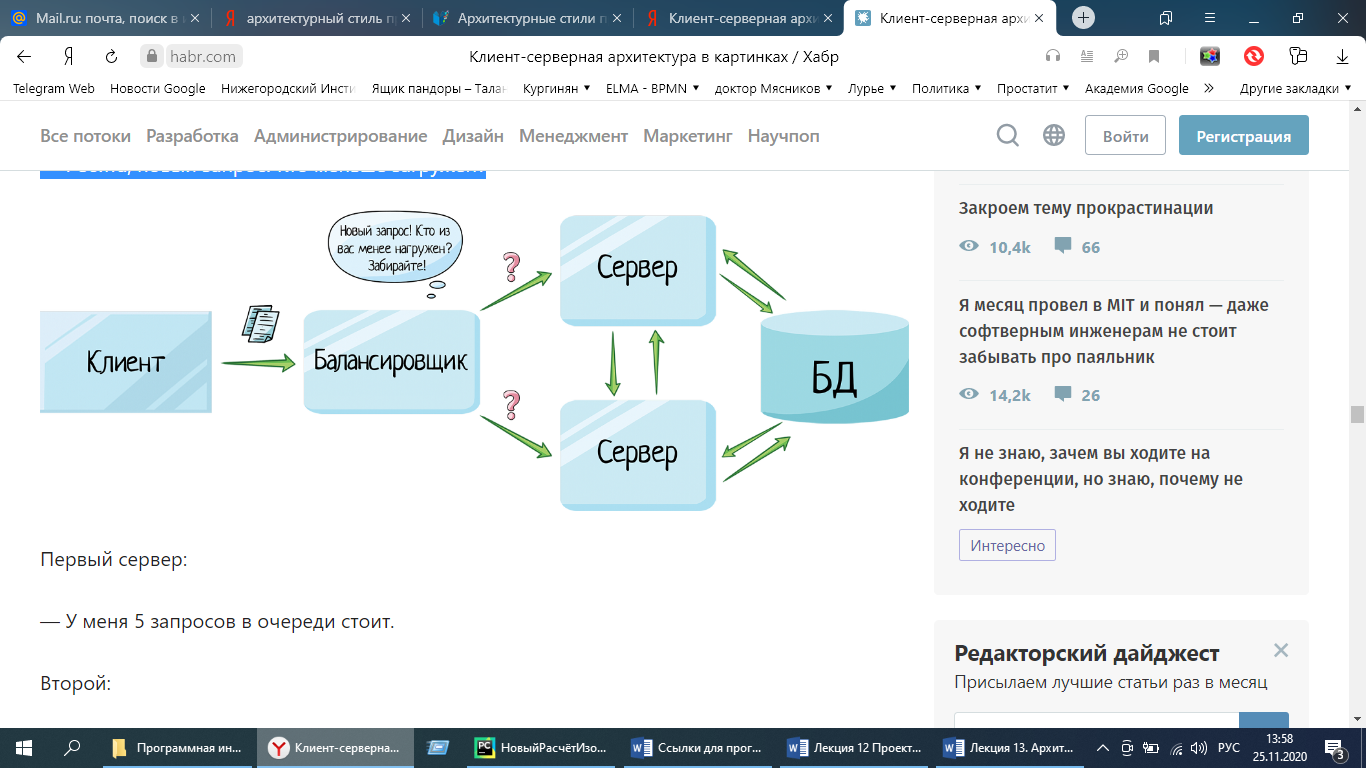
Как в таком случае клиент понимает, куда ему отправлять запрос?

Перед серверами ставят балансировщик, и клиент шлет запрос туда. Сколько бы серверов не поставили в кластер, клиенту это не интересно. У него есть один URL — адрес балансировщика.

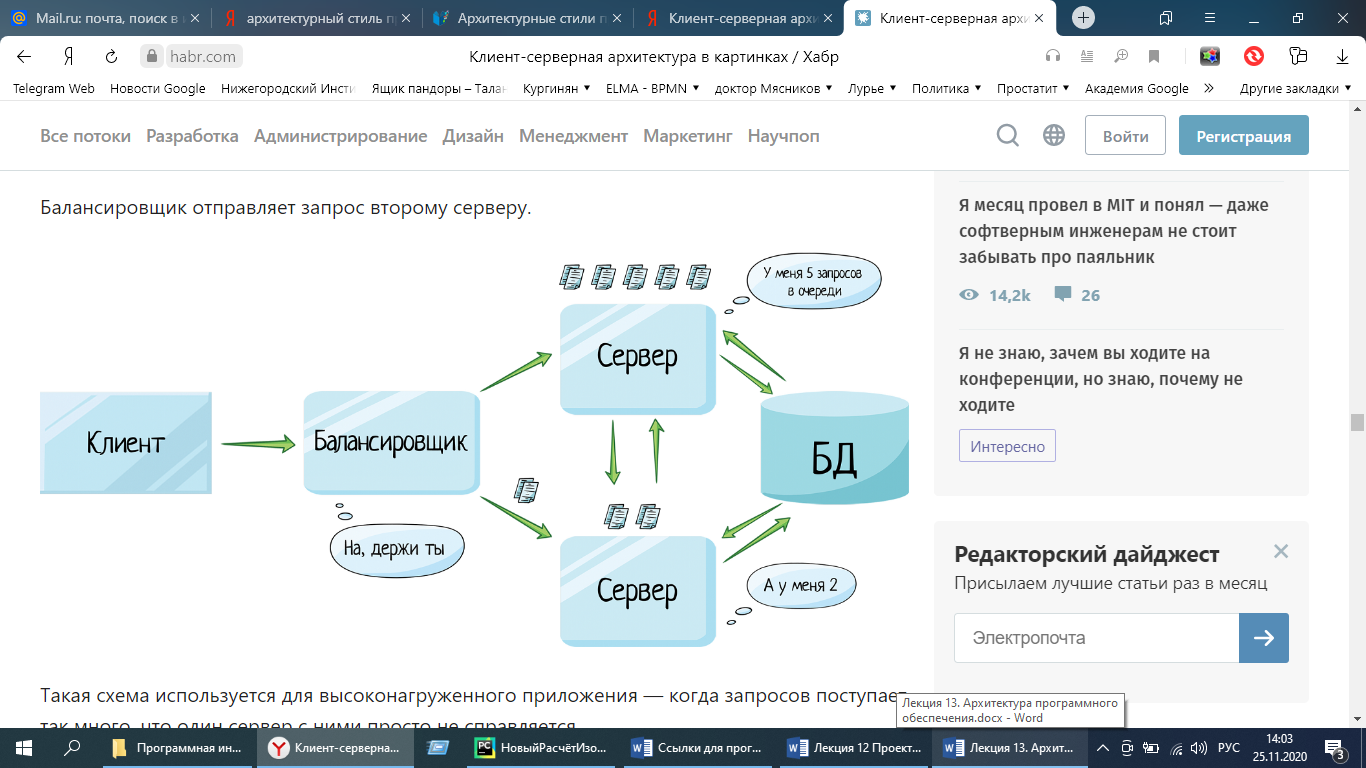


И вот с клиента поступает запрос: — Дай мне всю информацию по Васе Иванову.

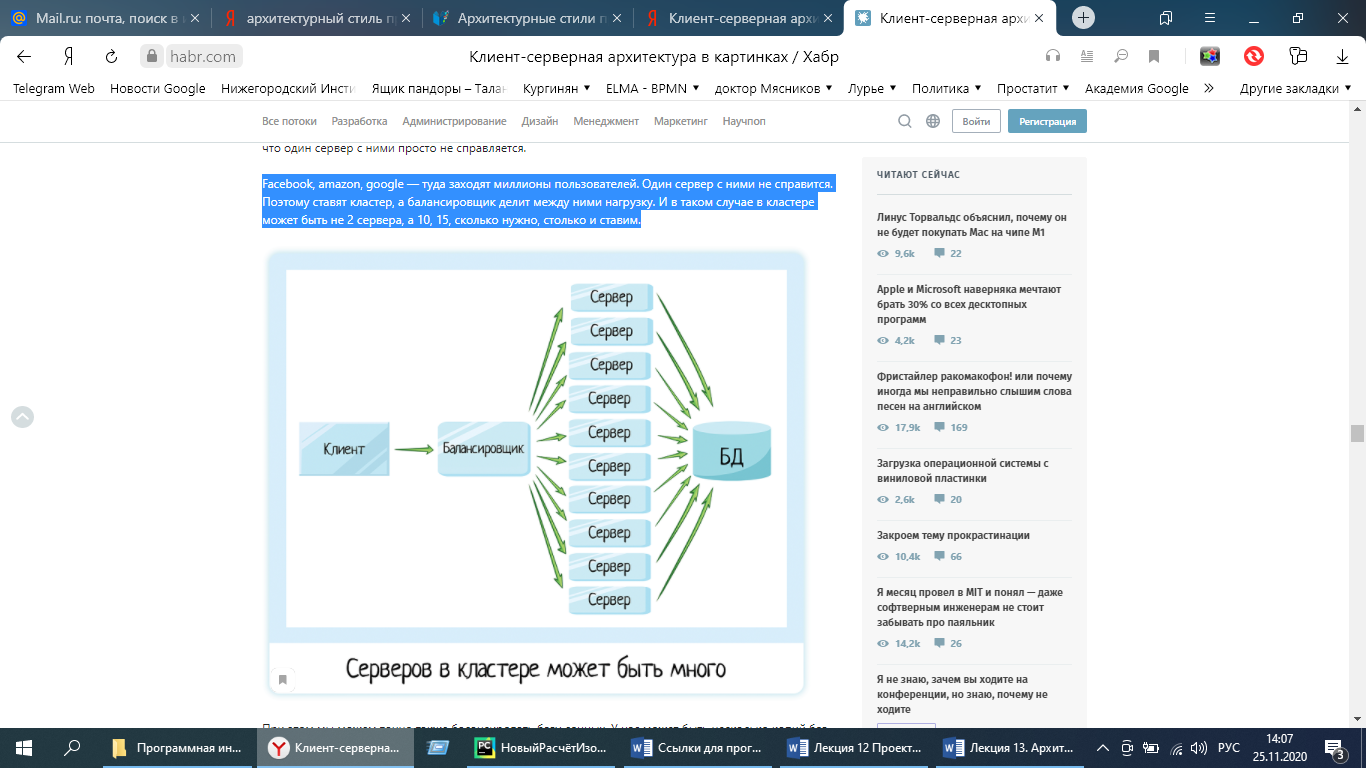
Балансировщик говорит: — Ребята, новый запрос! Кто меньше загружен?



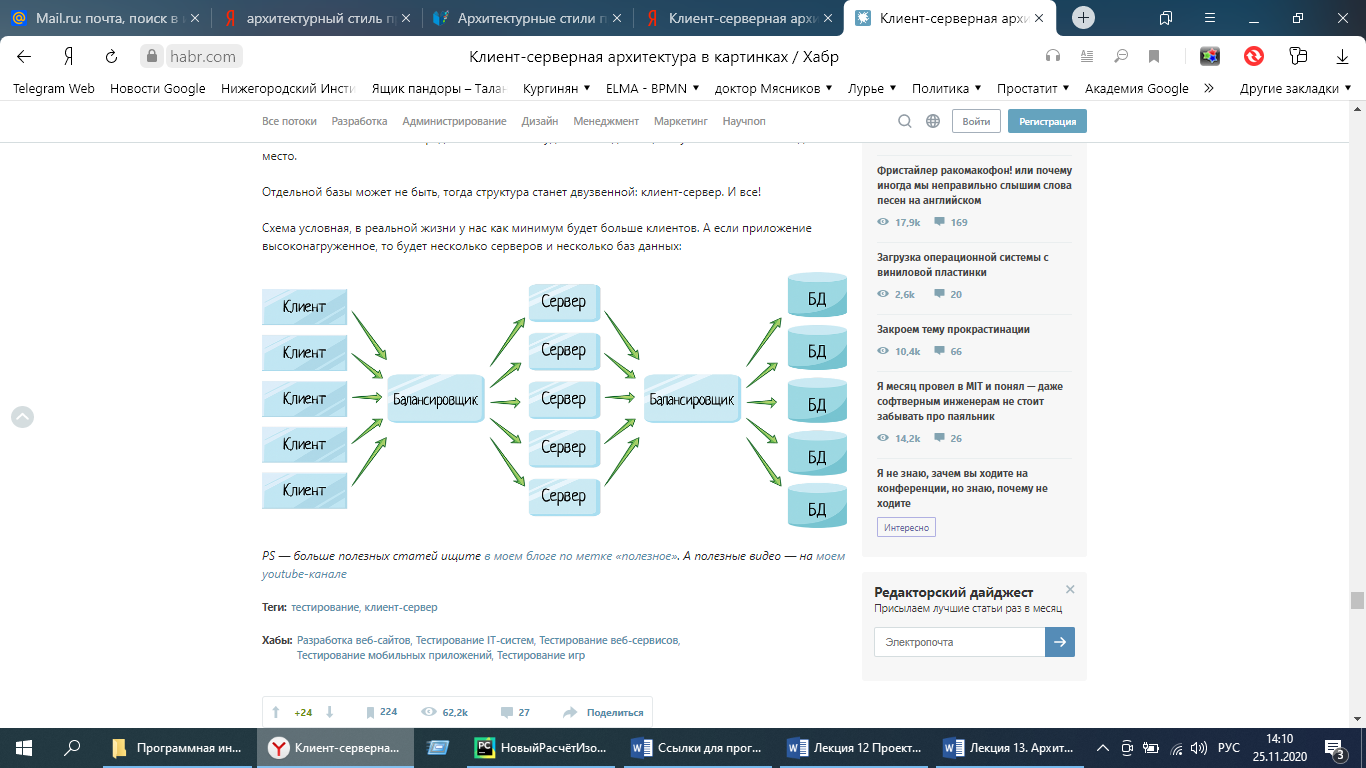
Сервера «сообщают» о числе запросов в буфере очереди, и балансировщик направляет запрос с меньшей очередью.



Facebook, amazon, google — туда заходят миллионы пользователей. Один сервер с ними не справится. Поэтому ставят кластер, а балансировщик делит между ними нагрузку. И в таком случае в кластере может быть не 2 сервера, а 10, 15, сколько нужно, столько и ставим.



Балансировка баз данных – возможна и необходима!



**2.2.2 Компонентная архитектура**

Дизайн приложения разлагается на функциональные (логические) компоненты, предоставляющие тщательно проработанные интерфейсы связи, с возможностью их повторного использования.

**2.2.3 Проблемно-ориентированное проектирование (дизайн на основе предметной области)**

Объектно-ориентированный стиль, ориентированный на моделирование сферы деловой активности и определяющий бизнес-объекты на основании сущностей данной предметной области.

**2.2.4 Многослойная архитектура**

Функциональные области приложения разделяются на многослойные группы (уровни).

**2.2.5 Архитектура на основе канала сообщений**

Стиль, предписывающий использование ПС, которая может принимать и отправлять сообщения по одному или более каналу связи. Приложения получают возможность взаимодействия, не располагая конкретными сведениями друг о друге.

**2.2.6 N-уровневая/З-уровневая**

Функциональность выделяется в отдельные сегменты во многом аналогично многослойному стилю, но сегменты физически располагаются на разных компьютерах (уровнях)

**2.2.7 Объектно-ориентированная архитектура**

Парадигма проектирования, основанная на распределении ответственности приложения (системы) между отдельными, многократно используемыми и самостоятельными объектами, содержащими данные и правила поведения

**2.2.8 Сервисно-ориентированная архитектура (SОА)**

Описывает приложения, предоставляющие и потребляющие функциональность в виде сервисов с помощью контрактов и сообщений

Полезная ссылка:

[www.baint.ru](http://www.baint.ru)