ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет прикладной математики и механики

Кафедра вычислительной математики и механики

Реферат

Тема: “Разработка высоконагруженных веб-приложений”

Группа: ИТСИ-17-1м

Студент: Ермакова О.А.

Преподаватель: Истомин Д.А.

Пермь 2018

Содержание

[*Часть 1* 3](#_Toc533596852)

[*1.* *Оптимизации БД (партиционирование, репликации, шардинг)* 3](#_Toc533596853)

[*2.* *Key-value хранилища (NOSQL, Redis и т.д.)* 6](#_Toc533596858)

[*3.* *Кэширование (memcached)* 7](#_Toc533596859)

[*4.* *Обзор технологий и фреймворков для реализации высоконагруженных приложений* 9](#_Toc533596860)

[*5.* *Сервис-ориентированная архитектура, микросервисы, оркестрация сервисов* 10](#_Toc533596861)

[*6.* *Шины ESB, очереди* 11](#_Toc533596862)

[*7.* *Тестирование нагрузки (подходы и инструменты)* 13](#_Toc533596863)

[*8.* *Мониторинг и логгирование (Zabbix, ELK Stack и др.)* 15](#_Toc533596866)

[*9.* *Оптимизация сети, протоколов взаимодействие, сетевой подсистемы ОС* 16](#_Toc533596868)

[*10.* *Балансировка и маршрутизация* 16](#_Toc533596869)

[*11.* *Веб серверы (например, оптимизация nginx)* 17](#_Toc533596870)

[*12.* *DDOS: как организовать и как с ним бороться* 17](#_Toc533596871)

[*13.* *CDN: виды, провайдеры, как использовать* 17](#_Toc533596872)

[*Часть 2* 18](#_Toc533596873)

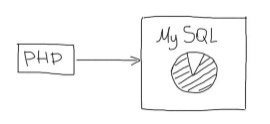
[*1.* *Файловые системы и организация хранилища (raid, zfs)* 18](#_Toc533596874)

# Часть 1

# Оптимизации БД (партиционирование, репликации, шардинг)

**Партиционирование (partitioning) —** это разбиение больших таблиц на логические части по выбранным критериям.

Масштабирование баз данных — самая сложная задача во время роста проекта. 90% всех усилий обычно приходится как раз на работу, связанную с ростом объема данных и операций с ними. Классическая схема работы приложения с базой данных выглядит так:



Один сервер базы данных в какой-то момент перестает справляться с нагрузкой. В этот момент и следует применять описанные тут техники масштабирования.

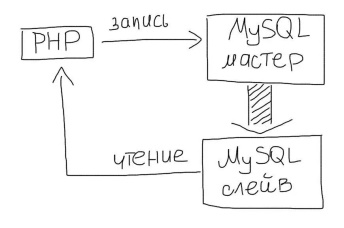
Перед тем, как приступать к масштабированию, необходимо провести[анализ медленных запросов](https://ruhighload.com/%d0%9f%d1%80%d0%be%d1%84%d0%b8%d0%bb%d0%b8%d1%80%d0%be%d0%b2%d0%b0%d0%bd%d0%b8%d0%b5+%d0%b2+mysql) и убедиться, что сервер [MySQL настроен оптимально](https://ruhighload.com/%d0%9e%d0%bf%d1%82%d0%b8%d0%bc%d0%b0%d0%bb%d1%8c%d0%bd%d0%b0%d1%8f+%d0%bd%d0%b0%d1%81%d1%82%d1%80%d0%be%d0%b9%d0%ba%d0%b0+mysql" \o "Оптимальная настройка Mysql" \t "_blank).

Стратегии

В основе масштабирования данных лежит тот же принцип, что и в основе масштабирования Web приложений. Это разделение данных на группы и выделение их на отдельные сервера. Существует две основные стратегии — репликация и шардинг.

Репликация

Репликация позволяет создать полный дубликат базы данных. Так, вместо одного сервера у Вас их будет несколько:



**Master-slave**

Чаще всего используют схему master-slave:

* **Master** — это основной сервер БД, куда поступают все данные. Все изменения в данных (добавление, обновление, удаление) должны происходить на этом сервере.
* **Slave** — это вспомогательный сервер БД, который копирует все данные с мастера. С этого сервера следует читать данные. Таких серверов может быть несколько.

Репликация позволяет использовать два или больше одинаковых серверов вместо одного. Операций чтения (SELECT) данных часто намного больше, чем операций изменения данных (INSERT/UPDATE). Поэтому, репликация позволяет разгрузить основной сервер за счет переноса операций чтения на слейв.

Работа из приложения

В приложении у Вас будет два соединения с базой данных. Одно — для мастера и одно для слейва:

<?

$master = mysql\_connect('10.10.0.1', 'root', 'pwd');

$slave = mysql\_connect('10.10.0.2', 'root', 'pwd');

# какой-то код и все такое...

$q = mysql\_query('INSERT INTO users ...', $master);

# еще какой-то код...

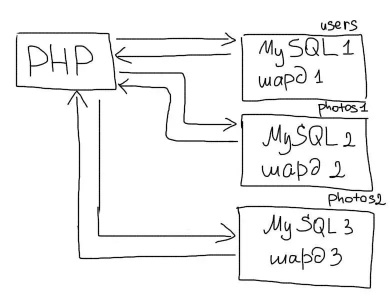
$q = mysql\_query('SELECT \* FROM users WHERE...', $slave);

Репликация обычно поддерживается самой СУБД (например, MySQL) и настраивается независимо от приложения. Читайте детальнее про настройку, использование и типы репликации данных на примере MySQL.

**Следует отметить**, что репликация сама по себе не очень удобный механизм масштабирования. Причиной тому — рассинхронизация данных и задержки в копировании с мастера на слейв. Зато это отличное средство для обеспечения отказоустойчивости. Вы всегда можете переключиться на слейв, если мастер ломается и наоборот. Чаще всего репликация используется совместно с шардингом именно из соображений надежности.

**Шардинг (sharding)**

Шардинг (иногда шардирование) — это другая техника масштабирования работы с данными. Суть его в разделении (партиционирование) базы данных на отдельные части так, чтобы каждую из них можно было вынести на отдельный сервер. Этот процесс зависит от структуры Вашей базы данных и выполняется прямо в приложении в отличие от репликации:



**Вертикальный шардинг**

Вертикальный шардинг — это выделение таблицы или группы таблиц на отдельный сервер. Например, в приложении есть такие таблицы:

* **users** — данные пользователей
* **photos** — фотографии пользователей
* **albums** — альбомы пользователей

Таблицу **users** Вы оставляете на одном сервере, а таблицы **photos** и **albums** переносите на другой. В таком случае в приложении Вам необходимо будет использовать соответствующее соединение для работы с каждой таблицей:

<?

$users\_connection = mysql\_connect('10.10.0.1', 'root', 'pwd');

$photos\_connection = mysql\_connect('10.10.0.2', 'root', 'pwd';

# какой-то код и все такое...

$q = mysql\_query('SELECT \* FROM users WHERE ...', $users\_connection);

# еще какой-то код...

$q = mysql\_query('SELECT \* FROM photos WHERE...', $photos\_connection);

# еще какой-то код...

$q = mysql\_query('SELECT \* FROM albums WHERE...', $photos\_connection);

В отличие от репликации, мы используем разные соединения для *любых* операций, но с *определенными* таблицами.

**Горизонтальный шардинг**

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

* 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.

Допустим, наше приложение работает с огромной таблицей, которая хранит фотографии пользователей. Мы подготовили два сервера (обычно они называются **шардами**) для этой таблицы. Для нечетных пользователей мы будем работать с первыми сервером, а для четных — со вторым. Таким образом, на каждом из серверов будет только часть всех данных о фотках пользователей. Это будет выглядеть так:

<?

# список соединений для таблицы с фотками

$photo\_connections = [

'1' => '10.10.0.1',

'2' => '10.10.0.2',

];

$user\_id = $\_SESSION['user\_id'];

# получение фотографий для пользователя $user\_id

$connection\_num = $user\_id % 2 == 0 ? 1 : 2;

$connection = mysql\_connect($photo\_connections[$connection\_num], 'root', 'pwd');

$q = mysql\_query('SELECT \* FROM photos WHREE user\_id = ' . intval($user\_id), $connection);

Результат вот этой операции **$user\_id % 2** будет остатком от деления на 2. Т.е. для четных чисел — 0, а для нечетных — 1.

Любая работа с таблицей **photos** теперь будет происходить только после получения нужного соединения на основе **$user\_id**.

Горизонтальный шардинг — это очень мощный инструмент масштабирования данных. Но в то же время и очень нетривиальный. Читайте детально об использовании горизонтального шардинга на практике.

Не следует применять технику шардинга ко всем таблицам. Правильный подход — это поэтапный процесс разделения растущих таблиц. Следует задумываться о горизонтальном шардинге, когда количество записей в одной таблице переходит за пределы от нескольких десятков миллионов до сотен миллионов.

**Совместное использование**

Шардинг и репликация часто используются совместно. В нашем примере, мы могли бы использовать по два сервера на каждый шард таблицы:

* **photos\_master\_1** — мастер первой половины таблицы.
* **photos\_slave\_1** — слейв первой половины таблицы.
* **photos\_master\_2** — мастер второй половины таблицы.
* **photos\_slave\_2** — слейв второй половины таблицы.

Тогда в приложении работа с этой табличкой может выглядеть так:

<?

# список соединений, для каждого шарда - мастер и слейв

$photo\_connections = [

'1' => [

'master' => '10.10.0.10',

'slave' => '10.10.0.11',

],

'2' => [

'master' => '10.10.0.20',

'slave' => '10.10.0.21',

],

];

$user\_id = $\_SESSION['user\_id'];

# Читаем данные со слейвов

$connection\_num = $user\_id % 2 == 0 ? 1 : 2;

$connection = mysql\_connect($photo\_connections[$connection\_num]['slave'], 'root', 'pwd');

$q = mysql\_query('SELECT \* FROM photos WHREE user\_id = ' . intval($user\_id), $connection);

# какой-то код...

# Изменение данных происходит на мастерах

$photo\_id = 7;

$connection\_num = $user\_id % 2 == 0 ? 1 : 2;

$connection = mysql\_connect($photo\_connections[$connection\_num]['master'], 'root', 'pwd');

$q = mysql\_query('UPDATE photos SET views = views + 1 WHREE photo\_id = ' . intval($photo\_id), $connection);

Такая схема часто используется не для масштабирования, а для обеспечения отказоустойчивости. Так, при выходе из строя одного из серверов шарда, всегда будет запасной.

**Key-value базы данных**

Следует отметить, что большинство Key-value баз данных поддерживает шардинг на уровне платформы. Например, Memcache. В таком случае, Вы просто указываете набор серверов для соединения, а платформа сделает все остальное:

<?

$m = new Memcache;

$m->addServer('10.5.0.1');

$m->addServer('10.5.0.2');

$m->addServer('10.5.0.3');

...

$m->get('user1')

Самое важное

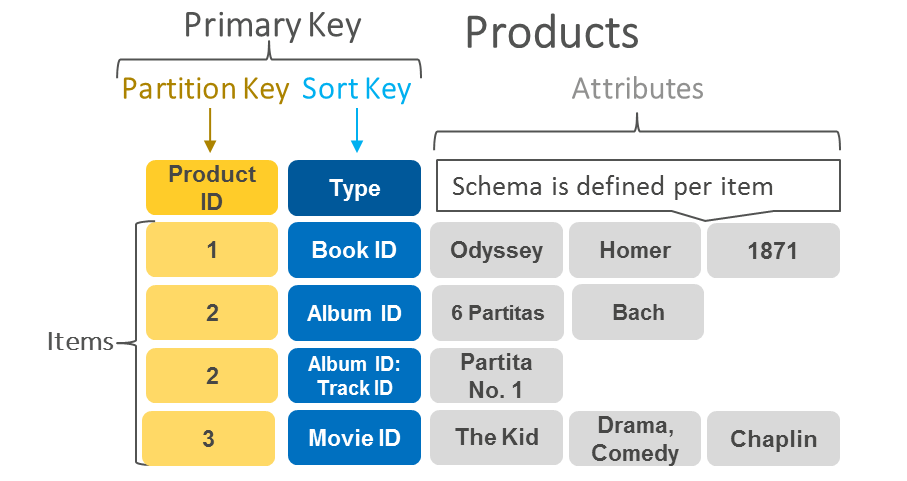
Шардинг и репликация — это популярные и мощные техники масштабирования систем работы с данными. Несмотря на примеры для MySQL, эти подходы универсальны и могут применяться для любой технологии.

**Помните**, процесс масштабирования данных — это архитектурное решение, оно не связано с конкретной технологией. Не делайте ошибок наших отцов — не переезжайте с известной Вам технологии на новую из-за поддержки или не поддержки шардинга. Проблемы обычно связаны с архитектурой, а не конкретной базой данных.

# Key-value хранилища (NOSQL, Redis и т.д.)

База данных на основе пар «ключ‑значение» – это тип нереляционных баз данных, в котором для хранения данных используется простой метод «ключ‑значение». База данных на основе пар «ключ‑значение» хранит данные как совокупность пар «ключ‑значение», в которых ключ служит уникальным идентификатором. Как ключи, так и значения могут представлять собой что угодно: от простых до сложных составных объектов. Базы данных с использованием пар «ключ‑значение» поддерживают высокую разделяемость и обеспечивают беспрецедентное горизонтальное масштабирование, недостижимое при использовании других типов баз данных. Например, [Amazon DynamoDB](https://aws.amazon.com/dynamodb/" \t "_blank) выделяет дополнительные разделы на таблицу, если существующий раздел заполняется до предела и требуется больше пространства для хранения.

На следующей диаграмме показан пример данных, хранящихся в виде пар «ключ‑значение» в DynamoDB.



**NoSQL** (от [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) *not only SQL* — *не только SQL*) — термин, обозначающий ряд подходов, направленных на реализацию хранилищ баз данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных СУБДс доступом к данным средствами языка SQL. Применяется к базам данных, в которых делается попытка решить проблемы масштабируемости и доступности за счёт атомарности и согласованности данных.

Решения NoSQL отличаются не только проектированием с учётом масштабирования. Другими характерными чертами NoSQL-решений являются:

* Применение различных типов хранилищ.
* Возможность разработки базы данных без задания [схемы](https://ru.wikipedia.org/wiki/%D0%A1%D1%85%D0%B5%D0%BC%D0%B0_%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85).
* Линейная масштабируемость (добавление процессоров увеличивает производительность).
* Инновационность: «не только SQL» открывает много возможностей для хранения и обработки данных.

**Redis** ([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) ***re****mote****di****ctionary****s****erver*) — сетевое журналируемое хранилище данных типа «[ключ — значение](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D1%8E%D1%87_%E2%80%94_%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)» с открытым исходным кодом. Нереляционная высокопроизводительная [СУБД](https://ru.wikipedia.org/wiki/%D0%A1%D0%A3%D0%91%D0%94).

Redis поддерживает репликацию типа master-slave. Данные с любого сервера Redis могут реплицироваться произвольное количество раз. Репликация полезна для масштабирования чтения (но не записи) или при очень больших объёмах данных. Все данные, которые попадают на один узел Redis (который называется master), будут попадать также на другие узлы (называются slave). Для конфигурирования slave-узлов можно изменить опцию slaveof или аналогичную по написанию команду (узлы, запущенные без подобных опций, являются master-узлами).

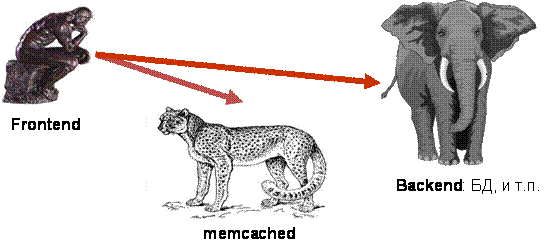
Репликация помогает защитить данные, копируя их на другие сервера. Репликация также может быть использована для увеличения производительности, так как запросы на чтение могут обслуживаться slave-узлами. Эти узлы могут ответить слегка устаревшими данными, но для большинства приложений это приемлемо.

Система репликации Redis ещё не поддерживает автоматическую отказоустойчивость. Если master-узел выходит из строя, необходимо вручную выбрать новый master из списка slave-узлов. Необходимо использовать Redis Sentinel для мониторинга и автоматического переключения master-узлов, если необходима устойчивая к сбоям система.

# Кэширование (memcached)

[Memcached](http://danga.com/memcached/) представляет собой огромную хэш-таблицу в оперативной памяти, доступную по сетевому протоколу. Он обеспечивает сервис по хранению значений, ассоциированных с ключами. Доступ к хэшу мы получаем через простой сетевой протокол, клиентом может выступать программа, написанная на произвольном языке программирования (существуют клиенты для C/C++, PHP, Perl, Java и т.п.).  
  
Самые простые операции – получить значение указанного ключа (get), установить значение ключа (set) и удалить ключ (del). Для реализации цепочки атомарных операций (при условии конкурентного доступа к memcached со стороны параллельных процессов) используются дополнительные операции: инкремент/декремент значения ключа (incr/decr), дописать данные к значению ключа в начало или в конец (append/prepend), атомарная связка получения/установки значения (gets/cas) и другие.  
  
Memcached был реализован Брэдом Фитцпатриком (Brad Fitzpatrick) в рамках работы над проектом ЖЖ (LiveJournal). Он использовался для разгрузки базы данных от запросов при отдаче контента страниц. Сегодня memcached нашел своё применение в ядре многих крупных проектов, например, Wikipedia, YouTube, Facebook и другие.

Общая схема кэширования

  
  
В общем случае схема кэширования выглядит следующим образом: frontend’у (той части проекта, которая формирует ответ пользователю) требуется получить данные какой-то выборки. Frontend обращается к быстрому как гепард серверу memcached за кэшом выборки (get-запрос). Если соответствующий ключ будет обнаружен, работа на этом заканчивается. В противном случае следует обращение к тяжелому, неповоротливому, но мощному (как слон) backend’у, в роли которого чаще всего выступает база данных. Полученный результат сразу же записывается в memcached в качестве кэша (set-запрос). При этом обычно для ключа задается максимальное время жизни (срок годности), который соответствует моменту сброса кэша.  
  
Такая стандартная схема кэширования реализуется всегда. Вместо memcached в некоторых проектах могут использоваться локальные файлы, иные способы хранения (другая БД, кэш PHP-акселератора и т.п.) Однако, как будет показано далее, в высоконагруженном проекте данная схема может работать не самым эффективным образом. Тем не менее, в нашем дальнейшем рассказе мы будем опираться именно на эту схему.

Архитектура memcached

Каким же образом устроен memcached? Как ему удаётся работать настолько быстро, что даже десятки запросов к memcached, необходимых для обработки одной страницы сайта, не приводят к существенной задержке. При этом memcached крайне нетребователен к вычислительным ресурсам: на нагруженной инсталляции процессорное время, использованное им, редко превышает 10%.  
  
Во-первых, memcached спроектирован так, чтобы все его операции имели алгоритмическую сложность O(1), т.е. время выполнения любой операции не зависит от количества ключей, которые хранит memcached. Это означает, что некоторые операции (или возможности) будут отсутствовать в нём, если их реализация требует всего лишь линейного (O(n)) времени. Так, в memcached отсутствуют возможность объединения ключей «в папки», т.е. какой-либо группировки ключей, также мы не найдем групповых операций над ключами или их значениями.  
  
Основными оптимизированными операциями является выделение/освобождение блоков памяти под хранение ключей, определение политики самых неиспользуемых ключей (LRU) для очистки кэша при нехватке памяти. Поиск ключей происходит через хэширование, поэтому имеет сложность O(1).  
  
Используется асинхронный ввод-вывод, не используются нити, что обеспечивает дополнительный прирост производительности и меньшие требования к ресурсам. На самом деле memcached может использовать нити, но это необходимо лишь для использования всех доступных на сервере ядер или процессоров в случае слишком большой нагрузки – на каждое соединение нить не создается в любом случае.  
  
По сути, можно сказать, что время отклика сервера memcached определяется только сетевыми издержками и практически равно времени передачи пакета от frontend’а до сервера memcached (RTT). Такие характеристики позволяют использовать memcached в высоконагруженных web-проектов для решения различных задач, в том числе и для кэширования данных.

Потеря ключей

Memcached не является надежным хранилищем – возможна ситуация, когда ключ будет удален из кэша раньше окончания его срока жизни. Архитектура проекта должна быть готова к такой ситуации и должна гибко реагировать на потерю ключей. Можно выделить три основных причины потери ключей:

1. Ключ был удален раньше окончания его срока годности в силу нехватки памяти под хранение значений других ключей. Memcached использует политику LRU, поэтому такая потеря означает, что данный ключ редко использовался и память кэша освобождается для хранения более популярных ключей.
2. Ключ был удален, так как истекло его время жизни. Такая ситуация строго говоря не является потерей, так как мы сами ограничили время жизни ключа, но для клиентского по отношению к memcached кода такая потеря неотличима от других случаев – при обращении к memcached мы получаем ответ «такого ключа нет».
3. Самой неприятной ситуацией является крах процесса memcached или сервера, на котором он расположен. В этой ситуации мы теряем все ключи, которые хранились в кэше. Несколько сгладить последствия позволяет кластерная организация: множество серверов memcached, по которым «размазаны» ключи проекта: так последствия краха одного кэша будут менее заметны.

Все описанные ситуации необходимо иметь в виду при разработке программного обеспечения, работающего с memcached. Можно разделить данные, которые мы храним в memcached, по степени критичности их потери.  
  
**«Можно потерять»**. К этой категории относятся кэши выборок из базы данных. Потеря таких ключей не так страшна, потому что мы можем легко восстановить их значения, обратившись заново к backend’у. Однако частые потери кэшей приводят к излишним обращениям к БД.  
  
**«Не хотелось бы потерять»**. Здесь можно упомянуть счетчики посетителей сайта, просмотров ресурсов и т.п. Хоть и восстановить эти значения иногда напрямую невозможно, но значения этих ключей имеют ограниченный по времени смысл: через несколько минут их значение уже неактуально, и будет рассчитано новое значение.  
  
**«Совсем не должны терять»**. Memcached удобен для хранения сессий пользователей – все сессии равнодоступны со всех серверов, входящих в кластер frontend’ов. Так вот содержимое сессий не хотелось бы терять никогда – иначе пользователей на сайте будет «разлогинивать». Как попытаться избежать? Можно дублировать ключи сессий на нескольких серверах memcached из кластера, так вероятность потери снижается.

# Обзор технологий и фреймворков для реализации высоконагруженных приложений

**Важные критерии при выборе технологий:** размер и тип проекта, сложность проекта, скорость разработки, стоимость специалистов, доступность специалистов, доступные инструменты разработки, наличие готовых решений, гибкость решения, наличие широкого сообщества, отказоустойчивость решения, тренд его развития, наличие подробной документации, стоимость поддержки, требования к нагрузкам, требования к безопасности, кроссплатформенность, возможности интеграции с другими решениями.

##### **Языки программирования**

Краткая характеристика популярных языков:

* PHP — его используют в основном для простых и средних проектов. Очень много коробочных решений. Относительно дешевые программисты. Антитренд последних лет, хотя с выходом последней версии языка под номером 7, он получил действительно мощные возможности.
* Python — современный язык, разработка на нем быстрая и качественная. Используют его для средних и больших проектов. Программистов найти проблематично, и стоят они не дешево.
* Ruby — современный язык, разработка на нем также быстрая. Его используют в основном для разработки простых и средних проектов, часто разрабатывают стартапы. Программистов также мало, и они дорогие.
* Java — разработка на нем очень долгая и дорогая. Его используют в основном для больших проектов со специфическими требованиями.
* C# — аналог Java, также используют для больших проектов, часть в сфере FinTech.
* JS — очень быстро развивается, тренд последних лет. Огромное количество наработок, и можно писать все, что угодно, даже игры. Его используют для средних и больших проектов, но действительно мощные возможности этот язык получил недавно, потому примеров больших проектов пока мало, специалисты самые дорогие, и найти их сложнее всего.

**Популярные фреймворки и платформы:**

Это некая среда разработки для программистов, где есть готовая инфраструктура и ряд готовых функций со стандартными решениями типичных задач. На каждом языке есть много разных фреймворков. Есть как общие, которые создавались для разработки любых решений, так и специализированные, под узкие задачи. Также есть те, на которых делаются большие и сложные решения, а другие для этого не предназначены. Ниже я опишу популярные фреймворки для каждого из языков, на которых можно писать большие и сложные решения.

На фреймворках разрабатываются довольно большие и сложные сайты с уникальным функционалом. Это значительно быстрее и дешевле, чем на чистом языке, но при этом такое решение позволяет разрабатывать действительно сложные вещи и оптимизировать все это под нагрузки.

* PHP: Symfony, Laravel
* Python: Django
* Ruby: Ruby On Rails
* Java: Spring
* C#: .NET
* JS: Node.js, AngularJS

**Стек технологий в больших проектах**

Выбор технологий зависит от предлагаемой архитектуры проекта. Именно архитектор продумывает основные блоки будущего сайта. Какой язык ляжет к основу, будет ли он нативным или фреймворк, какую систему кэширования выбрать, какие базы данных, как все это будет связано и т.д.

Для примера рассмотрим технологии Instagram (данные Insight IT):

* Ubuntu Server 14.04 LTS — основная серверная операционная система
* Python — основной язык программирования серверной части
* Django — фреймворк
* nginx — второй уровень балансировки входящих HTTP-запросов
* gunicorn — WSGI-сервер
* HAProxy — балансировка нагрузки внутри системы
* PostgreSQL — основное хранилище данных
* postgis — поддержка гео-запросов
* pgfouine — отчеты на основе логов
* pgbouncer — создание пула соединений
* Redis — дополнительное хранилище данных
* Memcached — кэширование
* Gearman — очередь задач
* Solr — гео-поиск
* munin, statsd, pingdom — мониторинг
* Fabric — управление кластером
* xfs — файловая система

# Сервис-ориентированная архитектура, микросервисы, оркестрация сервисов

**Се́рвис-ориенти́рованная архитекту́ра** (*SOA*, [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) *service-oriented architecture*) — [модульный](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) подход к разработке программного обеспечения, основанный на использовании [распределённых](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), [слабо связанных](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%BB%D0%B0%D0%B1%D0%BE%D0%B5_%D1%81%D0%B2%D1%8F%D0%B7%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) ([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) *[loose coupling](https://en.wikipedia.org/wiki/loose_coupling" \o "en:loose coupling)*) заменяемых компонентов, оснащённых стандартизированными [интерфейсами](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) для взаимодействия по стандартизированным [протоколам](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB).

Программные комплексы, разработанные в соответствии с сервис-ориентированной архитектурой, обычно реализуются как набор [веб-служб](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%BB%D1%83%D0%B6%D0%B1%D0%B0), взаимодействующих по протоколу [SOAP](https://ru.wikipedia.org/wiki/SOAP), но существуют и другие реализации (например, на базе [jini](https://ru.wikipedia.org/wiki/Jini" \o "Jini), [CORBA](https://ru.wikipedia.org/wiki/CORBA), на основе [REST](https://ru.wikipedia.org/wiki/REST)).

Интерфейсы компонентов в сервис-ориентированной архитектуре [инкапсулируют](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) детали реализации ([операционную систему](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), [платформу](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%BE%D1%81%D1%81%D0%BF%D0%BB%D0%B0%D1%82%D1%84%D0%BE%D1%80%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D1%8C), [язык программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) от остальных компонентов, таким образом обеспечивая комбинирование и [многократное использование](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0) компонентов для построения сложных [распределённых программных комплексов](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), обеспечивая независимость от используемых [платформ](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BB%D0%B0%D1%82%D1%84%D0%BE%D1%80%D0%BC%D0%B0) и инструментов разработки, способствуя [масштабируемости](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D0%BE%D1%81%D1%82%D1%8C) и [управляемости](https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%8F%D0%B5%D0%BC%D0%BE%D1%81%D1%82%D1%8C_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)) создаваемых систем.

Микросервисная архитектура — это подход к созданию приложения, подразумевающий отказ от единой, монолитной структуры. То есть вместо того чтобы исполнять все ограниченные контексты приложения на сервере с помощью внутрипроцессных взаимодействий, мы используем несколько небольших приложений, каждое из которых соответствует какому-то ограниченному контексту. Причём эти приложения работают на разных серверах и взаимодействуют друг с другом по сети, например посредством HTTP.

|  |  |
| --- | --- |
| **Преимущества** | |
| **Монолитная архитектура** | **Микросервисы** |
| **Простота**  Монолитная архитектура гораздо проще в реализации, управлении и развёртывании.  Микросервисы требуют тщательного управления, поскольку они развёртываются на разных серверах и используют API. | **Частичное развёртывание**   Микросервисы позволяют по мере необходимости обновлять приложение по частям.  При единой архитектуре нам приходится заново развёртывать приложение целиком, что влечёт за собой куда больше рисков. |
| **Согласованность (Consistency)**  При монолитной архитектуре проще поддерживать согласованность кода, обрабатывать ошибки и т. д. Зато микросервисы могут полностью управляться разными командами с соблюдением разных стандартов. | **Доступность**  У микросервисов доступность выше: даже если один из них сбоит, это не приводит к сбою всего приложения. |
| **Межмодульный рефакторинг**  Единая архитектура облегчает работу в ситуациях, когда несколько модулей должны взаимодействовать между собой или когда мы хотим переместить классы из одного модуля в другой. В случае с микросервисами мы должны очень чётко определять границы модулей! | **Сохранение модульности**  Сохранять модульность и инкапсуляцию может быть непросто, несмотря на правила SOLID. Однако микросервисы позволяют гарантировать отсутствие общих состояний (shared state) между модулями. |
|  | **Мультиплатформенность/гетерогенность**  Микросервисы позволяют использовать разные технологии и языки, в соответствии с вашими задачами. |

# Шины ESB, очереди

ESB (Enterprise Service Bus) — дословно можно перевести как «сервисная шина предприятия». ESB описывает вполне реальный программный продукт, в задачи которого входит упрощение вызова службы за счет управления всеми взаимодействиями на пути от потребителя службы к поставщику и обратно. Двумя наиболее часто упоминаемыми возможностями ESB являются преобразование сообщений и их маршрутизация. На шину ESB в архитектуре [SOA](http://www.tadviser.ru/index.php/SOA) возложена важнейшая задача обеспечения взаимодействия систем из слабосвязанных сервисов в сети. Аналитики [Gartner](http://www.tadviser.ru/index.php/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B0%D0%BD%D0%B8%D1%8F:Gartner" \o "Gartner) определяют ESB как новый тип программного обеспечения промежуточного уровня, который объединяет функциональные возможности других уже существующих типов промежуточного обеспечения. Корпоративная сервисная шина поддерживает Web-сервисы, реализуя протокол [SOAP](http://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:SOAP) (Simple Object Access Protocol, Простой протокол доступа к объектам) и используя язык [WSDL](http://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:WSDL) (Web Services Description Language, Язык описания Web-сервисов) и спецификацию [UDDI](http://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:UDDI) (Universal Description, Discovery and Integration, Универсальное описание, обнаружение и интеграция).

**Архитектура ESB**

Основа архитектуры ESB — это идея использования общей интеграционной инфраструктуры всеми корпоративными приложениями на базе обмена сообщениями. Все приложения взаимодействуют через одну точку, которая, в случае необходимости, обеспечивает сохранность обращений, преобразование данных и транзакции. При этом целью интеграции приложения является создание единственного модуля (или адаптера), который отвечает за «подключение» приложения к ESB. Последующую обработку сообщений и их маршрутизацию в другие системы, ESB выполняет на основании установленных бизнес-правил самостоятельно. Этот подход обеспечивает превосходную гибкость, простоту масштабирования и переноса, поэтому в случае замены одного из приложений подключенного к шине, перенастраивать остальные не нужно.

**Достоинства и недостатки**

Достоинствами ESB является:

* Организация размещения существующих систем осуществляется быстрее и дешевле.
* Повышение гибкости.
* ESB основывается на общепризнанных стандартах.
* Наличие большого количества конфигурации для интеграции.

К числу недостатков ESB относят:

* Сложность реализации
* Требует больших ресурсов.
* Разработка Enterprise Service Bus

Отличительной чертой Web-служб является то, что потребитель имеет возможность динамически связываться с провайдером службы.

Все это происходит благодаря двум главным функциональным возможностям:

* **Обнаруживаемость.** В автоматически поддерживаемые каталоги могут быть собраны провайдеры Web-служб. Таким образом, потребителю предоставляется возможность просматривать такой каталог, чтобы найти провайдера нужной службы.
* **Самоописание.** Web-служба способна описывать себя удобным для машинного чтения способом. Так, можно распознать двух или более провайдеров одной и той же службы, даже если они выполнены совершенно по-разному, так как их описательные интерфейсы отвечают одному и тому же описанию.

Такие функциональные способности Web-служб кардинально отличаются от существовавших подходов к интеграции, в которых интерфейсы определялись во время компиляции и во время связи потребителя с провайдерами. Форматы сообщений выражались не описательно .Они не могли заставить следовать этому формату поэтому были основаны на внутренней договоренности, в результате чего получатель не мог обработать структуру, созданную отправителем.

Самоописание Web-служб облегчило интеграцию с помощью объявления интерфейсов, которым нужно было следовать. Благодаря динамическому обнаружению службы, потребитель был освобожден от зависимости от конкретного провайдера с определенным адресом, однако, и эта возможность создала свои собственные проблемы. Достаточно тяжело было решить вопрос связи потребителя с провайдером раз и навсегда во время компиляции и потенциальным поиском нового провайдера при каждом вызове во время исполнения. Тогда ESB предложила другой вариант — дать возможность потребителю один раз динамически связаться с прокси-службой, имея при этом возможность использовать нескольких провайдеров и провайдеров, которые могут появиться в будущем, через эту прокси-службу. Все это говорит о том, что ESB делает службы доступными для их вызова потребителями и дает возможность потребителям найти службы программным способом.

**Шлюз служб**

Так называемый шлюз служб является краеугольным камнем синхронной ESB. Он выступает посредником между провайдерами и потребителями служб, оказывая при этом содействие в обработке синхронных вызовов с использованием брокера. Данный шлюз открывает доступ ко всем известным службам и прокси-службам каждой из них, поэтому он является своего рода «единым окном» для потребителя, желающего вызывать любую службу у любого провайдера, который известен шлюзу. Когда Web-службы координирует шлюз, они обладают способностью к самоописанию.

Каждая служба предоставляет свой интерфейс с помощью WSDL, который состоит из следующих четырех частей:

* Типы портов — набор операций, которые выполняет Web-служба.
* Сообщения — то есть формат запросов и ответов
* Типы — Типы данных, которые используются Web-службой
* Связи — адрес для вызова операции

Web-службы шлюза, а точнее их прокси-службы являются также обнаруживаемыми. Шлюз дает такую возможность в виде [UDDI](http://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:UDDI)-службы. Чтобы найти адрес вызова службы потребитель подает запрос в UDDI-службу шлюза список провайдеров необходимой [WSDL](http://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:WSDL)-операции и получает обратно адрес прокси-службы шлюза для этой операции.

**Шина сообщений**

Шаблон Message Bus (Шина сообщений) является основой асинхронной ESB. Шина сообщений — это набор каналов сообщений, которые настроены как пары каналов запрос-ответ. Каждая пара представляет службу, которую может вызвать потребитель, использующий шину. Потребитель посылает сообщение в очередь запросов службы и после этого выслушивает очередь ответов для получения результата. О том, кто предоставляет службу потребитель на самом деле не знает. Провайдеры служб также подсоединены к шине сообщений и прослушивают ее на наличие сообщений запросов. При наличии нескольких провайдеров службы, между ними происходит соревнование как между потребителями за получение конкретного запроса. Система сообщений, выполняемая шиной сообщений, функционирует как диспетчер сообщений и рассылает запросы провайдерам служб, оптимизируя эту рассылку в зависимости от степени нагрузки, сетевых задержек и т. д. После того как провайдер получил запрос, он выполняет службу и вносит результат в сообщение в очередь ответов, то есть провайдер и потребитель никогда не знают о месторасположении друг друга. Эта шина сообщений и является сущностью ESB.

# Тестирование нагрузки (подходы и инструменты)

**Нагрузочное тестирование** ([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) *load testing*) — подвид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

### Основные принципы нагрузочного тестирования

**1. Уникальность запросов**

*Даже сформировав реалистичный сценарий работы с системой на основе статистики её использования, необходимо понимать, что всегда найдутся исключения из этого сценария.*

В случае *Примера 1* это может быть пользователь, обращающийся к отличным от всех остальных, уникальным страницам веб-сервиса.

**2. Время отклика системы**

*В общем случае время отклика системы подчиняется*[*функции нормального распределения*](https://ru.wikipedia.org/wiki/%D0%9D%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5)*.*

В частности, это означает, что, имея достаточное количество измерений, можно определить вероятность с которой отклик системы на запрос попадёт в тот или иной интервал времени.

**3. Зависимость времени отклика системы от степени распределённости этой системы.**

[*Дисперсия*](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BF%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D0%BE%D0%B9_%D0%B2%D0%B5%D0%BB%D0%B8%D1%87%D0%B8%D0%BD%D1%8B)*нормального распределения времени отклика системы на запрос пропорциональна отношению количества узлов системы, параллельно обрабатывающих такие запросы и количеству запросов, приходящихся на каждый узел.*

То есть, на разброс значений времени отклика системы влияет одновременно количество запросов приходящихся на каждый узел системы и само количество узлов, каждый из которых добавляет некоторую случайную величину задержки при обработке запросов.

**4. Разброс времени отклика системы**

*Из утверждений 1, 2 и 3 можно также заключить, что при достаточно большом количестве измерений величины времени обработки запроса в любой системе всегда найдутся запросы, время обработки которых превышает определённые в требованиях максимумы; причем, чем больше суммарное время проведения эксперимента тем выше окажутся новые максимумы.*

Этот факт необходимо учитывать при формировании требований к производительности системы, а также при проведении регулярного нагрузочного тестирования.

**5. Точность воспроизведения профилей нагрузки**

*Необходимая точность воспроизведения профилей нагрузки тем дороже, чем больше компонент содержит система.*

Часто невозможно учесть все аспекты профиля нагрузки для сложных систем, так как чем сложнее система, тем больше времени будет затрачено на проектирование, программирование и поддержку адекватного профиля нагрузки для неё, что не всегда является необходимостью. Оптимальный подход в данном случае заключается в балансировании между стоимостью разработки теста и покрытием функциональности системы, в результате которого появляются допущения о влиянии на общую производительность той или иной части тестируемой системы.

### Инструментарий для тестирования производительности

Существуют различные инструменты для обнаружения и исследования проблем в различных узлах системы. Все узлы системы могут быть классифицированы следующим образом:

* Приложение,
* База данных,
* Сеть,
* Обработка на клиентской стороне,
* Балансировка нагрузки.

|  |  |  |
| --- | --- | --- |
| **ПО** | **Наименование производителя** | **Комментарии** |
| OpenSTA | 'Open System Testing Architecture' | Свободно распространяемое программное обеспечение для нагрузочного/стресс тестирования, лицензированное GNU GPL. Использует распределённую архитектуру приложений, основанную на [CORBA](https://ru.wikipedia.org/wiki/CORBA). Доступна версия под Windows, хотя имеются проблемы с совместимостью с Windows Vista. Поддержка прекращена в 2007 году. |
| IBM Rational Performance Tester | IBM | Основанное на среде разработки [Eclipse](https://ru.wikipedia.org/wiki/Eclipse_(%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8)" \o "Eclipse (среда разработки)) ПО, позволяющее создавать нагрузку больших объёмов и измерять время отклика для приложений с клиент-серверной архитектурой. Требует лицензирования. |
| [JMeter](https://ru.wikipedia.org/wiki/JMeter) | Открытый проект Apache Jakarta Project | Основанный на Java кроссплатформенный инструментарий, позволяющий производить нагрузочные тесты с использованием JDBC / FTP / LDAP / SOAP / JMS / POP3 / HTTP / TCP соединений. Даёт возможность создавать большое количество запросов с разных компьютеров и контролировать процесс с одного из них. |
| [HP LoadRunner](https://ru.wikipedia.org/wiki/HP_LoadRunner) | HP | Инструмент для нагрузочного тестирования, изначально разработанный для эмуляции работы большого количества параллельно работающих пользователей. Также может быть использован для [unit-](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \o "Модульное тестирование) или [интеграционного тестирования](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). |
| [LoadComplete](https://smartbear.com/product/loadcomplete/overview/) | SmartBear | Проприетарный продукт, позволяющий проводить нагрузочное тестирование веб-приложений |
| [SilkPerformer](https://ru.wikipedia.org/wiki/SilkPerformer) | Micro Focus (Borland) |  |
| Siege | Joe Dog Software | Siege — это утилита для нагрузочного тестирования веб-серверов.[[3]](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BE%D1%87%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-3) |
| [Visual Studio](https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio)Team System | [Microsoft](https://ru.wikipedia.org/wiki/Microsoft) | Visual Studio предоставляет инструмент для тестирования производительности включая load / unit testing |
| QTest | Quotium |  |
| HTTPerf |  |  |
| QALoad | Compuware Ltd. |  |
| (The) Grinder |  |  |
| WebLOAD | RadView Software | Нагрузочное тестирование инструмент для веб-и мобильных приложений, включая веб-панели для тестирования производительности анализа. Используется для крупномасштабных нагрузок, которые могут быть сгенерированы также из облака. лицензированный.[[4]](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BE%D1%87%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-4) |
| [Яндекс.Танк](https://ru.wikipedia.org/wiki/%D0%AF%D0%BD%D0%B4%D0%B5%D0%BA%D1%81.%D0%A2%D0%B0%D0%BD%D0%BA) | Яндекс | Модульный и расширяемый инструмент, позволяющий использовать внутри разные генераторы, в частности, знакомый многим JMeter. Это open-source проект, опубликованный Яндексом в 2012 году. |

# Мониторинг и логгирование (Zabbix, ELK Stack и др.)

**Zabbix** — [свободная](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) система [мониторинга](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%BD%D0%B8%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3) и отслеживания статусов разнообразных сервисов [компьютерной сети](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C), [серверов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%B0%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)) и [сетевого оборудования](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B5_%D0%BE%D0%B1%D0%BE%D1%80%D1%83%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

Для хранения данных используется [MySQL](https://ru.wikipedia.org/wiki/MySQL" \o "MySQL), [PostgreSQL](https://ru.wikipedia.org/wiki/PostgreSQL" \o "PostgreSQL), [SQLite](https://ru.wikipedia.org/wiki/SQLite" \o "SQLite) или [Oracle Database](https://ru.wikipedia.org/wiki/Oracle_Database" \o "Oracle Database), веб-интерфейс написан на [PHP](https://ru.wikipedia.org/wiki/PHP).

## Архитектура и возможности

Zabbix-сервер — ядро системы, которое дистанционно контролирует сетевые сервисы и является хранилищем, в котором содержатся все конфигурационные, статистические и оперативные данные. Он является тем субъектом в программном обеспечении Zabbix, который оповещает администраторов о проблемах с контролируемым оборудованием.

Zabbix-прокси собирает данные о производительности и доступности от имени Zabbix-сервера. Все собранные данные заносятся в буфер на локальном уровне и передаются Zabbix-серверу, к которому принадлежит прокси-сервер. Zabbix-прокси является идеальным решением для дистанционного контроля филиалов и других точек, в т.ч. сетей, не имеющих местных администраторов. Он может быть также использован для распределения нагрузки одного Zabbix-сервера. В этом случае, прокси только собирает данные, тем самым на сервер ложится меньшая нагрузка на ЦПУ и на устройства ввода/вывода.

Zabbix-агент — программа контроля локальных ресурсов и приложений (таких как накопители, оперативная память, статистика процессора и т.д.) на сетевых системах, эти системы должны работать с запущенным Zabbix-агентом.

Zabbix-агенты являются чрезвычайно эффективными из-за использования специфических системных вызовов для сбора информации и подготовки статистики.

Веб-интерфейс — часть Zabbix-сервера, и, как правило (но не обязательно), запускается на том же физическом узле, что и Zabbix-сервер. Работает на [PHP](https://ru.wikipedia.org/wiki/PHP), требует веб-сервер (например, [Apache httpd](https://ru.wikipedia.org/wiki/Apache_HTTP_Server" \o "Apache HTTP Server)).

Основные возможности:

* Распределённый мониторинг — до нескольких тысяч узлов. Конфигурация младших узлов полностью контролируется старшими узлами, находящимися на более высоком уровне иерархии.
* Сценарии на основе мониторинга
* Автоматическое обнаружение
* Централизованный мониторинг журналов
* Веб-интерфейс для администрирования и настройки
* Отчётность и тенденции
* [SLA](https://ru.wikipedia.org/wiki/SLA)-мониторинг
* Поддержка высокопроизводительных агентов (zabbix-agent) практически для всех платформ
* Комплексная реакция на события
* Поддержка [SNMP](https://ru.wikipedia.org/wiki/SNMP) v1, 2, 3
* Поддержка [SNMP-ловушек](https://ru.wikipedia.org/wiki/SNMP#Trap)
* Поддержка [IPMI](https://ru.wikipedia.org/wiki/IPMI)
* Поддержка мониторинга JMX-приложений
* Поддержка выполнения запросов в различные базы данных без необходимости использования сценарной обвязки
* Расширение за счёт выполнения внешних скриптов
* Гибкая система шаблонов и групп
* Возможность создавать карты сетей

Отдельный блок возможностей связан с автоматическим обнаружением: устройств по диапазону IP-адресов, доступных на них сервисах, также реализована [SNMP](https://ru.wikipedia.org/wiki/SNMP)-проверка. Обеспечивается автоматический мониторинг обнаруженных устройств, автоматическое удаление отсутствующих узлов, распределение по группам и шаблонам в зависимости от возвращаемого результата. Низкоуровневое обнаружение может быть использовано для обнаружения и для начала мониторинга файловых систем, сетевых интерфейсов. Начиная с Zabbix 2.0, поддерживаются три встроенных механизма низкоуровневого обнаружения:

* обнаружение файловых систем;
* обнаружение сетевых интерфейсов;
* обнаружение нескольких SNMP OID.

Поддерживаемые платформы (сервер и агент): [AIX](https://ru.wikipedia.org/wiki/AIX), [FreeBSD](https://ru.wikipedia.org/wiki/FreeBSD" \o "FreeBSD), [HP-UX](https://ru.wikipedia.org/wiki/HP-UX), [Linux](https://ru.wikipedia.org/wiki/Linux" \o "Linux), [Mac OS](https://ru.wikipedia.org/wiki/Mac_OS" \o "Mac OS), [OpenBSD](https://ru.wikipedia.org/wiki/OpenBSD" \o "OpenBSD), [SCO OpenServer](https://ru.wikipedia.org/wiki/SCO_OpenServer), [Solaris](https://ru.wikipedia.org/wiki/Solaris" \o "Solaris), [Tru64/OSF](https://ru.wikipedia.org/wiki/Tru64_UNIX); кроме того, реализованы агенты для [Novell Netware](https://ru.wikipedia.org/wiki/Novell_Netware" \o "Novell Netware) и операционных систем семейства [Windows](https://ru.wikipedia.org/wiki/Windows" \o "Windows).

ELK stack – это набор из трех программ Logstash, Elasticsearch и Kibana, которые позволяют получать информация о работе системы, при проведении нагрузочного тестирования, т.е. повышать качество тестирования.

ELK stack:

* Elasticsearch - это база данных NoSQL, основанная на поисковой системе Lucene.
* Logstash - это инструмент конвейерного журнала, который принимает входные данные из различных источников, выполняет различные преобразования и экспортирует данные для различных целей. Это динамический конвейер сбора данных с расширяемой экосистемой плагинов и сильной синергией Elasticsearch.
* Kibana - это слой пользовательского интерфейса для визуализации, который работает поверх Elasticsearch.

# Оптимизация сети, протоколов взаимодействие, сетевой подсистемы ОС

Настройки сетевой подсистемы ОС, на которой развернуто веб-приложение, мо-жет оказывать существенное влияние на его производительность. Некоторые пара-метры сетевой подсистемы Linux, доступные для настройки:

* Длина очереди для входящих пакетов: net.core.netdev\_max\_backlog
* Количество возможных подключений к сокету: net.core.somaxconn
* Размер буфера под хранение SYN запросов на соединение: net.ipv4.tcp\_max\_-syn\_backlog
* Возможное количество сокетов в состоянии TIME\_WAIT: net.ipv4.tcp\_max\_-tw\_buckets
* Быстрая утилизация сокетов, находящихся в состоянии TIME\_WAIT: net.ipv4.tcp\_-tw\_recycle
* Разрешение использовать существующие сокеты, которые находятся в состоя-нии TIME\_WAIT: net.ipv4.tcp\_tw\_reuse
* Размеры буферов выделяемых под сетевые соединения: net.core.wmem\_max, net.core.rmem\_max, net.core.rmem\_default, net.core.wmem\_default, net.ipv4.tcp\_-rmem, net.ipv4.tcp\_wmem

Длина очереди исходящих пакетов

* Для просмотра и установки этих параметров можно использовать утилиты sysctl, ifconfig, ip.

# Балансировка и маршрутизация

**балансировка нагрузки –** метод распределения заданий между несколькими [сетевыми устройствами](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B5_%D0%BE%D0%B1%D0%BE%D1%80%D1%83%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (например, [серверами](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%B0%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5))) с целью оптимизации использования ресурсов, сокращения времени обслуживания запросов, [горизонтального масштабирования](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D0%BE%D1%81%D1%82%D1%8C#%D0%93%D0%BE%D1%80%D0%B8%D0%B7%D0%BE%D0%BD%D1%82%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BC%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) кластера (динамическое добавление/удаление устройств), а также обеспечения [отказоустойчивости](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D0%B0%D0%B7%D0%BE%D1%83%D1%81%D1%82%D0%BE%D0%B9%D1%87%D0%B8%D0%B2%D0%BE%D1%81%D1%82%D1%8C) ([резервирования](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B7%D0%B5%D1%80%D0%B2%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

В компьютерах балансировка нагрузки распределяет нагрузку между несколькими вычислительными ресурсами, такими как компьютеры, компьютерные кластеры, сети, центральные процессоры или диски. Цель балансировки нагрузки — оптимизация использования ресурсов, максимизация пропускной способности, уменьшение времени отклика и предотвращение перегрузки какого-либо одного ресурса. Использование нескольких компонентов балансировки нагрузки вместо одного компонента может повысить надежность и доступность за счет резервирования. Балансировка нагрузки предполагает обычно наличие специального программного обеспечения или аппаратных средств, таких как многоуровневый коммутатор или система доменных имен, как серверный процесс.

Примеры устройств, к которым применима балансировка:

* [Серверные кластеры](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80_(%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BE%D0%B2))
* [Прокси-серверы](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%BA%D1%81%D0%B8-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80)
* [Межсетевые экраны](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D1%81%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D1%8D%D0%BA%D1%80%D0%B0%D0%BD)
* [Коммутаторы](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BA%D0%BE%D0%BC%D0%BC%D1%83%D1%82%D0%B0%D1%82%D0%BE%D1%80)
* [Серверы инспектирования содержимого](https://ru.wikipedia.org/wiki/Deep_packet_inspection)
* Серверы [DNS](https://ru.wikipedia.org/wiki/DNS)
* [Сетевые адаптеры](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%B0%D1%8F_%D0%BF%D0%BB%D0%B0%D1%82%D0%B0)

# Веб серверы (например, оптимизация nginx)

# DDOS: как организовать и как с ним бороться

# CDN: виды, провайдеры, как использовать

# Часть 2

# Файловые системы и организация хранилища (raid, zfs)

**Что такое RAID и зачем оно нужно**

**RAID** — это дисковый массив из нескольких устройств (жестких дисков). Этот массив служит для повышения надёжности хранения данных и/или для повышения скорости чтения/записи информации.

Собственно, то чем именно занимается оная связка из дисков, – ускорением работы или повышением безопасности данных, зависит от Вас, а точнее, от выбора текущей конфигурации рейда(ов). Разные типы этих конфигураций как раз и отмечаются разными номерами, – **1, 2, 3, 4** и пр, – и выполняют разные функции.

**RAID 0**

Данные разбиваются на блоки, и каждый блок записывает/считывает информацию на отдельный диск поочередно. Если из строя выходит один из дисков, то полностью и безвозвратно пропадает вся информация. RAID 0 дает ускорение в работе из-за чередования чтения/записи. RAID 0 часто используют для размещения временных файлов.

*Применение: Графика, видео*

**RAID 1**

Зеркалирование дисков. На одном диске хранятся данные, на другом – его копия.

*Применение: малые файл-серверы*

**RAID 2**

Применяется код Хемминга. Диски делятся на два типа: один под данные, другой под коды коррекции ошибок. Если диск с данными выходит из строя, то всю информацию можно восстановить со второго диска.

*Применение: мейнфреймы*

**RAID 3**

Данные разбиваются на куски (обычно не больше 1024 байт) и распределяется по дискам. Один диск используют для хранения блоков четности. Нет коррекции ошибок в отличии от RAID2.

*Применение: Графика, видео*

**RAID 4**

Отличается от RAID 3 тем, что данные делятся не на байты, а на блоки. Из-за этого скорость немного увеличивается.

*Применение: файл-серверы*

**RAID 5**

Блоки данных и контрольные суммы циклически записываются на все диски. Под контрольными суммами здесь понимается операция XOR. Этот метод обеспечивает отказоустойчивость. Результаты XOR хранятся на отдельном диске. Минимальное количество дисков 3.

*Применение: серверы баз данных*

**RAID 6**

Похож на RAID 5, но имеет более высокую степень надёжности — три диска данных и два диска контроля чётности. Основан на [кодах Рида — Соломона](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%A0%D0%B8%D0%B4%D0%B0_%E2%80%94_%D0%A1%D0%BE%D0%BB%D0%BE%D0%BC%D0%BE%D0%BD%D0%B0) и обеспечивает работоспособность после одновременного выхода из строя любых двух дисков.

*Применение: используется крайне редко*

**RAID 7**

Для понимания архитектуры RAID 7 рассмотрим ее особенности: 

|  |  |  |
| --- | --- | --- |
| **1.** |  | Все запросы на передачу данных обрабатываются асинхронно и независимо. |
| **2.** |  | Все операции чтения/записи кэшируются через высокоскоростную шину x-bus. |
| **3.** |  | Диск четности может быть размещен на любом канале. |
| **4.** |  | В микропроцессоре контроллера массива используется операционная система реального времени ориентированная на обработку процессов. |
| **5.** |  | Система имеет хорошую масштабируемость: до 12-ти host-интерфейсов, и до 48-ми дисков. |
| **6.** |  | Операционная система контролирует коммуникационные каналы. |
| **7.** |  | Используются стандартные SCSI диски, шины, материнские платы и модули памяти |
| **8.** |  | Используется высокоскоростная шина X-bus для работы с внутренней кэш-памятью. |
| **9.** |  | Процедура генерации четности интегрирована в кэш. |
| **10.** |  | Диски, присоединенные к системе, могут быть задекларированы как отдельно стоящие. |
| **11.** |  | Для управления и мониторинга системы можно использовать SNMP агент. |

**Комбинированные RAID-массивы**

Под RAID 1+0 имеют в виду вариант RAID 10, когда два RAID 1 объединяются в RAID 0. Вариант, когда два RAID 0 объединяются в RAID 1 называется RAID 0+1, и "снаружи" представляет собой тот же RAID 10.

**Реализация RAID систем**

* программная (software-based);
* аппаратная — шинно-ориентированная (bus-based);
* аппаратная — автономная подсистема (subsystem-based).

Главное преимущество программной реализации — низкая стоимость. Но при этом у нее много недостатков: низкая производительность, загрузка дополнительной работой центрального процессора, увеличение шинного трафика. Программно обычно реализуют простые уровни RAID — 0 и 1, так как они не требуют значительных вычислений. Учитывая эти особенности, RAID системы с программной реализацией используются в серверах начального уровня.

Аппаратные реализации RAID соответственно стоят больше чем программные, так как используют дополнительную аппаратуру для выполнения операций ввода вывода. При этом они разгружают или освобождают центральный процессор и системную шину и соответственно позволяют увеличить быстродействие.

Шинно-ориентированные реализации представляют собой RAID контроллеры, которые используют скоростную шину компьютера, в который они устанавливаются (в последнее время обычно используется шина PCI).

Учитывая то, что шинно-ориентированные реализации подключаются прямо к внутренней PCI шине компьютера, они являются наиболее производительными среди рассматриваемых систем (при организации одно-хостовых систем). Максимальное быстродействие таких систем может достигать 132 Мбайт/с (32bit PCI) или же 264 Мбайт/с (64bit PCI) при частоте шины 33MHz.

**ZFS (Zettabyte File System)**

Для создания отказоустойчивого и масштабируемого хранилища данных используют технологии RAID массива, а у файловой системы ZFS имеется свой инструмент для работы с физическими дисками и организации их в **RAID-Z**массивы (аналог RAID5). При этом, в отличие от аналогичных технологий, данная FS самостоятельно восстанавливает поврежденные блоки и исправляет их на лету без вмешательства пользователя.

К основным плюсам данной FS — практически неограниченное дисковое пространство и встроенные инструменты построения отказоустойчивых RAID-Z массивов, относится также и **атомарная запись данных**. Под этим термином понимается следующее — данные либо полностью записываются на диск, либо не записываются вовсе. Такая технология позволяет избежать проблем, которые возникают в случаях сбоев работы сети, подсистемы питания и отказа операционной системы. Поддерживает ОС: Solaris, OpenSolaris, Apple Mac OS X 10.5, FreeBSD, Linux.

**Существует несколько разновидностей RAID-Z массивов, наиболее популярны raid-z1 и raid-z2:**

— raid-z1 — здесь используется для контроля четности один диск из пула (минимум дисков для организации данного вида массива — 3 шт). При выходе из строя одного диска, массив будет работать корректно, при его замене массив перестроится самостоятельно. При выходе из строя двух дисков — массив разрушается и данные восстановлению не подлежат.

— raid-z2 — в данном случае для контроля четности выделяется 2 диска (минимум дисков для такой конфигурации — 5 шт.). Эта система является более отказоустойчивой.

**Достоинства ZFS:**

* встроенные инструменты для работы с разделами HDD и организации RAID-Z
* нет привязки к оборудованию
* нечувствительна к незапланированным отключениям электропитания
* автоматическая подмена вышедших из строя HDD, исправление ошибок и перестроение RAID
* поддерживаются огромные размеры томов, файлов, пулов, а также легкая масштабируемость хранилища
* быстрое и удобное администрирование ZFS Pool’ов
* при увеличении HDD повышается производительность хранилища
* дедупликация и сжатие данных

**Недостатки:**

* высокие требования к ресурсам CPU и RAM
* хранилища, используемые для важных корпоративных данных, должны быть построены при использовании ECC RAM.