Вестник Евразийской науки / The Eurasian Scientific Journal https://esj.today

2018, №4, Том 10 / 2018, No 4, Vol 10 https://esj.today/issue-4-2018.html

URL статьи: https://esj.today/PDF/48ITVN418.pdf

Статья поступила в редакцию 27.08.2018; опубликована 15.10.2018

Ссылка для цитирования этой статьи:
Городничев М.Г., Кочупалов А.Е. Исследование методов межпроцессного взаимодействия в информационной системе с горизонтальным взаимодействием // Вестник Евразийской науки, 2018 №4, https://esj.today/PDF/48ITVN418.pdf (доступ свободный). Загл. с экрана. Яз. рус., англ.

For citation:
Gorodnichev M.G., Kochupalov A.E. (2018). Investigation of interprocess communication methods in the information system with horizontal interaction. *The Eurasian Scientific Journal*, [online] 4(10). Available at: https://esj.today/PDF/48ITVN418.pdf (in Russian)

Грант РФФИ 17-29-03419 «Алгоритмы и технология on-line распознавания на мобильных устройствах движущихся объектов и характеристик потоков»

УДК 654.9

ГРНТИ 20.53.33

Городничев Михаил Геннадьевич
ФГБОУ ВО «Московский технический университет связи и информатики», Москва, Россия
Доцент кафедры «Математической кибернетики и информационных технологий»
Руководитель центра индивидуального обучения
Кандидат технических наук
E-mail: Gorodnichev89@yandex.ru
РИНЦ: http://elibrary.ru/author_profile.asp?id=893531

Кочупалов Александр Евгеньевич
ФГБОУ ВО «Московский технический университет связи и информатики», Москва, Россия
Магистр
E-mail: chupik1994@yandex.ru

Исследование методов межпроцессного взаимодействия в информационной системе с горизонтальным взаимодействием

Аннотация. С появлением Интернета стали развиваться распределено-вычислительные системы (РВС). В настоящее время наиболее распространенной архитектурой РВС является клиент-сервер. Данная архитектура позволяет перенести вычислительную нагрузку сервер, а клиент только отображает результат. В связи с чем ресурсов одного сервера конечны, и независимо от типа серверов, в масштабируемых проектах, в связи с наличием большого количество процессов, запущенных на разных процессорах или серверах, возникает проблема обмена данными между процессами. Авторами освещены методы проектирования клиент-серверных приложений, основанных на трехзвенной архитектуре с использованием различных вариантов межпроцессного взаимодействия. В данной статье рассматриваются методы межпроцессного взаимодействия в проектах с клиент-серверной архитектурой при горизонтальном масштабировании. Авторы провели комплексный анализ проблем, возникающих при межпроцессном взаимодействии, и предложили методы решения. В данной работе исследованы методы организации обмена данными между серверными процессами, клиентскими и серверными процессами. Авторами разработана методика проведения тестирования производительности системы, основанной на основных парадигмах РВС, и

современных технологиях. Исходя из результатов авторами обозначена важность проектирования межпроцессного взаимодействия на всех уровнях трехзвенной архитектуры.

Ключевые слова: межпроцессное взаимодействие; клиент-сервер; синхронизация данных; высоконагруженные системы; кеширование данных; облачные приложения

Облачные технологии все чаще используются в современных приложениях и постепенно вытесняют приложения с другими архитектурами. Облачные приложения – это приложения, имеющие в основе распределенную клиент-серверную архитектуру, где на клиент, в основном, возложена задача ввода и вывода данных, а вычисления выполняется в «облаке» – удаленном сервере или группе серверов, связь с которыми осуществляется через Интернет.

У облачных приложений есть неоспоримое достоинство – они уменьшают требования к вычислительным возможностям клиентских устройств. Это позволяет получать результаты обработки больших массивов данных не только на слабых компьютерах, но и на мобильных устройствах, устройствах IoT (Интернета вещей). Главное требование к устройствам – это наличие стабильного подключения к сети Интернет. На начало 2018 года это легко осуществимо в связи с широким распространением таких технологий, как Wi-Fi, 3G и LTE.

Для того, чтобы обслуживать большое количество клиентских устройств (а их может быть сотни тысяч или миллионы в масштабных проектах), недостаточно иметь один сервер. Особенно это актуально для облачных приложений, которые работают с видеоинформацией и информацией, которая быстро устаревает. Для покрытия мощностных нужд, помимо вертикального масштабирования (увеличение мощности сервера путем замены комплектующих на более производительные), прибегают и к горизонтальному масштабированию. Горизонтальное масштабирование заключается в увеличении количества серверов, узлов, процессоров, обрабатывающих информацию.

При использовании стратегии горизонтального масштабирования, можно увеличивать как количество однотипных серверов, равномерно распределяя между ними задачи одного типа, так и использовать сервера разного типа, выделяя под каждый сервер узкоспециализированные задачи. В современных проектах используют комбинированный подход – используют группы серверов, работающих с разнотипными задачами.

Независимо от типа серверов, в масштабируемых проектах, в связи с наличием большого количество процессов, запущенных на разных процессорах или серверах, возникает проблема обмена данными между процессами.

Данная работа посвящена анализу средств межпроцессного взаимодействия в распределенных вычислительных системах с горизонтальным масштабированием.

Объектами исследования являются взаимодействие между серверными процессами и взаимодействие между серверными и клиентскими процессами. Предметом исследования является информационная система с трехзвенной архитектурой.

Задачи, которые были рассмотрены в данной работе:

1. Исследование проблем, возникающих при организации межпроцессного взаимодействия.
2. Исследование способов организации обмена данными между серверными процессами, клиентскими и серверными процессами.

3. Проектирование клиент-серверного приложения с использованием трехзвенной архитектуры.
4. Тестирование приложения с использованием разных методов обмена информацией о сессиях и клиент-серверного взаимодействия.

1.1 Виды взаимодействия в информационных системах с горизонтальным масштабированием

Межпроцессное взаимодействие является ключевым аспектом в современных информационных системах. От того, насколько грамотно построено взаимодействие, особенно если система представляет из себя распределенную систему с горизонтальным масштабированием, зависит, насколько быстро конечный пользователь будет получать ответную реакцию на свои действия, сколько одновременно пользователей сможет обслужить информационную систему (ИС) и насколько дорого будет содержать и обслуживать ИС [1].

В распределенных информационных системах взаимодействие между процессами происходит как внутри уровней, так и между ними.

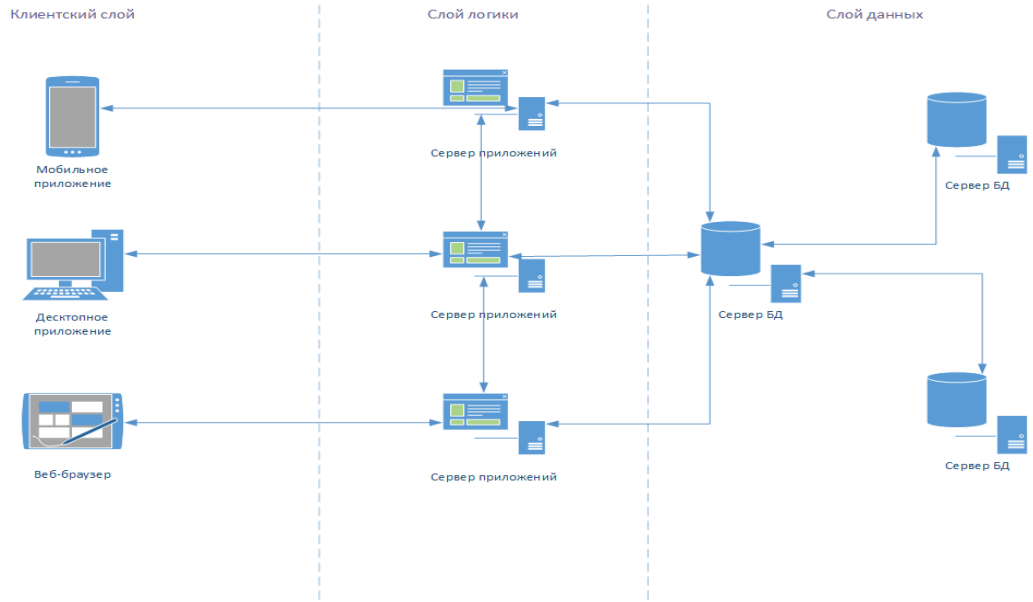


Рисунок 1.1. *Трехуровневая архитектура распределенных систем (составлено авторами)*

На рисунке 1.1 представлена классическая трехзвенная архитектура [2]. Первый вид взаимодействия происходит между слоями клиентской части приложения и серверной части. Взаимодействие происходит со связью вида «многие к одному» (в случае отсутствия горизонтального масштабирования слоя серверов приложения) или «многие ко многим». Реализация этого вида взаимодействия полностью возложена на разработчиков информационной системы. Они могут использовать как готовые программные решения с использованием проверенных архитектурных паттернов, так и собственные наработки. Но, независимо от типа реализации, подавляющее большинство решений работает поверх протокола сетевого уровня IP, так как архитектура сети Интернет основана на этом протоколе.

Следующий вид взаимодействия происходит внутри слоя серверов приложений. В случае масштабируемой системы, на этом слое находятся несколько серверов приложений, которые могут выполнять как одинаковые задачи, так и разнотипные, с узкой специализацией.

Обмен данными между элементами слоя серверов приложений может происходить следующими способами:

1. Обмен данными, не выходя за рамки слоя (рис. 1.2а). Используется специальное программное обеспечение (ПО) для обмена данными. Это могут быть как отдельные системы обмена сообщениями между компонентами системы и key-value хранилища, так и встроенные в серверные процессы механизмы, использующие сокеты и другие виды сетевого взаимодействия.
2. Обмен данными с использованием слоя серверов базы данных (рис. 1.2б). В этом случае, данные, которые нужно передать «соседнему» серверу приложений, отправляются в базу данных, из которой второй сервер приложений производит чтение.

Каждый из способов имеет преимущества и недостатки. Способ с использованием локальных средств, не выходящих за рамки слоя, имеет лучшую производительность и экономит ресурсы серверов базы данных. Но реализация дополнительных механизмов увеличивает сложность разработки ПО для серверов приложений и, в некоторых случаях, может стать причиной снижения отказоустойчивости системы из-за появления дополнительных звеньев в цепи взаимодействия.

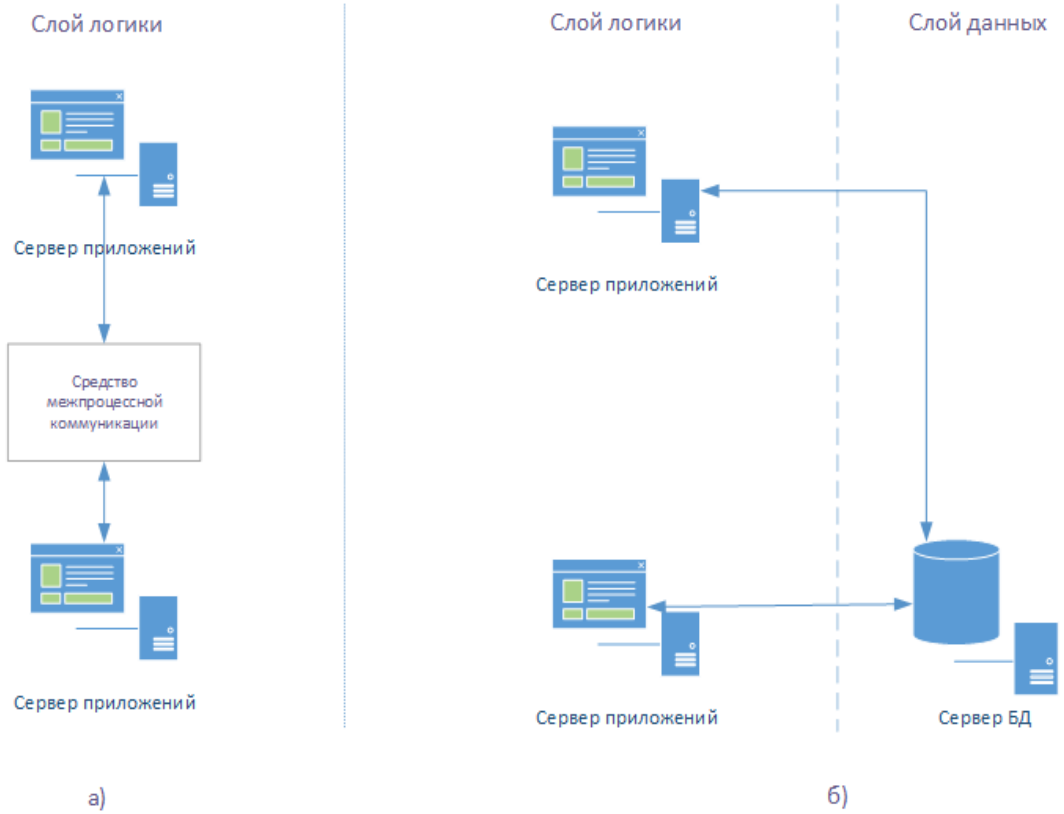


Рисунок 1.2. *Варианты обмена данными между серверами приложений (составлено авторами)*

В последнее время возникла тенденция в осуществлении взаимодействия между клиентскими процессами. Такое решение позволяет разгрузить серверы приложений. В основном, такие решения используется в голосовой и видеосвязи (Skype устанавливает прямые защищенные соединения при разговоре между двумя пользователями), в защищенных средствах обмена сообщениями (Telegram в режиме секретного чата между двумя пользователями [3]) и в кооперативных средствах обмена файлами (протокол BitTorrent).

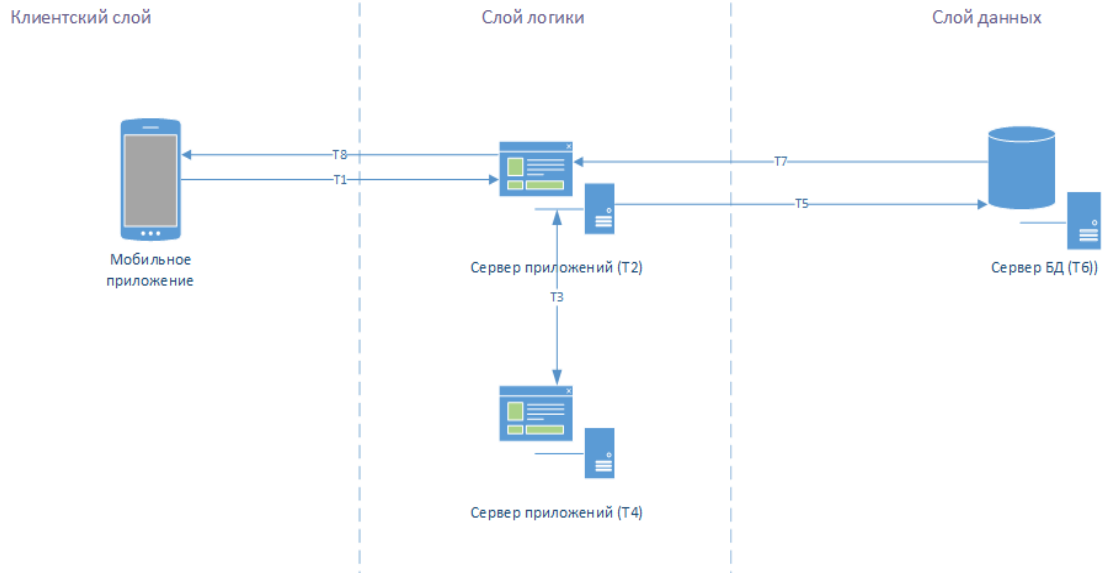


Рисунок 1.3. *Последовательность взаимодействия между элементами информационной системы (составлено авторами)*

На рисунке 1.3 представлена общая схема последовательности взаимодействия ИС с горизонтальным масштабированием [4]. Общее время обработки запроса рассчитывается по формулам 1.1-1.4:

$$Т = Ткл + Тпр + Тбд \quad (1.1)$$

$$Ткл = Т1 + Т8 \quad (1.2)$$

$$Тпр = Т2 + Т3 + Т4 \quad (1.3)$$

$$Тбд = Т5 + Т6 + Т7 \quad (1.4)$$

где:

Т – общее время обработки данных (от отправки запроса с клиентского устройства до получения ответа);

Ткл – время, затраченное на клиентском слое;

Тпр – время, затраченное на слое логики;

Тбд – время, затраченное на слое БД;

Т1 – время передачи запроса от клиентского процесса до сервера приложений;

Т2 – время работы алгоритмов бизнес-логики;

Т3 – суммарное время, затраченное на связь с другим сервером приложений. В некоторых ИС, в которых взаимодействие построено через слой БД (см. рисунок 2.2б), Т3 = 0;

Т4 – время, затраченное на обработку запросов от другого сервера приложений при межпроцессной коммуникации;

Т5 – время, затраченное на передачу запроса от сервера приложений к серверу БД;

Т6 – время обработки запроса к БД;

Т7 – время, затраченное на передачу ответа на запрос к БД;

Т8 – время, затраченное на передачу ответа от сервера приложений к клиентскому процессу.

В формуле (1.1), помимо времени отработки алгоритмов бизнес-логики, большой вклад вносят составляющие, которые относятся к передаче данных между слоями трехзвенной архитектуры или внутри слоя. Как сказано ранее, за T5, T6, T7 ответственны разработчики выбранной СУБД. За остальные составляющие отвечают разработчики конкретной распределенной информационной системы.

В свою очередь, каждый временной компонент, относящийся к взаимодействию между процессами, состоит из вложенных компонентов (формула 1.5).

$$Твз = Тсети + Тсессии + Тданных \quad (1.5)$$

где:

Твз – общее время, затраченное на взаимодействие;

Тсети – время, затраченное протоколами 1 и 2 уровня модели OSI (Ethernet, 802.11) на специфичные этим уровням временные расходы. Такими расходами могут быть, например, время на преодоление коллизий Ethernet, время на шифрование и дешифрование трафика WPA2 и т. д.;

Тсессии – время, затраченное на установление сессии протоколом транспортного уровня или более высокого уровня;

Тданных – время, затраченное непосредственно на передачу данных.

1.2 Проблемы межпроцессного взаимодействия

Оптимизация времени, затрачиваемого на совершение операций межпроцессного взаимодействия, является далеко не единственной задачей, с которой сталкиваются разработчики распределенных систем. Существует также ряд проблем, присущий системам, вычисления которых распределены между несколькими машинами.

1.2.1 Надежность и производительность сетей

Любая система, которая имеет в себе более одной вычислительной машины, требует наличия некоторой сети, через которую происходит обмен данными, построенной по некоторой топологии (рисунок 1.4) [5].

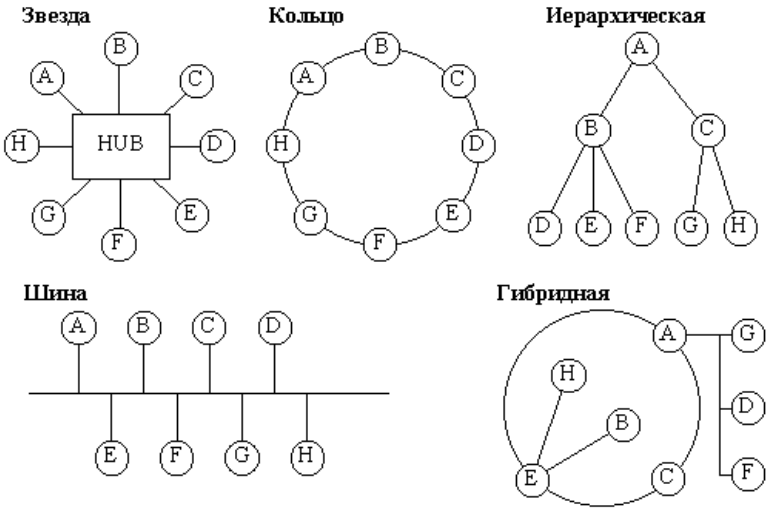


Рисунок 1.4. *Топологии вычислительных сетей. Таненбаум Э., ван Стеен М. Распределенные системы. Принципы и парадигмы*

Топология «шина» представляет из себя единую шину (сетевой кабель), к которой подключены устройства сети. Все устройства делят между собой пропускную способность шины, передавая данные по очереди (когда один компьютер передает данные, все остальные находятся в режиме прослушивания сети). Неисправность шины влечет за собой недоступность элементов, находящихся по обе стороны разрыва. Данная топология почти не применяется в современных сетях по причине низкой производительности и надежности.

Кольцевая топология связывает между собой все устройства в кольцо, данные в котором передаются в одном направлении. В такой топологии каждое устройство часто оказывается занятым передачей чужого трафика, что влечет за собой лишние расходы ресурсов. Выход одного из элементов кольцевой топологии приводит к частичной недоступности других элементов сети.

Иерархическая топология представляет собой древовидную структуру. Скорость передачи данных в иерархических сетях зависит от того, сколько сегментов располагается между источником информации и конечным узлом, для которого эта информация предназначается.

Топология «звезда» объединяет элементы сети с помощью централизованного элемента – коммутатора. Такие сети производительнее ранее представленных, однако отказ коммутатора влечет за собой полную потерю функциональности сети.

Так как каждая топология имеет свои преимущества и недостатки, проектировщики информационных систем пришли к выводу, что наилучшим вариантом является комбинированная топология.

В случае с трехуровневой моделью, на слое БД используется иерархическая топология с главными серверами (Master-server), которые находятся на вершине древовидной топологии и вторичными (Slave-server), что повышает надежность и производительность базы данных за счет дублирования информации на вторичных серверах.

На остальных слоях применяется также комбинированные звездно-кольцевые топологии для обеспечения высокой отказоустойчивости сетей с высокой производительностью. Задача обеспечения оптимальной маршрутизации и коммутации в таких топологиях возлагается на протоколы динамической маршрутизации и предотвращения петель коммутации.

1.2.2 Синхронизация данных

В распределенных системах не представляется возможным рассчитать точное время, за которое произойдет исполнение процессом того или иного действия и рассчитать время, за которое произойдет обмен информации между процессами. В связи с этим, разработаны две модели, которые имеют разные ограничения по времени [6].

Синхронная модель распределенных систем предъявляет строгие рамки ко времени взаимодействия. Главные принципы модели:

1. Локальные часы каждого процесса имеют отклонение от идеального времени, строго не превышающие заданный предел.
2. Время исполнения каждого действия каждым процессом имеет ограничения сверху и снизу.
3. Время обмена сообщениями между процессами ограничено верхним и нижним значением.

Нарушение любого из правил этой модели влечет к неправильному функционированию синхронной распределенной системы или к полному отказу.

Асинхронная модель противоположна синхронной:

1. Локальные часы процесса могут отклоняться на любую величину от идеального значения.
2. Время исполнения действия конечно.
3. Время обмена сообщениями между процессами конечно.

В связи с тем, что современные сети являются сложными и гетерогенными, невозможно предугадать время обмена данными между элементами сети. Современные операционные системы разделяемого времени не дают гарантии заранее предопределить время исполнения даже детерминированного алгоритма с постоянной сложностью и обеспечить точность синхронизации времени процесса. Поэтому, за редким исключением, практически все современные распределенные вычислительные системы являются асинхронными с проблемами, присущими асинхронным системам.

Основной проблемой асинхронных систем является синхронизация данных.

Смоделируем ситуацию, при которой у нас имеется аналогичная топология, но в нее добавлен третий MySQL-сервер в режиме репликации Master-Master. Инициируем запрос на изменение данных на сервере БД2 и, не дожидаясь завершения процедуры синхронизации данных, попытаемся прочесть данные из БД1 и БД2 (рисунок 1.6). В результате, в качестве ответа сервер приложений получил неактуальные данные, несмотря на то, что он их изменил во время прошлой операции. Тем самым, возникло явление рассинхронизации данных, которое может привести к потерям данных при попытке изменить рассинхронизированный объект.

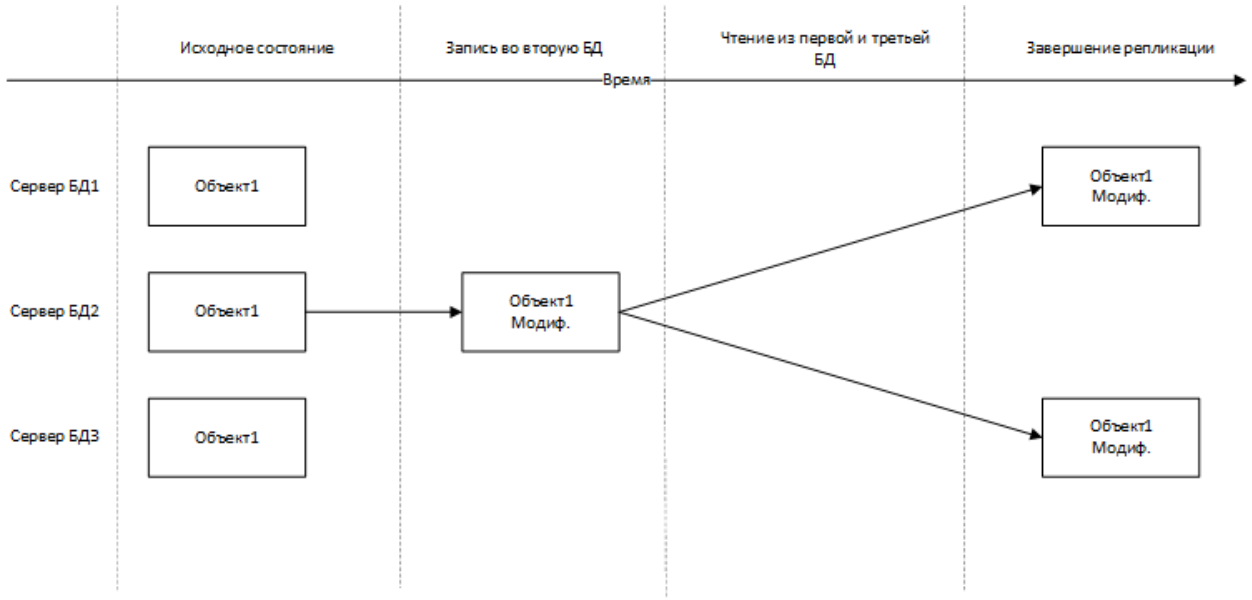


Рисунок 1.6. *Временная диаграмма состояния объекта на серверах БД в режиме репликации Master-Master (составлено авторами)*

Основным способом борьбы с этим явлением является использование промежуточных узлов, которые берут на себя функцию распределения запросов (сообщений) и управления синхронизацией. Репликация данных в режиме Master-Slave (рисунок 1.7) является примером конкретной реализации централизованного управления синхронизацией.

В режиме Master-Slave, все запросы на запись обязательно приходят на первичный сервер (который выбирается предварительно администратором БД в ходе настройки), который инициирует процедуру репликации данных через некоторое время. В случае, когда необходимо получить данные без изменений, обращение идет к вторичным серверам (Slave), но, если необходимо прочесть данные с целью их изменить, то чтение производится с первичного сервера.

2.1 Необходимость локального взаимодействия процессов в системах с горизонтальным масштабированием

Как было сказано ранее, в трехзвенной модели распределенных вычислительных систем (РВС) с горизонтальным масштабированием иногда необходимо синхронизировать данные между узлами, находящимися в пределах одного слоя. В случае со слоем БД, синхронизация реализуется разработчиками СУБД с помощью встроенных механизмов СУБД и драйверов БД. Для серверов приложений синхронизация требуется не всегда. В редких случаях, серверная логика позволяет масштабирование ее обработчиков в самостоятельные ноды без потери функциональности. Но существуют ситуации, когда необходим прямой и быстрый обмен информацией между процессами, реализующими серверную логику.

В качестве первого примера выступает система, в которой имеются разнородные серверы приложений, выполняющие разный функционал (рисунок 2.1). Один сервер обрабатывает бизнес-логику и записывает изменения в базу данных, второй сервер занимается конвертированием и масштабированием изображений, третий – обработкой видео.

Двухсторонняя связь в данной архитектуре необходима для того, чтобы можно было передавать на исполнение задачи от сервера логики приложения к серверам-обработчикам мультимедийной информации. Прямая связь между серверами позволяет разгрузить элементы слоя базы данных.

Вторым примером выступает система, в которой есть необходимость в очень частом доступе к общей информации, которая имеет большой объем и при этом относительно редко изменяется. Такой информацией является данные о сессиях пользователей веб-приложений.

Сессия пользователя представляет собой набор данных, состоящий из следующих полей:

1. Уникальный идентификатор пользователя, который позволяет однозначно идентифицировать его (обязательное поле).
2. Токен. Представляет собой хеш, сгенерированный после последней аутентификации, который используется при авторизации пользователя (обязательное поле).
3. Срок действия токена (поле, обязательность которого зависит от требований, предъявляемых к безопасности информационной системы).
4. Другие необязательные поля, которые характеризуют текущее состояние клиента (последний открытый раздел).

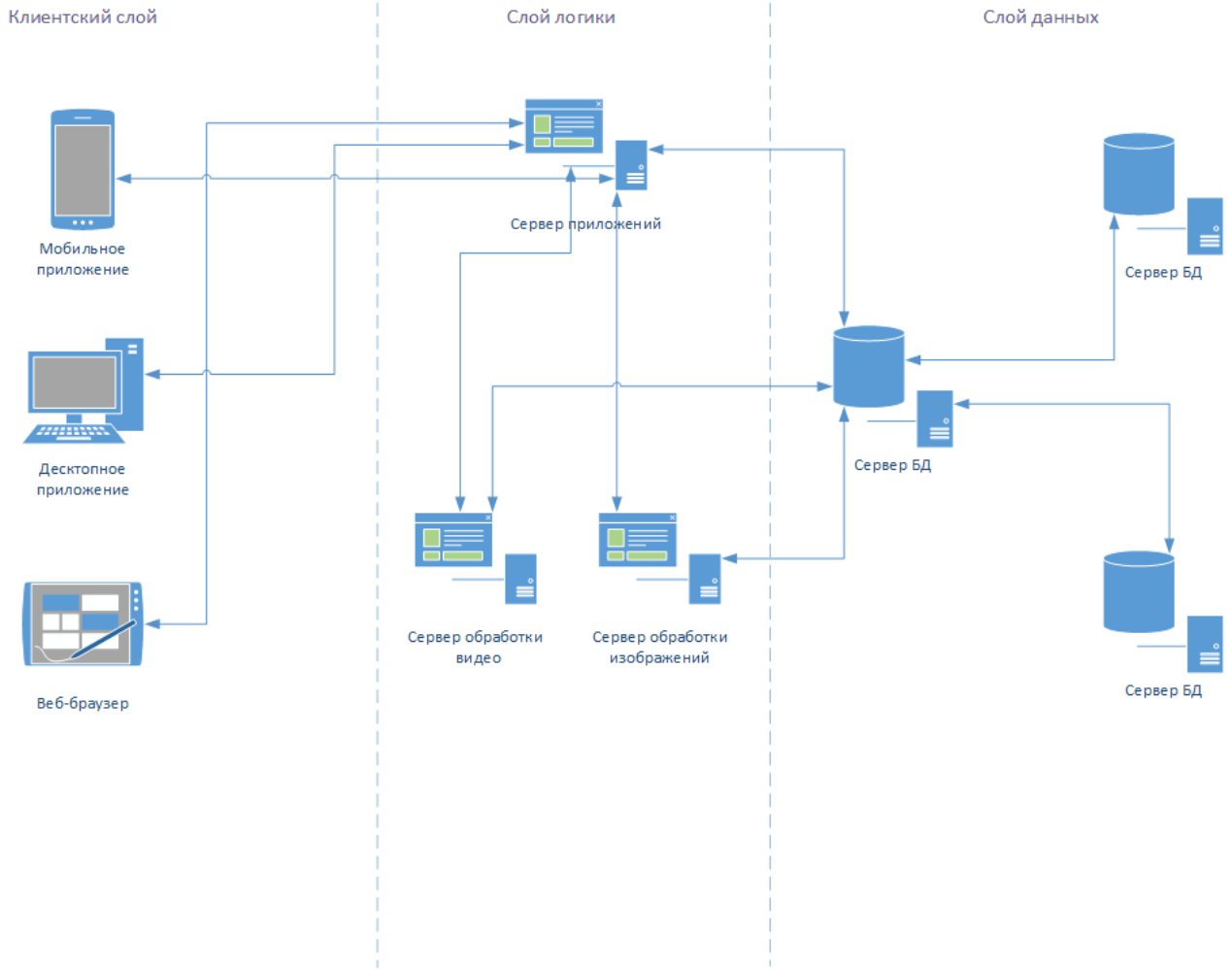


Рисунок 2.1. *Трехзвенная архитектура с серверами приложений, выполняющие разные задачи (составлено авторами)*

В целях безопасности, корректно реализованные приложения производят авторизацию перед каждым действием пользователя, независимо от его характера – чтение данных, изменение, добавление или удаление записей. Операция чтения данных является очень частой процедурой, а авторизация этой операции производится с такой же интенсивностью.

Хранение данных о сессии в основной базе данных означает как минимум двукратное увеличение запросов к ней. Это влечет за собой увеличение количества сетевого трафика между серверами приложений и серверами БД, а также увеличение времени выполнения алгоритмов бизнес-логики в связи с увеличением времени ожидания данных. Реляционные базы данных, зачастую, сильно неэффективны в плане выполнения частых и простых запросов. Поэтому, задачу хранения и получения данных о сессиях решают с помощью других механизмов.

Возможные способы синхронизации данных между серверами приложений:

1. Синхронизация через слой базы данных. Является простым решением в плане реализации, но не подходит в случаях, когда требуется высокопроизводительное решение, а в качестве основных баз данных выступают реляционные БД.
2. Использование средств кеширования данных в ОЗУ с возможностью объединения хранилища между несколькими нодами (memcached).
3. Использование специализированных быстрых хранилищ типа «ключ-значение» (Redis).

4. Использование средств межпроцессного обмена сообщениями (RabbitMQ).
5. Использование собственных наработок, спроектированных под конкретную распределенную систему. Возможны как простые решения, так и вариации с возможностью соединения p2p, самоорганизации сети и контролем доступности узлов.

Возможно комбинирование сразу несколько вариантов из перечисленных выше.

2.2 Синхронизация через слой базы данных

Этот способ заключается в хранении данных о сессии в основной базе данных (рисунок 2.2). Выделяется отдельная таблица, в которую записываются сессионные данные (в случае реляционной БД), или отдельный документ (в случае нереляционных документоориентированных БД).

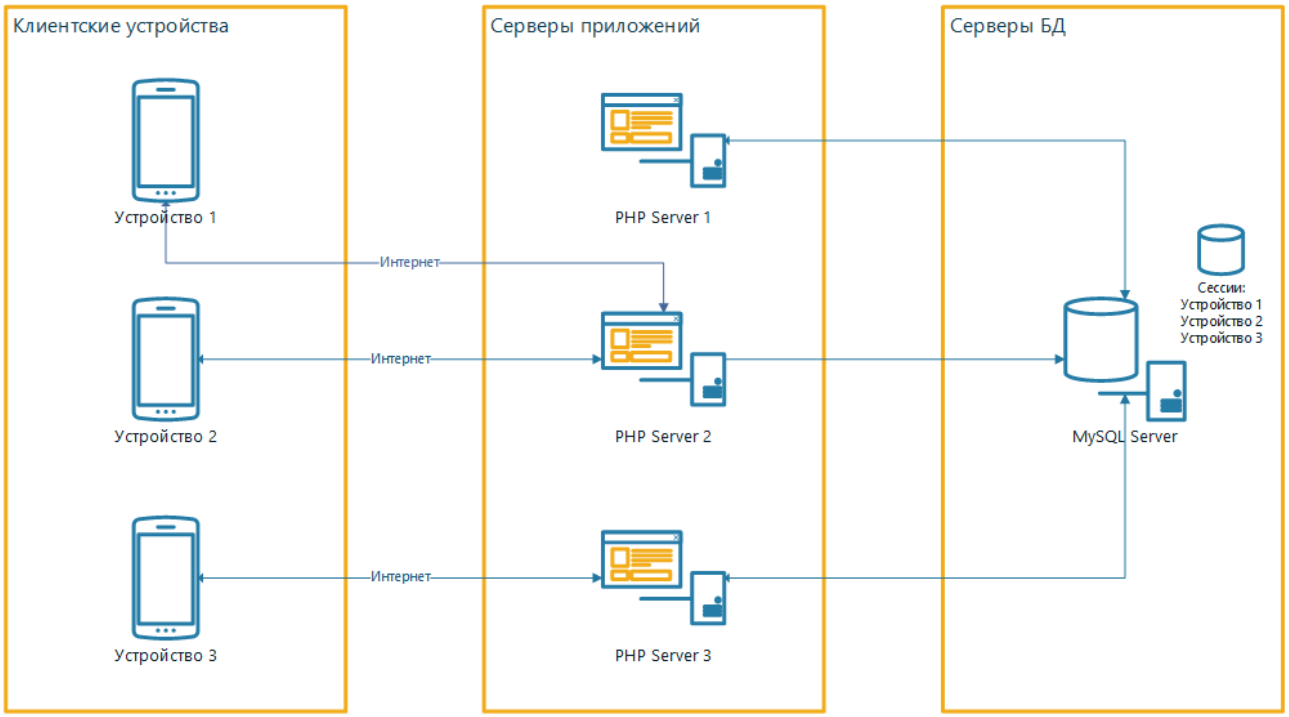


Рисунок 2.2. *Синхронизация сессий через основную базу данных (составлено авторами)*

Такому виду синхронизации присущи следующие особенности:

1. Простота реализации. Отсутствие необходимости в дополнительных инструментах. Иногда требуется дополнительная настройка СУБД для ускорения выполнения простых запросов.
2. Увеличение нагрузки на основной сервер БД.
3. Медленная реализация даже в случае оптимизации настроек БД. Современные БД (особенно реляционные) имеют сложные многоступенчатые алгоритмы парсинга запросов, а также алгоритмы поиска и чтения запрошенных данных. На больших выборках такие СУБД работают быстро, обеспечивая приемлемое время исполнения, но на кратких повторяющихся запросах данные БД избыточны и медленны.

4. Устойчивость к потерям данных. СУБД имеют множество механизмов, которые направлены на обеспечение целостности информации, даже в случаях, когда во время транзакций происходит сбой ОС или аппаратного обеспечения.

2.3 Средства кэширования данных с возможностью объединения памяти

Первичной задачей средств кеширования данных является ускорение доступа к повторно запрашиваемым данным. Кэш – это промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть с высокой долью вероятности запрошена [7].

Размер кэш-памяти намного меньше размера основного хранилища данных.

Важным аспектом кэш-памяти является способ адресации данных. Данные в кэш-памяти представляют из себя пару ключ-значение. Адресация происходит через ключ. В качестве ключа может выступать:

1. Полный идентификатор объекта.
2. Часть идентификатора объекта.
3. Результат вычисления хэш-функции от всего объекта или от некоторых его полей.

Использование хеша или части идентификатора объекта в качестве ключа позволяет снизить сложность поиска значения по заданному адресу до O(1), то есть, независимо от расположения объекта, поиск будет выполнять приблизительно за одинаковое время (с некоторыми поправками).

При доступе к данным по заданному ключу возможно два варианта – кэш-попадание (по заданному адресу находятся нужные данные) и кэш-промах (данные отсутствуют, либо относятся к другому объекту). Запись возможна как в пустую ячейку данных, так и поверх другого записанного объекта.

Несмотря на то, что кэш является непостоянным хранилищем данных и данные могут быть утеряны в результате исчерпания лимита кэша или перезаписи значения, в некоторых сценариях возможно безопасное использование средств кеширования в качестве инструмента информационного обмена между серверами. Важно предусмотреть дублирование данных (если это необходимо) в постоянной памяти.

2.4 Система распределенного кеширования memcached

Memcached – ПО, осуществляющее кэширование данных в ОЗУ на основе хэш-таблицы. Это свободное ПО с открытым исходным кодом под лицензией BSD. Memcached возможно запустить под Unix-подобными ОС (Linux, macOS, FreeBSD) и Windows.

Memcached предоставляет огромную хэш-таблицу, распределенную (при необходимости) между несколькими машинами. Когда таблица заполнена, поступающие записи перезаписывают старые, менее всего используемые, данные. Приложения, работающие с распределенным кэшированием, обычно сначала обращаются к memcached с запросом на получение данных и, в случае если данные отсутствуют, запрашивают уже данные из медленных хранилищ.

Memcached основывается на клиент-серверной архитектуре взаимодействия (рисунок 2.3). Серверы содержат в ОЗУ ассоциативный массив типа «ключ-значение», клиенты заполняют массив и совершают запросы к нему. Максимальный размер ключа составляет 250 байт, максимальный размер значения – 1 мегабайт. В качестве значений поддерживаются

только строки, что не всегда удобно и эффективно для хранения и обработки некоторых структур данных.

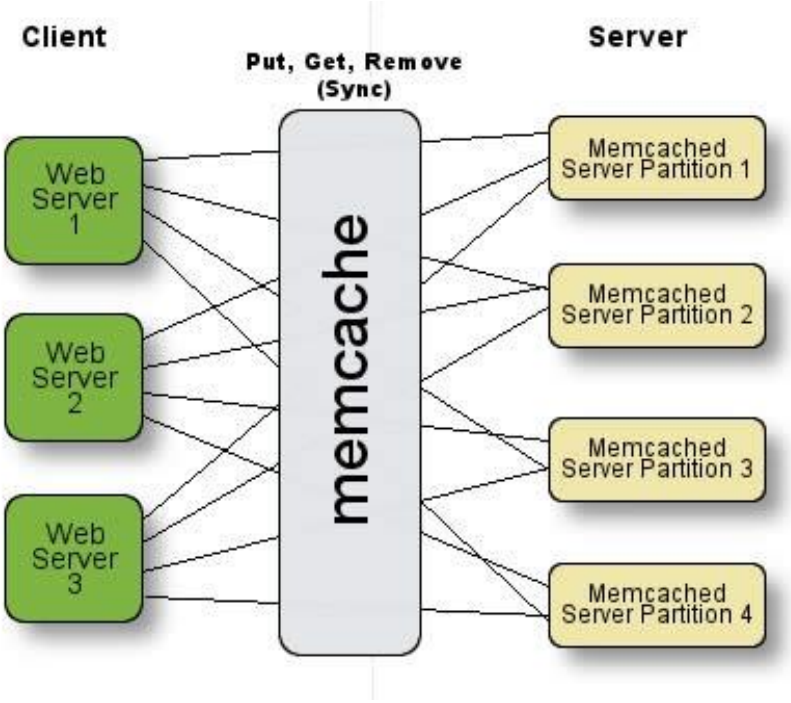


Рисунок 2.3. *Масштабирование memcached. Fitzpatrick B. Distributed caching with memcached*

Клиенты совершают запросы к серверам memcached с помощью клиентских библиотек, устанавливая соединение через TCP или UDP протокол (порт 11211). Клиенту известен весь набор серверов, тогда как серверы не имеют информации о клиентах (запросы инициируются только клиентами). При запросе данных клиентом, клиентская библиотека на основе хэша ключа определяет, какой сервер использовать. Распределение данных посредством разделения диапазонов ключей обеспечивает масштабируемость решения (рисунок 2.3).

На рисунке 2.4 представлен вариант реализации хранилища сессий с использованием распределенного кэша memcached. Сессии, помимо нахождения в БД, также кэшируются на серверах memcached и доступны со всех серверов приложений.

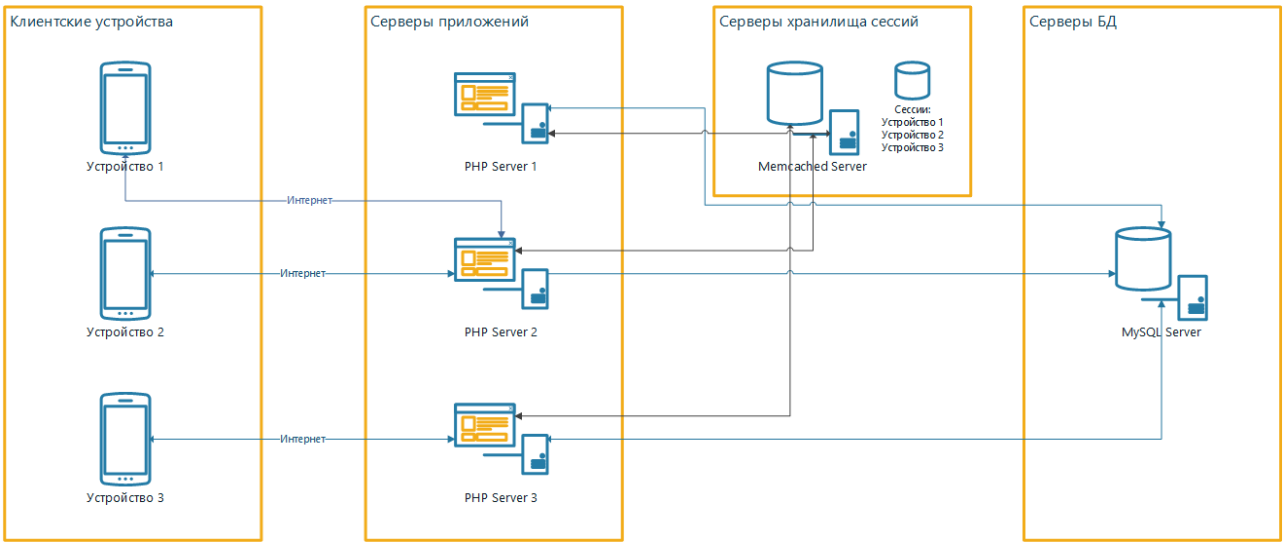


Рисунок 2.4. *Хранение сессий с использованием Memcached (составлено авторами)*

Разработчики проектов с использованием memcached обычно реализуют следующий сценарий:

1. Редко используемые данные и сложные данные хранятся в основной БД.
2. Часто используемые и устойчивые к изменениям данные генерируются при первом запросе, сохраняются в основную БД и также добавляются в хранилище memcached.
3. В случае повторного запроса, данные запрашиваются из memcached и, если они отсутствуют, производится запрос к основной БД (либо данные генерируются заново).
4. В случае редактирования данных, изменения немедленно записываются в основную БД или запись производится периодически (в зависимости от требований к производительности и критичности к потерям данных).

Важно помнить, что в memcached нельзя хранить, не используя дублирование данных, пользовательские данные, которые невозможно восстановить из уже имеющихся данных. В то же время, в memcached возможно хранение данных сессии – токенов, полученных в результате аутентификации. Их потеря в случае форс-мажорных ситуаций вызовет лишь внеплановую необходимость заново аутентифицироваться, а в некоторых случаях лишняя генерация новых токенов полезна с точки зрения безопасности.

2.5 Хранилище данных типа «ключ-значение» Redis

Redis – это ПО с открытым исходным кодом, реализующее хранилище данных в ОЗУ и ПЗУ типа «ключ-значение» и используемое в качестве базы данных, кэша и средства передачи сообщений (средства межпроцессного взаимодействия) [8]. Redis поддерживает несколько типов хранимых данных, имеет встроенную систему репликации данных, поддержку скриптов Lua, систему кеширования данных с продвинутыми алгоритмами замещения, поддержку транзакций и несколько уровней «постоянства» хранения данных.

Для достижения высоких показателей производительности, Redis реализует хранение данных в оперативной памяти. В зависимости от надобностей, разработчик может гибко настраивать процесс сохранения данных в постоянную память (от периодического дампа всех данных хранилища до записи новых данных после каждой операции). При необходимости, сохранение данных в ПЗУ можно отключить, превратив Redis в полноценную распределенную сетевую систему кеширования.

Redis может хранить следующие типы данных:

1. Строка.
2. Массив строк.
3. Набор строк (коллекция строк без повторных значений).
4. Упорядоченные наборы строк (коллекция строк без повторов, сортированная по значению).
5. Хэш-таблицы.
6. Данные типа «HyperLogLogs», служащие для приблизительной оценки количества уникальных значений в наборах данных.
7. Геолокационные данные.

Redis поддерживает master-slave репликацию. Аналогично MySQL, данные основного сервера Redis могут быть продублированы на любом количестве подчиненных серверов. Репликация может быть настроена таким образом, чтобы было возможно использование серверов в несинхронизированном состоянии, путем использования механизма подписи на события и реализации логики обработки этих событий на серверах приложений. Как и в случае master-slave репликации MySQL, это решение обеспечивает масштабируемость чтения, но не обеспечивает масштабируемость записи.

Для более гибкого масштабирования, разработана подсистема Redis Sentinel. Основные ее задачи:

1. Мониторинг. Процессы Sentinel постоянно следят за тем, чтобы основные и подчиненные серверы были в рабочем состоянии.
2. Оповещение. Уведомления о событиях, происходящих с экземплярами Redis, могут высылаться администратору или другим программам через API в случае возникновения нештатных ситуаций.
3. Автоматизация отказоустойчивости. В случае отказа основного сервера, Sentinel производит реконфигурацию и назначает один из бывших подчиненных серверов в качестве основного.
4. Обеспечение конфигурации. Клиенты, обращаясь к Sentinel, могут узнавать актуальные адреса основного сервера в случаях реконфигурации.

Sentinel на момент 2018 года является уже устаревшим решением по масштабированию Redis. В качестве замены выступает кластеризация Redis с использованием набора утилит Redis Cluster. Основным его преимуществом является одновременная поддержка шардинга и репликации. Под шардингом понимается распределение частей БД на несколько серверов, причем каждый сервер хранит только определенную часть, а не полную копию, как в случае с репликацией. Это позволяет построить архитектуру, которая отказоустойчива и быстра одновременно.

Каждый экземпляр сервера Redis работает в однопоточном режиме. В случае, когда необходимо записать изменения в файлы, запись в которые происходит в режиме «только добавление», возможна работа в двух потоках. Это является слабым местом Redis, так как процесс Redis не может запускать хранимые процедуры и другие задачи параллельно. Все операции выполняются последовательно, что неэффективно в случае использования многопроцессорных систем.

Хотя Redis является key-value хранилищем данных, возможен запрос данных не только по ключу, но и по соответствию одному и более критериев. Поддерживаются операции пересечения и объединения, выборка диапазона данных, операции агрегации данных, сортировки и множество других операций, которые присущи «полномасштабным» реляционным и NoSQL СУБД. Но, в отличие от операций прямого доступа данных по ключу, доступ к данным через дополнительные операции имеют другой порядок сложности, что следует помнить при разработке высоконагруженных проектов.

Redis является более гибким решением, чем memcached, и позволяет реализовать быстрое и безопасное хранилище с меньшими затратами сил на обеспечение сохранения данных в ПЗУ. Использование этого решения также позволяет разгрузить основной сервер базы данных, так как, благодаря наличию механизмов сохранения данных в надежном месте и возможности их репликации, нет необходимости дублировать эти данные в основной базе данных.

3. Исследование межпроцессного взаимодействия между серверными и клиентскими процессами

Компьютерные сети прошли и продолжают проходить этапы стремительного развития. Совершенствовались методы доступа в сеть, освоена оптическая среда с целью передачи данных, увеличивалась пропускная способность сетей. Прогресс продолжается, и пользователи с мобильных устройств могут получать доступ к сети Интернет на скоростях свыше 144 МБит/с (LTE Advanced).

Высокий темп развития сетей не является причиной для остановки развития методов взаимодействия между клиентскими и серверными процессами. На это имеются следующие причины:

1. Зачастую, пользователи не могут использовать всю мощь современных сетей. Провайдеры предоставляют линейку из нескольких тарифных планов с ограничением максимальной скорости загрузки данных и доступного количества трафика в месяц.
2. Специфика передачи данных в неограниченных средах (беспроводная связь). Для WiFi и мобильных сетей выделяется ограниченный диапазон радиочастот, в которых необходимо «уместить» всех пользователей. Чем больше пользователей находится в пределах зоны действия одной базовой станции или WiFi точки доступа и чем больше трафика генерируется в этих сетях, тем меньшая скорость выделяется на каждого отдельного пользователя.
3. Отказ мобильных операторов от тарифных планов без ограничения на объем трафика (*прим.* в РФ, на момент 2018 г.) [10] и дороговизна трафика, полученного или переданного сверх тарифного плана.

Исходя из вышеприведенных причин, необходимо тщательно подходить к вопросу разработки методов взаимодействия клиентов с серверами.

Современные сети, Next Generation Networks, основываются на протоколе сетевого уровня IP [11]. Разработчикам приложений, которые работают в сети Интернет, на выбор остаются только уровни модели OSI, которые располагаются выше третьего. Взаимодействие может реализовано следующими способами:

1. Проектирование собственного стека протоколов, начиная с транспортного уровня. Эффективный, но не применимый в современных ОС вариант. Передача данных на транспортном уровне без протоколов TCP и UDP требует использования сырых сокетов, которые полностью разрешены только в Unix-подобных ОС. В других ОС (Windows и iOS) сырые сокеты или запрещены, или сильно ограничены вплоть до возможности передачи только ICMP-пакетов. Данное ограничение обходится написанием драйверов, что усложняет установку приложения или невозможно в принципе в мобильных операционных системах.
2. Реализация протокола прикладного уровня поверх TCP или UDP. Эффективный, но трудозатратный вариант. Так как браузеры не поддерживают взаимодействие через TCP/UDP-сокеты, веб-приложения при выборе только этого вида взаимодействия реализовать невозможно.
3. Использование широко распространенных протоколов прикладного уровня. К ним относятся HTTP. Разработка API поверх HTTP является наименее затратной в плане времени и сил. Используется почти во всех Web-приложениях.

4. Использование протоколов транспортного уровня, поддерживаемого браузерами (WebSocket). На базе этих протоколов возможно построение высокопроизводительных решений, имеющих все преимущества реализаций на TCP-протоколе и главное преимущество HTTP-поддержку взаимодействия с браузерами. Как и в случае с HTTP, необходима реализация собственного протокола прикладного уровня.

4. Тестирование реализаций средств межпроцессного взаимодействия

Часто возникают ситуации, когда результаты даже детально проработанного теоретического исследования не всегда совпадают с реальными, так как могут быть не учтены некоторые аспекты, которые не очевидны с первого взгляда. Поэтому, прежде чем использовать наработки в больших проектах, необходимо закрепить теорию с помощью практических исследований.

4.1 Описание функций и архитектуры приложения

Разработанное приложение под названием Notes является многопользовательским приложением. Основная задача приложения – хранение пользовательских заметок пользователя на удаленном сервере. Для заметок доступны следующие операции:

1. Создание новой пустой заметки.
2. Редактирование содержимого заметки.
3. Удаление заметки.
4. Отображение полного списка заметок пользователя.

Пользователь взаимодействует с заметками через приложение, установленное на мобильное устройство под управлением операционной системы iOS. Приложение отправляет запросы через сеть Интернет на сервер приложений (рисунок 4.1), который, в свою очередь, обрабатывает запрос, запрашивая, при необходимости, данные на сервере БД, и отправляет ответ мобильному устройству. Приложение использует классическую трехзвенную архитектуру, которая была описана в начале статьи. При необходимости, возможно увеличение количества серверов приложений или серверов базы данных.

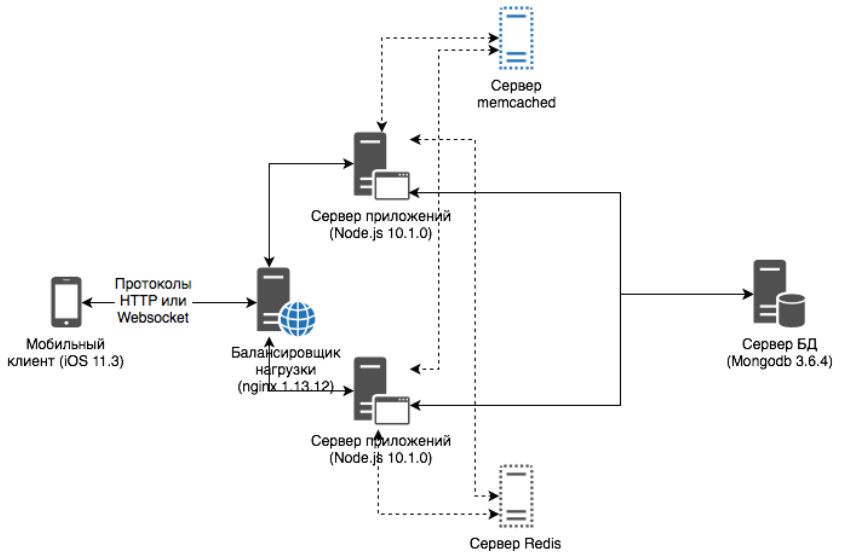


Рисунок 4.1. *Архитектура приложения Notes (составлено авторами*

Взаимодействие сервера и клиента возможно через протоколы HTTP/1.1, HTTP/2, WebSocket и их защищенные варианты с использованием TLS [12].

Доступ к заметкам предоставляет после прохождения процедур аутентификации и авторизации. Аутентификация осуществляется через связку «имя пользователя – пароль». Каждый пользователь имеет доступ только к своим заметкам.

На рисунке 4.1 также обозначены дополнительные серверы для хранения данных сессий – сервер memcached и сервер Redis. На сервере приложений есть возможность выбрать метод работы с сессиями – хранение в основной БД, кеширование на сервере memcached и кеширование на сервере Redis. Метод указывается перед запуском сервера приложений в виде параметра командной строки (-redis для включения кеширования на Redis-сервере, -memcached в случае необходимости хранения сессий на memcached-сервере).

4.2 Описание использованных инструментов

Кроме Redis и memcached, при разработке приложения были использованы инструменты, которые не были упомянуты ранее. Список инструментов включает в себя платформу Node.js, документоориентированная СУБД MongoDB и комплект средств разработки iOS SDK.

4.2.1 Программная платформа Node.js

Node.js – кроссплатформенная программная среда с открытым исходным кодом, которая исполняет JavaScript-код на стороне сервера.

Изначально язык программирования JavaScript был предназначен для написания скриптов для веб-страниц, исполняемых на стороне клиента браузером. Node.js представляет новую парадигму под названием «JavaScript везде», что означает, что JS-код можно использовать не только для динамических веб-страниц, но и для реализации десктопных приложений и серверных приложений.

Node.js поддерживается Windows, Linux, macOS и другими ОС. Базовый функционал включает в себя модули, обеспечивающие поддержку файловой системы, сетевые возможности (HTTP, TCP, UDP, DNS, TLS), работу с бинарными данными, криптографию и др. Функциональность можно бесконечно расширять, используя системы управления зависимостями с широким спектром модулей от других разработчиков.

4.2.2 Менеджер пакетов npm

npm (Node Package Manager) – программное обеспечение, основной задачей которого является управление зависимостями приложений, написанных с использованием программной платформы Node.js. Менеджер представляет из себя совокупность консольного клиентского приложений и онлайн базу данных из готовых пакетов. Поиск по реестру доступен как через консольный клиент, так и через веб-сайт.

В базе представлены как небольшие вспомогательные модули, имеющие узкий функционал, так и фреймворки, которые составляют основу веб-приложений. Модули, представленные в базе, также могут иметь зависимости от других модулей.

В приложении Notes, на котором было произведено тестирование, использовались пакеты из базы npm:

1. Express. Базовый фреймворк для написания веб-приложений, работающих по HTTP-протоколу. В фреймворке представлены функции для обработки запросов, составления ответов, обеспечения маршрутизации и др.
2. Body Parser. Модуль, использующийся для парсинга содержимого тела запроса в формате json.
3. WebSoket. Модуль, реализующий протокол WebSocket согласно RFC 6455 [13].
4. Mongoose. Модуль, являющийся драйвером для базы данных MongoDB.
5. Bcrypt. В данном модуле содержится набор криптографических функций. Некоторые из них являются удобными высокоуровневыми обертками над базовыми функциями криптографии Node.js.
6. Memjs. Модуль для взаимодействия с серверами memcached.
7. Redis. Драйвер для СУБД Redis.
8. Spdy. Модуль, добавляющий поддержку протокола HTTP/2 в Node.js.

4.2.3 СУБД MongoDB

MongoDB – нереляционная документоориентированная СУБД [14]. Данные в базе данных хранятся в виде JSON-документов (рисунок 4.2).

Отличительной особенностью документоориентированной базы данных от реляционной является возможность хранения дочерних документов внутри основного документа (рисунок 4.2) [14]. Реляционная БД не может хранить несколько объектов в одном поле данных, так как это нарушает условие атомарности данных (1 нормальная форма). Для связи объектов в реляционных БД используются внешние ключи, а для организации связи с кратностью «многие-ко-многим» требуется создание и сопровождение дополнительной таблицы данных.

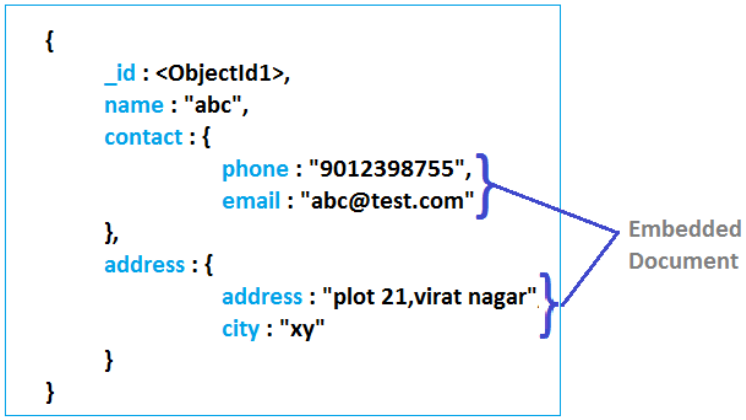


Рисунок 4.2. *Структура документа MongoDB со вложенными поддокументами*

В MongoDB, при необходимости, также можно использовать связи между объектами с помощью внешних ключей (рисунок 4.3).

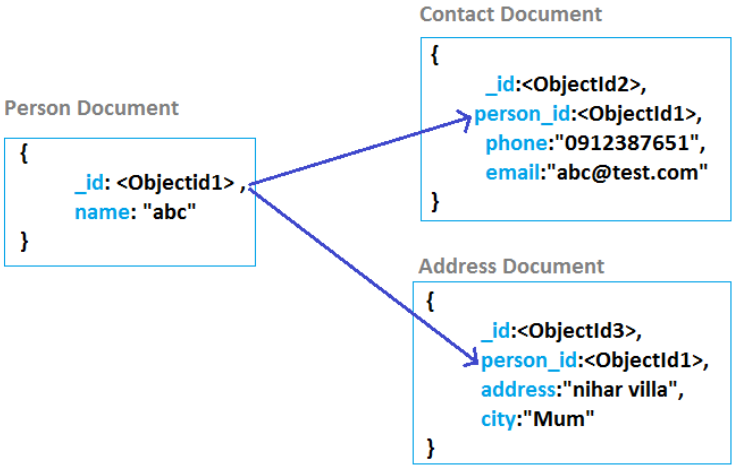


Рисунок 4.3. *Структура документов MongoDB с реализацией связей через внешние ключи* *http://pingax.com/mongodb-schema-design*

Запросы в MongoDB осуществляются на языке MongoDB Query Language. Запросы имеют структуру, похожую на JSON документы. Доступны регулярные выражения, транзакции и агрегация данных.

4.2.4 Платформа для автоматизации развертывания приложений Docker

Процессы масштабируемого приложения могут быть распределены по различным физическим и виртуальным серверам. С ростом пользователей, количество серверов может возрастать, следовательно, возрастает сложность развертывания и поддержки приложения. С целью упрощения этих задач и автоматизации, созданы специальные инструменты развертывания приложений.

Docker является одним из таких инструментов. Первоначально разработанный под ОС Linux, Docker использует принцип виртуализации на уровне ОС, используя такие функции ядра, как cgroups (изоляция ресурсов CPU, RAM, сети и т. д.) и kernel namespaces (изоляция виртуальных ресурсов ядра).

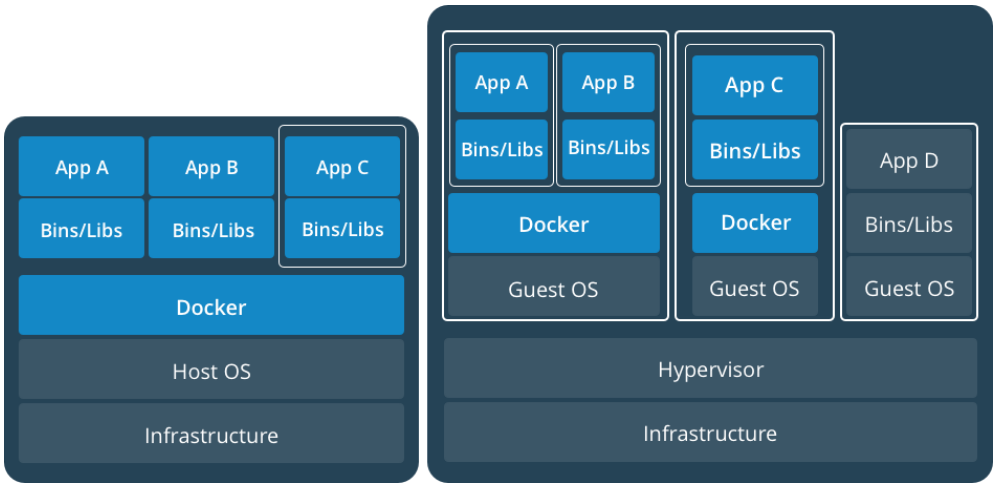


Рисунок 4.4. *Сравнение Docker с полной виртуализацией* *https://www.docker.com/what-container*

Использование такого подхода обеспечивает высокую производительность контейнеров с минимальными потерями, в отличие от полной виртуализации, где необходимо для каждого контейнера запускать свой экземпляр ядра.

Образы контейнеров Docker очень компактные, так как в них содержатся только дополнительные файлы, не входящие в состав базовой ОС. Разворачивание готового образа, хранящегося в репозитории образов, происходит за считанные минуты. Возможно разворачивание одного образа сразу на весь кластер (при использовании вспомогательных инструментов).

4.2.5 Набор инструментов iOS SDK

iOS SDK – набор инструментов для разработки приложений для устройств, работающих под управлением операционной системы iOS. Набор инструментов поставляется вместе с Xcode – интегрированной средой разработки для ОС, разработанной корпорацией Apple. В состав SDK входят компиляторы Swift и Objective-C под архитектуры процессоров ARMv7 и ARM64, набор высокоуровневых и низкоуровневых библиотек и симуляторы iOS, с помощью которых можно протестировать приложения на компьютере под управлением macOS.

Помимо инструментов, предназначенных непосредственно для разработки, в комплект входит набор вспомогательных утилит для подписи приложения сертификатом разработчика и подготовки приложения к публикации в магазине приложений AppStore.

4.3 Проектирование базы данных

Предметная область «Сервис облачных заметок» состоит из двух основных объектов – пользователь заметок и заметка. На рисунке 4.5 представлена инфологическая модель предметной области [14].

Пользователь может регистрироваться под своим именем (username) и паролем (password) и входить в систему, введя правильные данные, указанные на этапе регистрации.

Объект «заметки» состоит из двух полей – заголовка (heading) и основного текста (text). Объекты возможно создавать, изменять и удалять.

Каждая заметка прикреплена к определенному пользователю. Пользователь может просматривать, изменять и удалять только те заметки, которые были им созданы.

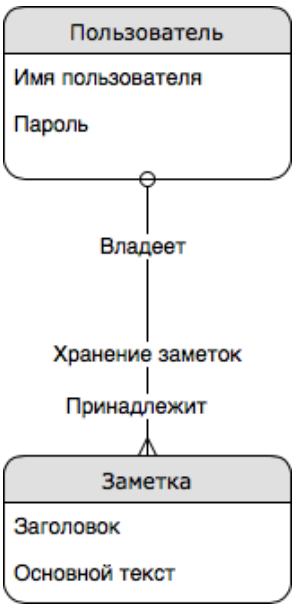


Рисунок 4.5. *Инфологическая модель предметной области «Сервис облачных заметок» (составлено авторами)*

Даталогическое проектирование заключается в преобразовании инфологической модели в выбранную даталогическую модель. Преобразованная модель представлена на рисунке 4.6.

Даталогическая модель представлена в виде схем объектов документоориентированной БД. UserSchema описывает структуру документа «пользователь»:

1. Поле username. Строковый тип данных. Содержит имя пользователя. Наложены ограничения, не позволяющие задавать повторяющиеся (unique) и пустые (required) значения.
2. Поле password. Содержит хеш пароля (в целях безопасности). Пустое значение не допускается.
3. Поле access-token. Дополнительное строковое поле, в котором содержится токен, сгенерированный после успешной аутентификации пользователем. Токен используется для авторизации пользователя при совершении операций над заметками.
4. Поле notes. Имеет тип данных «массив notes». Содержит в себе массив документов – заметок пользователя.

```
var noteSchema = new mongoose.Schema({
  heading: String,
  text: String
});

var userSchema = new mongoose.Schema({
  username: {type: String, required: true, unique: true},
  password: {type: String, required: true},
  access_token: String,
  notes: [noteSchema]
}, { usePushEach: true });
```

Рисунок 4.6. *Даталогическая модель предметной области «Сервис облачных заметок» (составлено авторами)*

NoteSchema содержит два строковых поля – heading (заголовок) и text (основной текст).

4.4 Проектирование API

API для клиентских приложений спроектировано с учетом архитектурного стиля REST (англ. Representation State Transfer – передача состояния представления).

Это определенный набор рекомендаций для проектирования API, разработанный Роем Филдингом [29], который не закреплен в виде стандартов.

Основные элементы концепции, выдвинутые Филдингом:

1. Клиент-серверная модель взаимодействия.
2. На сервере отсутствует информация о состоянии клиентов между запросами.
3. Применение кеширования информации на стороне клиента.

4. Стандартизированный интерфейс. К этому пункту относятся правила идентификации ресурсов и правила манипуляции ресурсами через представление.
5. Многослойная архитектура информационной системы.

Разработчики API с использованием концепции REST используют следующий стандартизированный интерфейс:

1. HTTP в качестве протокола обмена информацией между клиентом и сервером.
2. JSON или XML в качестве формата данных, передаваемых в теле HTTP запроса или ответа.
3. Доступ к каждому ресурсу осуществляется через отдельный адрес. Например, example.com/book/1 обеспечивает доступ к первой книге.
4. HTTP-метод указывает на действие, которое необходимо произвести с ресурсом по заданному адресу. Метод GET возвращает объект, POST – записывает изменения, метод PUSH добавляет новый экземпляр объекта или заменяет текущий, DELETE – удаляет.

Реализация API приложения с использованием HTTP включает в себя следующие методы (формат передаваемых данных – JSON):

1. PUSH example.com/register – регистрация нового пользователя. В теле запроса указываются имя пользователя (username) и пароль (password).
2. GET example.com/auth – аутентификация пользователя. В теле запроса указываются имя пользователя и пароль. В ответе, в случае успешной аутентификации, возвращается access_token, который необходимо вставлять в заголовок запросов методов, указанных ниже (в поле x-access-token).
3. GET example.com/notes – возвращает список всех заметок пользователя.
4. PUSH example.com/notes – добавляет новую заметку. В теле запроса передается структура заметки с полями heading (заголовок) и text (основной текст).
5. POST example.com/notes/id, где id – идентификатор существующей заметки пользователя. Редактирование заметки с заданным идентификатором. В теле запроса передаются те же данные, что и в пункте 4.
6. DELETE example.com/notes/id, где id – идентификатор существующей заметки пользователя. Удаление заметки с заданным идентификатором.

Так как в WebSocket невозможно вставлять свою информацию в заголовок пакета, было решено разработать свой простой протокол обмена данными. Протокол представляет из себя обмен данными в виде JSON-структур, имеющих следующие поля:

1. headers – словарь со строковыми ключами и значениями. Имитирует функционал MIME-заголовков HTTP. Опциональное поле и не обязательное для заполнения.
2. parameters – словарь со строковыми ключами и значениями. Имитирует функционал GET-параметров и других данных, передаваемых через URL. Необязательно для заполнения.
3. data – поле, в котором размещается массив любых объектов. В этом поле размещается полезная нагрузка при необходимости.

Основная цель разработанного протокола является использование кода, написанного для HTTP REST API без изменений. Реализован класс, который «оборачивает» обработчики HTTP-запросов и использует их в качестве обработчиков запросов, пришедших по WebSocket.

4.5 Реализация клиентского и серверного приложений

Серверное приложение написано на языке JavaScript. Исходный код серверного приложения занимает 595 строк.

Серверное приложение, без осуществления изменений в коде, может одновременно осуществлять прием подключений по протоколам HTTP/1.1, HTTP/2, HTTPS, WebSocket, Secure WebSocket. Выбор метода кеширования осуществляется во время запуска серверного скрипта, путем указания входного параметра командной строки.

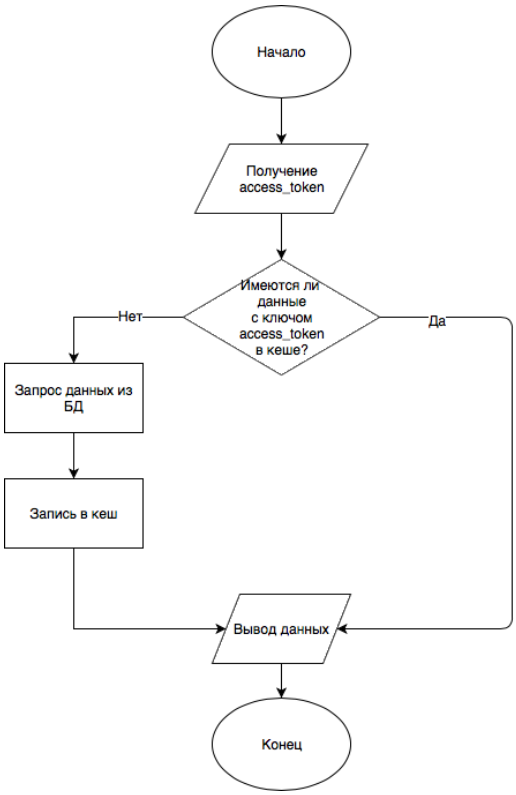
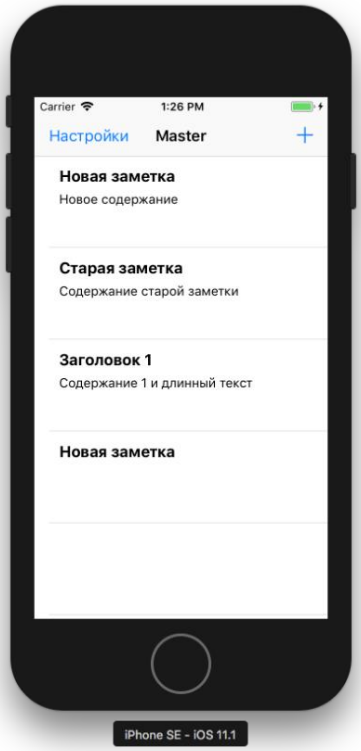


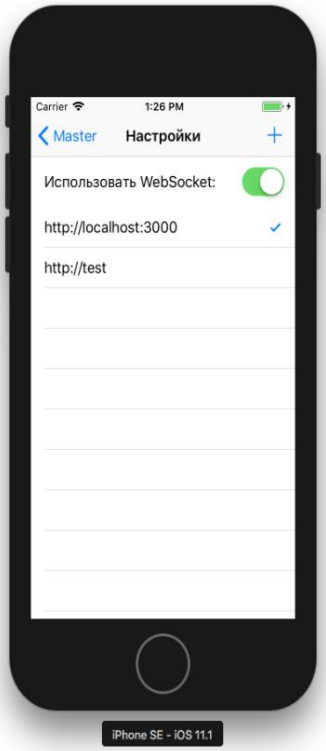
Рисунок 4.7. *Общий алгоритм кеширования сессий (составлено авторами)*

Как для случая использования Redis, так и в случае использования memcached, использован одинаковый алгоритм доступа к сессиям (схема представлена на рисунке 4.7). Когда memcached и Redis не используются, данные всегда запрашиваются из основной базы данных.

Клиентское приложение написано на языке Swift. Приложение содержит 900 строк кода. Возможность выбора адреса сервера и протокола предоставлена в настройках приложения (рисунок 4.8б). Переключение на версии протоколов с использованием TLS-туннеля происходит путем использования в URL схем «wss://» и «https://».



а)



б)

Рисунок 4.8. *Скриншоты iOS приложения Notes (составлено авторами)*

Для реализации обмена данных по протоколам HTTP/1.1 и HTTP/2 использована стандартная библиотека iOS – Foundation. В iOS версии 11.3, на которой производилось тестирование, не имеется встроенной поддержки WebSocket., поэтому была использована сторонняя библиотека SwiftWebSocket, в которой протокол реализован согласно RFC 6455.

4.6 Тестирование межпроцессного обмена между серверами приложений

4.6.1 Методика тестирования

Тестирование межпроцессного между серверами приложений взаимодействия производилось с помощью ApacheBench. Это консольная утилита, предназначенная для нагрузочного тестирования HTTP-серверов. Утилита в качестве входных параметров может принимать адрес тестируемого узла, HTTP-заголовки, тело запроса, количество запросов, количество потоков запросов.

Тестирование производилось со следующими параметрами:

1. Количество запросов – 1000.
2. Количество потоков – 100.
3. Адрес – http://localhost:8080/list. По данному адресу возвращается список заметок пользователя (при условии верного access-token).
4. Заголовки запроса – «Content-Type: application/json; x-access-token: token», где token – валидный access-token одного из пользователей с несколькими заметками.

Тестирование производилось на минимальном количестве сторонних процессов. При каждом тестировании количество сторонних процессов в системе контролировалось.

Инфраструктура, имитирующая распределенную вычислительную систему, развернута на базе ПО Docker, представляющее из себя средства виртуализации на уровне ОС. Каждый процесс (процессы Node.js, Redis, memcached, MongoDB, балансировщик нагрузки на nginx) был запущен в рамках отдельного контейнера Docker. На каждый контейнер было выделено не менее 1 логического ядра процессора. К nginx предоставлен доступ из ОС хоста Docker. Все остальные процессы взаимодействуют друг с другом, используя виртуальный сетевой интерфейс Docker. ApacheBench запускался из ОС хоста.

Тестирование повторялось 6 раз для каждого метода хранения сессий. Брались последние 5 результатов. Делалось это с целью уменьшения влияния механизмов macOS, которые, в целях оптимизации, вводят в спящий режим фоновые процессы после некоторого времени неактивности.

Версии использованного ПО:

1. macOS: 10.13.4.
2. Docker Community Edition: 18.03.1.
3. NodeJS: 10.1.0.
4. MongoDB: 3.6.4.
5. memcached: 1.5.7.
6. nginx: 1.13.12.
7. Redis: 4.0.9.
8. body-parser: 1.18.2.
9. express: 4.15.4.
10. mongoose: 4.8.7.
11. websocket: 1.0.26.
12. bcrypt: 2.0.1.
13. memjs: 1.2.0.
14. redis: 2.8.0.
15. spdy: 3.4.7.
16. Apache Bench: 2.3.

4.6.2 Результаты тестирования

В таблице 4.1 представлены результаты каждой итерации тестирования. Рисунок 4.9 предоставляет графическую интерпретацию результатов.

Также были посчитаны средние значения результатов тестирования для каждого метода хранения сессий. Значения приведены в виде таблицы (таблица 4.2) и в виде графика (рисунок 4.10).

Таблица 4.1

Результаты исполнения теста

Номер замера/название измерения	1	2	3	4	5
Время исполнения теста без кеширования сессий, с	4,539	3,761	3,851	3,700	3,419
Время исполнения теста с использованием memcached, с	2,704	2,534	2,734	2,857	2,878
Время исполнения теста с использованием Redis, с	2,932	2,736	2,783	2,732	2,607

Составлено авторами

Таблица 4.2

Средние значения всех итераций теста

	Среднее время исполнения одного запроса, мс
Без кеширования	3,9504
memcached	3,3654
Redis	3,3448

Составлено авторами

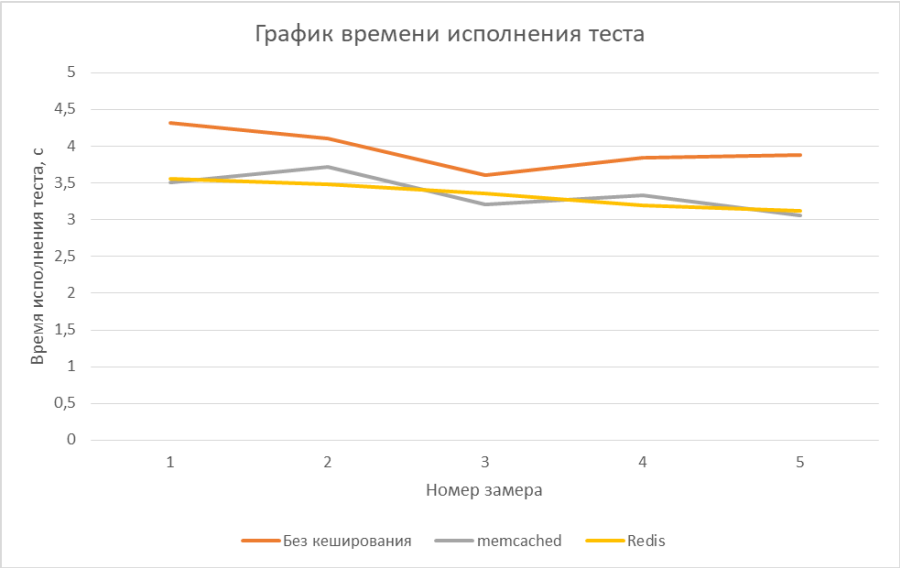


Рисунок 4.9. *График времени исполнения теста (составлено авторами)*

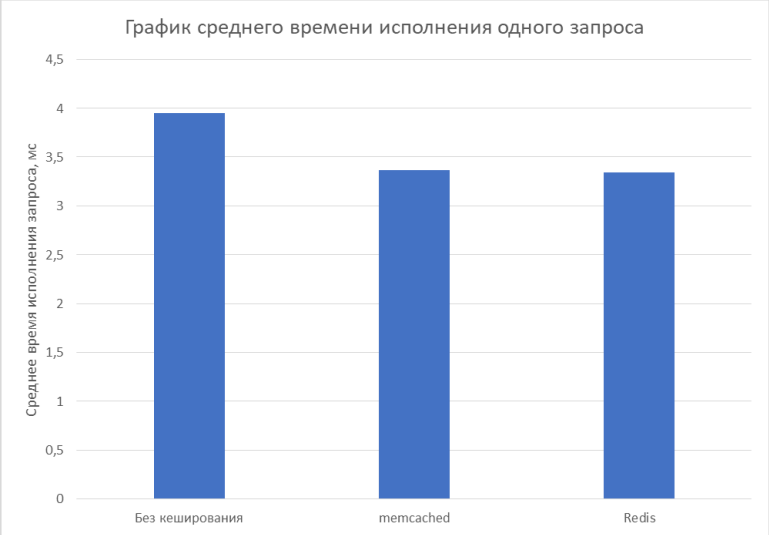


Рисунок 4.10. *График среднего времени исполнения теста (составлено авторами)*

Redis и memcached показали схожий результат (разница составляет менее 1 %). Хранение сессий в основной базе данных оказалось медленнее других способов на ~15 %. Этот

результат оказался намного меньше, чем ожидалось от систем хранения данных в ОЗУ, но стоит учитывать факт, что испытания производились с использованием разгруженной БД. В реальных высоконагруженных проектах, БД загружена обработкой запросов, связанных с бизнес-логикой приложения, поэтому время отклика может быть в разы меньше.

Тем не менее, хранение сессий в основной БД, особенно если это NoSQL решение MongoDB, оправдано для малонагруженных проектов.

4.7 Тестирование межпроцессного обмена между сервером и клиентом

4.7.1 Методика тестирования

Тестирование способов взаимодействия между клиентом и сервером осуществлялось на симуляторе iPhone, запущенном на том же компьютере, где исполнялся процесс сервера приложений. Делалось это с целью уменьшения влияния беспроводных сетей на итоговые результаты [15].

Измерения производились инструментом XCTest, который входит в стандартный комплект поставки Xcode. Помимо юнит-тестирования, инструмент может измерять время исполнения кода программы. XCTest проводит каждое тестирование 10 раз и берет среднее значение. Имеется поддержка асинхронных функций, что важно для обеспечения тестирования сетевых функций приложения.

Каждый замер XCTest для каждого из способов соединения сервера с клиентом повторялся 6 раз. Брались последние 5 значений.

Соединение между сервером и клиентом при использовании WebSocket и Secure WebSocket устанавливалось до начала измерений. Подразумевается, что при реальном использовании приложения соединение между клиентом и сервером будет постоянным в случае WebSocket.

Версии использованного ПО (помимо указанных в предыдущем разделе):

1. iOS: 11.3.
2. Xcode: 9.3.
3. Simulator: 10.0.

4.7.2 Результаты тестирования

В таблице 4.3 приведены результаты всех итераций тестирования. Рисунок 4.11 содержит графическую интерпретацию результатов. В таблице 4.4 и рисунке 4.12 приведены усредненные результаты всех итераций тестирования.

Таблица 4.3

Результаты тестирования

Название измерения/номер измерения	1	2	3	4	5
Среднее время исполнения запроса HTTP, мс	35	25	26	21	33
Среднее время исполнения запроса HTTPS, мс	115	119	118	128	112
Среднее время исполнения запроса HTTP/2, мс	122	121	122	131	115
Среднее время исполнения запроса WebSocket, мс	10	11	10	11	8
Среднее время исполнения запроса Secure WebSocket, мс	11	8	8	7	9

Составлено авторами

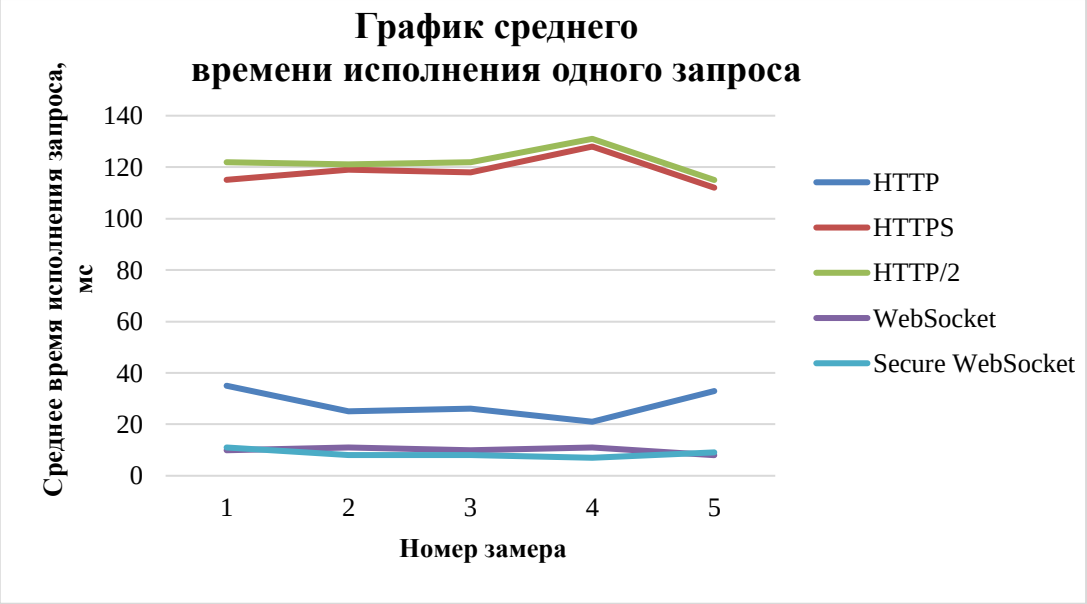


Рисунок 4.11. *График значения всех итераций тестирования (составлено авторами)*

Таблица 4.4

Средние значения измерений

Тип соединения	Среднее время исполнения запроса, мс.
HTTP	28
HTTPS	118,4
HTTP/2	122,2
WebSocket	10
Secure WebSocket	8,6

Составлено авторами

WebSocket с использованием небезопасного соединения оказался в 2,8 раза быстрее HTTP. Безопасная версия WebSocket, вопреки ожиданиям, оказалась на 14 % быстрее, чем аналог без использования TLS. HTTPS оказался более, чем в 13 раз медленнее безопасной версии WebSocket. Связано это, скорее всего, из-за многократной процедуры установления соединения TLS (в случае с WebSocket, постоянное соединение уже установлено на момент тестирования). HTTP/2 по уровню производительности оказался близок версии HTTP/1.1 с использованием защищенного соединения.

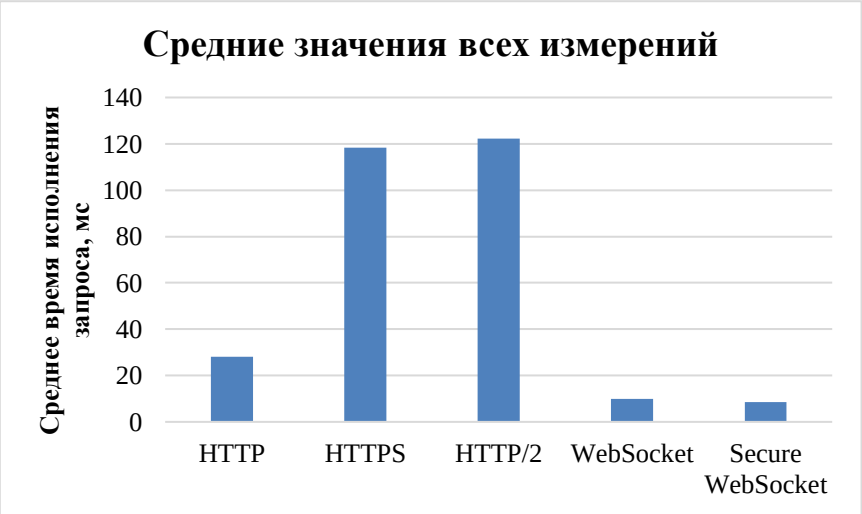


Рисунок 4.12. *График средних значений результатов тестирования (составлено авторами)*

Дополнительно было решено исследовать размеры пакетов, передаваемых по протоколам HTTP и WebSocket. Для этого использовалось ПО WireShark 2.4.4. Результаты представлены в таблице 4.5.

Таблица 4.5

Размеры пакетов, передаваемых по протоколам HTTP и WebSocket

	WebSocket	HTTP
Размер запроса, байт	218	462
Размер ответа, байт	416	232 (кеш-попадание) 600 (кеш-промах)

Составлено авторами

API, реализованное на WebSocket, имеет более чем в 2 раза меньший размер запросов. Размер ответов WebSocket больше в случае, если запрашивается статические данные, которые не изменялись с прошлого запроса (это обусловлено тем, что в HTTP есть встроенный механизм кеширования). Однако, если данные динамические, WebSocket имеет преимущество в 44 %.

Заключение

В данной работе решены следующие задачи:

1. Обозначена важность грамотного проектирования межпроцессного взаимодействия на всех уровнях трехзвенной архитектуры.
2. Перечислены варианты организации межпроцессного обмена между серверами приложений и между клиентом и сервером.
3. Спроектировано клиент-серверное приложение, основывающееся на трехзвенной архитектуре с использованием нескольких вариантов обеспечения межпроцессного обмена.
4. Реализовано RESTful API на базе протокола WebSocket.
5. Произведено тестирование приложения с использованием разных методов обмена информацией о сессиях и клиент-серверного взаимодействия.

В ходе исследования, были выявлены способы ускорить общее время отклика в информационной системе с горизонтальным масштабированием. В дополнение, было реализовано клиент-серверное приложение для тестирования методов взаимодействия, функционал которого приближен к функционалу реальных приложений, с целью оценить, насколько применение тех или иных методов ощутимы с точки зрения пользователя.

Несмотря на то, что методика тестирования не является совершенной, полученные в результате исследований результаты можно использовать при проектировании новых приложений. В дальнейшем, планируется внесение изменений в методику тестирования, расширение списка используемых инструментов и публикация новых результатов в тематические Интернет-ресурсы.

ЛИТЕРАТУРА

1. Городничев М.Г. Методы проектирования и разработки клиент серверных приложений. Технологии информационного общества Международная отраслевая научно-техническая конференция: сборник трудов. 2017. С. 439-440.
2. Таненбаум Э., ван Стеен М. Распределенные системы. Принципы и парадигмы. СПб: Питер.
3. Grashoff K., Heemskerk, B., Usta B., Vonk M. Telegram-Web.
4. Буслаев А.П., Яшина М.В., Городничев М.Г. SSHD-мониторинг многополосного движения и автоматизация обработки информации о трафике. Учебное пособие. МТУСИ. Москва. 2012.
5. Таненбаум Э., Уэзролл, Д. Компьютерные сети. СПб: Питер.
6. Косяков М. Введение в распределенные вычисления. СПб: НИУ ИТМО.
7. Ласницкая М. Бесконечный интернет подошел к концу. Москва.
8. Nelson J. Mastering Redis. Birmingham, UK: PACKT Publishing.
9. Bartenev V.V. The HTTP/2 Module in NGINX. San Francisco.
10. Деарт В.Ю., Маньков, В.А., Пилюгин, А.В. Статистические характеристики трафика современного провайдера доступа в Интернет. T-Comm.
11. Деарт В.Ю. Мультисервисные сети связи. Москва: Инсвязьиздат.
12. Fette, A., Melnikov A. RFC 6455 – The WebSocket Protocol.
13. Kaushal P. MongoDB Schema Design.
14. Карпова Т. Базы данных: модели, разработка, реализация. Интуит.
15. Buslaev A.P., Gorodnichev M.G., Provorov A.V. One-dimensional models of particles flow and infocommunication methods of verification. Proceedings – 2014 International Conference on Computational Science and Computational Intelligence, CSCI 2014 2014. С. 253-256.

Gorodnichev Mikhail Gennad′evich
Moscow technical university of communications and informatics, Moscow, Russia
E-mail: Gorodnichev89@yandex.ru

Kochupalov Alexander Evgenyevich
Moscow technical university of communications and informatics, Moscow, Russia
E-mail: chupik1994@yandex.ru

Investigation of interprocess communication methods in the information system with horizontal interaction

Abstract. With the advent of the Internet, distributed computing environment (DCE) began to develop. Currently, client-server is the most common DCE architecture. This architecture allows to transfer the computation load to the server, while the client only displays the result. When interacting with other resources, regardless of the type of servers, in scalable projects, due to the presence of a large number of processes running on different processors or servers, there is a problem of data exchange between processes. The authors describe the methods of designing client-server applications based on a III-tier architecture using various variants of interprocess communication. This article discusses methods of interprocess communication in projects with client-server architecture with horizontal scaling. The authors carried out a complex analysis of the problems arising in the interprocess interaction, and suggested methods of solution. In this paper, we study methods for exchanging data between server processes, and between client and server processes. The authors developed test systems procedure, based on the main paradigms of the DCE, and modern technologies. Proceeding from the results, the authors outlined the importance of designing interprocess communication at all levels of the III-tier architecture.

Keywords: interprocess communication (IPC); client/server (C/S); data synchronization; high-load systems; data caching