Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра математической
кибернетики и компьютерных наук

РАЗРАБОТКА ВЫСОКОНАГРУЖЕННЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVASCRIPT.

БАКАЛАВРСКАЯ РАБОТА

Студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Низамутдинова Артура Салаватовича

Научный руководитель
доцент _____ И. А. Борзов

Заведующий кафедрой
к.ф.-м.н. _____ С. В. Миронов

Саратов 2016

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В 2009 году Райан Дал после двух лет экспериментирования над созданием серверных веб-компонентов разработал NodeJS.

NodeJS является программной платформой, основанной на разработанном компанией Google JavaScript — движке V8 (транслирующем JavaScript в машинный код), преобразующий JavaScript из узкоспециализированного в язык общего назначения. NodeJS используется преимущественно на сервере, выполняя роль веб-сервера, но кроме этого есть возможность разрабатывать на NodeJS и десктопные приложения (с использованием NW.js, Electron или AppJS для Windows, Linux и Mac OS) и даже программировать микроконтроллеры (например, espruino и tessel).

В основе Node.js лежит событийно-ориентированное и асинхронное программирование с неблокирующим вводом/выводом.

Поставленные задачи:

- изучить архитектуру NodeJS и NGINX;
- рассмотреть способы асинхронного обмена данными с сервером с использованием websocket, comet, iframe и jsonp;
- реализовать веб-сервер на websocket и comet;
- настроить NGINX как прокси-сервер и балансировщик нагрузки.

1 Краткое описание архитектуры NodeJS

В основе NodeJS лежит выполнение приложения в одном программном потоке, а также асинхронная обработка всех событий. При запуске NodeJS-приложения создается единственный программный поток. NodeJS-приложение выполняется в этом потоке в ожидании, что некое приложение сделает запрос. Когда NodeJS-приложение получает запрос, то никакие другие запросы не обрабатываются до тех пор, пока не завершится обработка текущего запроса.

На первый взгляд, все это кажется не очень эффективным, если бы не то обстоятельство, что NodeJS работает в асинхронном режиме, используя цикл обработки событий и функции обратного вызова. Цикл обработки событий просто опрашивает конкретные события и в нужное время вызывает обработчики событий. В NodeJS таким обработчиком событий является функция обратного вызова.

В отличие от других однопоточных приложений, когда к NodeJS - приложению делается запрос, оно должно, в свою очередь, запросить какие-то ресурсы (например, получить доступ к файлу или обратиться к базе данных). В этом случае NodeJS инициирует запрос но не ожидает ответа на этот запрос. Вместо этого запросу назначается некая функция обратного вызова. Когда запрошенное значение будет готово (или завершено) генерируется событие, активизирующее соответствующую функцию обратного вызова, призванную что-то сделать либо с результатами запрошенного действия, либо с запрошенными ресурсами.

Если несколько человек обращаются к NodeJS-приложению в одно и то же время и приложению нужно обратиться к ресурсам из файла, для каждого запроса NodeJS назначает свою функцию обратного вызова событию ответа. Когда для каждого из них ресурс становится доступен, вызывается нужная функция обратного вызова, и запрос удовлетворяется. В промежутке NodeJS-приложение может обрабатывать другие запросы либо для того же приложения, либо для какого-нибудь другого.

Хотя приложение не обрабатывает запросы в параллельном режиме, в зависимости от своей загруженности и конструкции можно даже не заметить задержки в ответе. А что лучше всего, приложение очень экономно относится к памяти и к другим ограниченным ресурсам. [1]

2 Краткое описание архитектуры NGINX

NGINX (сокращение от engine x) — это HTTP-сервер и обратный прокси-сервер, почтовый, а также TCP/UDP прокси-сервер общего назначения, изначально написанный Игорем Сысоевым.

Для лучшего представления устройства, сперва необходимо понять как NGINX запускается. У NGINX есть один мастер-процесс (который от имени суперпользователя выполняет такие операции, как открытие портов и чтение конфигурации), а также некоторое количество рабочих и вспомогательных процессов. Например, на 4-х ядерном сервере мастер-процесс NGINX создает 4 рабочих процесса и пару вспомогательных кэш-процессов, которые в свою очередь управляют содержимым кэша на жестком диске.

Говоря о многопоточности важно отметить, что любой процесс или поток — это набор самодостаточных инструкций, который операционная система может запланировать для выполнения на ядре процессора. Большинство сложных приложений параллельно запускают множество процессов или потоков по двум причинам:

1. Чтобы одновременно задействовать больше вычислительных ядер;
2. Процессы и потоки позволяют проще выполнять параллельные операции (например работать с множеством соединений одновременно).

2.1 Несколько слов о важности архитектуры

Процессы и потоки сами по себе расходуют дополнительные ресурсы. Каждый из процессов или потоков потребляет некоторое количество памяти, и кроме того постоянно подменяют друг друга на процессоре (так называемое переключение контекста). Современные серверы могут справляться с сотнями активных процессов и потоков, но производительность сильно падает, как только заканчивается память или огромное количество операций ввода-вывода приводит к слишком частой смене контекста.

Наиболее типичный подход к построению сетевого приложения — это выделять для каждого соединения отдельный процесс или поток. Такая архитектура действительно проста для понимания и легка в реализации, но при этом плохо масштабируется, когда приложению приходится работать с тысячами соединений одновременно. [2]

2.2 Основные моменты работы NGINX

NGINX использует модель с фиксированным числом процессов, которая наиболее эффективно задействует доступные ресурсы системы:

- Единственный мастер-процесс выполняет операции, которые требуют повышенных прав, такие, как чтение конфигурации и открытие портов, а затем порождает небольшое число дочерних процессов (следующие три типа).
- Загрузчик кэша запускается на старте чтобы загрузить данные кэша, расположенные на диске, в оперативную память, а затем завершается. Его работа спланирована так, чтобы не потреблять много ресурсов.
- Кэш-менеджер просыпается периодически и удаляет объекты кэша с жесткого диска, чтобы поддерживать его объем в рамках заданного ограничения.
- Рабочие процессы выполняют всю работу. Они обрабатывают сетевые соединения, читают данные с диска и пишут на диск, общаются с бэкенд-серверами. [2]

2.3 Внутри рабочего процесса

Каждый рабочий процесс NGINX инициализируется с заданной конфигурацией и набором слушающих сокетов, унаследованных от мастер-процесса.

Рабочие процессы начинают с ожидания событий на слушающих сокетах. События извещают о новых соединениях. Эти соединения попадают в конечный автомат — наиболее часто используемый предназначен для обработки HTTP, но NGINX также содержит конечные автоматы для обработки потоков TCP трафика (модуль stream) и целого ряда протоколов электронной почты (SMTP, IMAP и POP3).

Конечный автомат в NGINX по своей сути является набором инструкций для обработки запроса. Большинство веб-серверов выполняют такую же функцию, но разница кроется в реализации. [2]

2.4 Устройство конечного автомата

Конечный автомат можно представить себе в виде правил для игры в шахматы. Каждая HTTP транзакция — это шахматная партия. С одной стороны шахматной доски веб-сервер — гроссмейстер, который принимает решения

очень быстро. На другой стороне — удаленный клиент, браузер, который запрашивает сайт или приложение по относительно медленной сети.

Как бы то ни было, правила игры могут быть очень сложными. Например, веб-серверу может потребоваться взаимодействовать с другими ресурсами (проксировать запросы на бэкенд) или обращаться к серверу аутентификации. Сторонние модули способны ещё сильнее усложнить обработку. [2]

2.5 Блокирующийся конечный автомат

Вспомните наше определение процесса или потока, как самодостаточного набора инструкций, выполнение которых операционная система может назначать на конкретное ядро процессора. Большинство веб-серверов и веб-приложений используют модель, в которой для «игры в шахматы» приходится по одному процессу или потоку на соединение. Каждый процесс или поток содержит инструкции, чтобы сыграть одну партию до конца. Все это время процесс, выполняясь на сервере, проводит большую часть времени заблокированным в ожидании следующего хода от клиента.

1. Процесс веб-сервера ожидает новых соединений (новых партий инициированных клиентами) на слушающих сокетах.
2. Получив новое соединение, он играет партию, блокируясь после каждого хода в ожидании ответа от клиента.
3. Когда партия сыграна, процесс веб-сервера может находиться в ожидании желания клиента начать следующую партию (это соответствует долгоживущим keepalive-соединениям). Если соединение закрыто (клиент ушел или наступил таймаут), процесс возвращается к встрече новых клиентов на слушающих сокетах.

Важный момент, который стоит отметить, заключается в том, что каждое активное HTTP-соединение (каждая партия) требует отдельного процесса или потока (гроссмейстера). Такая архитектура проста и легко расширяема с помощью сторонних модулей (новых «правил»). Однако, в ней существует огромный дисбаланс: достаточно легкое HTTP-соединение, представленное в виде файлового дескриптора и небольшого объема памяти, соотносится с отдельным процессом или потоком, достаточно тяжелым объектом в операционной системе. Это удобно для программирования, но весьма расточительно. [2]

3 Способы асинхронного обмена данными с сервером

В современном Web асинхронный обмен данными с сервером является практически его неотъемлемой частью. Подобный обмен данными позволяет без перезагрузки страницы клиентского приложения обмениваться различной информацией с сервером, что в свою очередь позволяет как повысить интерактивность web-приложений, так и дает возможность обмена данными в режиме реального времени с сервером.

В далее качестве примеров рассмотрим способы обмена данными с использованием websocket, comet, iframe и jsonp.

3.1 WebSocket

WebSocket протокол был утвержден в качестве стандарта RFC 6455 в декабре 2011 года. Данный тип соединения предоставляет двунаправленное полнодуплексное соединение. С помощью WebSocket можно создавать интерактивные браузерные веб-приложения, которые постоянно обмениваются данными с сервером, но при этом не нуждаются в открытии нескольких HTTP-соединений.

Хоть и WebSocket использует HTTP как основной механизм для передачи данных, однако канал связи не закрывается после получения данных клиентом. Используя WebSocket API вы полностью свободны от ограничений типичного цикла HTTP (request/responce). Это также означает, что до тех пор пока соединение остается открытым, клиент и сервер могут свободно отправлять данные в асинхронном режиме без опроса для чего-нибудь нового. [3]

Цикл работы WebSocket соединения состоит из следующих этапов:

— установления соединения (opening handshake);
— оформления и отправки данных;
— закрытие соединения (closing handshake).

3.1.1 Установление WebSocket соединения

Поскольку протокол WebSocket работает поверх HTTP, то это означает, что при подключении браузер отправляет следующие специальные заголовки:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
```

```
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Описание заголовков:

`GET, Host` — стандартные HTTP—заголовки из URL запроса.

`Upgrade, Connection` — указывают, что браузер хочет перейти на websocket.

`Origin` — протокол, домен и порт, откуда отправлен запрос.

`Sec-WebSocket-Key` — случайный ключ, который генерируется браузером: 16 байт в кодировке Base64.

`Sec-WebSocket-Version` — версия протокола. Текущая версия: 13.

Все заголовки, кроме `GET` и `Host`, браузер генерирует сам, без возможности вмешательства JavaScript.

Далее сервер, проанализировав эти заголовки, решает, разрешает ли он соединение WebSocket с данного домена `Origin`.

В случае, если сервер разрешает WebSocket подключение, то он возвращает клиенту ответ следующего вида:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

Здесь заголовок `Sec-WebSocket-Accept` представляет собой перекодированный с использование специального алгоритма ключ `Sec-WebSocket-Key`. Браузер в свою очередь использует ее для проверки, что ответ предназначается именно ему.

Затем данные передаются по специальному протоколу, структура которого («фреймы») изложена далее. И это уже совсем не HTTP. [3]

3.1.2 Оформление и передача данных с использованием WebSocket протокола

В протоколе WebSocket предусмотрены несколько видов пакетов («фреймов»).

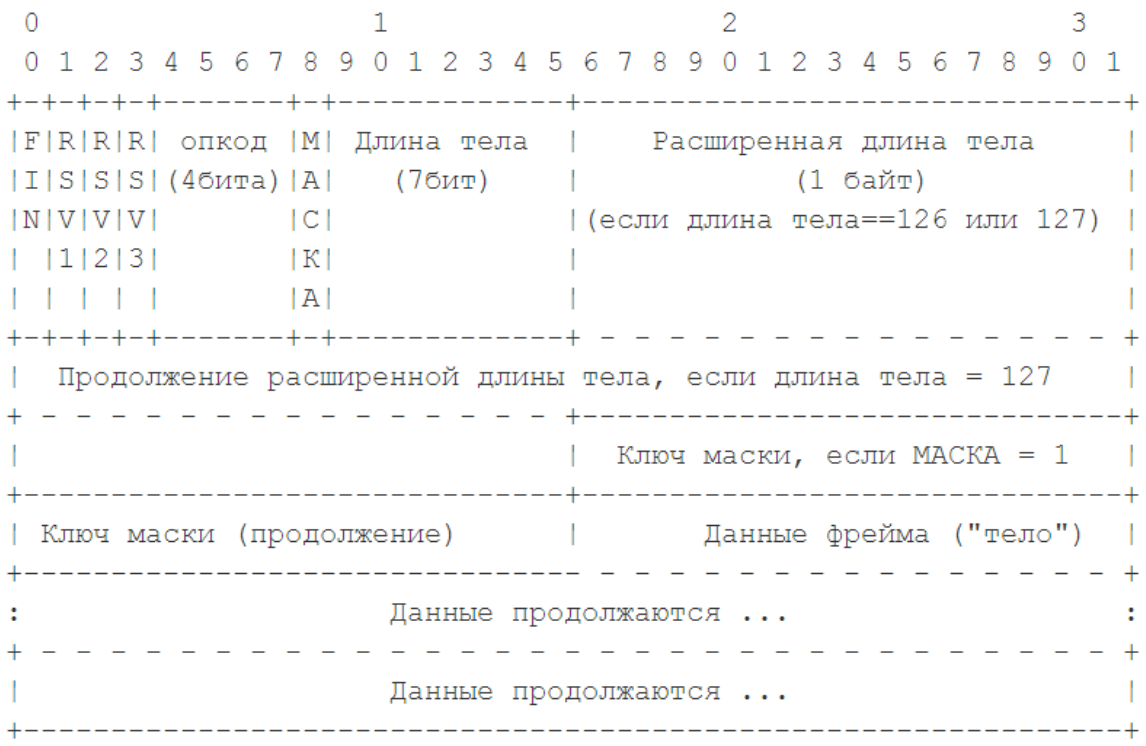

Рисунок 1 – Схема WebSocket фрейма

Они делятся на два больших типа: фреймы с данными («data frames») и управляющие («control frames»), предназначенные для проверки связи (PING) и закрытия соединения.

На рисунке 1 можно увидеть общий вид WebSocket фрейма, согласно его стандарту.

3.1.3 Закрытие WebSocket соединения

Процесс закрытия WebSocket соединения гораздо проще, чем процесс открытия.

Любая из сторон WebSocket соединения может отправить управляющий фрейм с данными содержащими специальную управляющую последовательность для начала процесса закрытия. После отправки управляющего фрейма, указывающего, что соединение должно быть закрыто, другая сторона в дальнейшем отбрасывает любые данные, которые были отправлены. В добавок ко всему выше сказанному, процесс закрытия WebSocket соединения безопасен для одновременной инициализации обеими сторонами. [4]

3.2 Comet

Другое название метода — «Очередь ожидающих запросов».

Основа работы данного метода состоит из следующей последовательности шагов:

— Отправляется запрос на сервер
— Соединение не закрывается сервером пока не появится событие;
— Событие отправляется в ответ на запрос;
— Клиент тут же отправляет новый ожидающий запрос.

Ситуация, когда браузер отправляет запрос и держит соединение с сервером, ожидая ответа, является стандартной и прерывается только доставкой сообщений.

При этом в случае, когда соединение рвется само, к примеру, из-за ошибки в сети, то браузер тут же отсылает новый запрос. [5]

3.3 IFrame

По сути IFrame представляет собой окно браузера, вложенное в основное окно.

3.3.1 Общая схема работы

1. На клиентской стороне создается невидимый IFrame на специальный URL;
2. При наступлении событий на сервере в IFrame тут же поступает тег `<script>` — пакет с данными вида:

```
<script>
parent.handleMessage({txt:"Hello",time:123456789})
</script>
```

3. Соединение закрывается в случаях:
   — при возникновении ошибки;
   — каждые 20—30 секунд;
   — когда требуется очистка памяти от старых сообщений (время от времени создаем новый IFrame и удаляем старый).

На рисунке 2 можно увидеть схему работы IFrame транспорта. [6]

3.4 JSONP

Если попробовать создать тег `<script src>`, то при добавлении его в документ запустится процесс загрузки с данного `src`. В ответ на запрос сервер

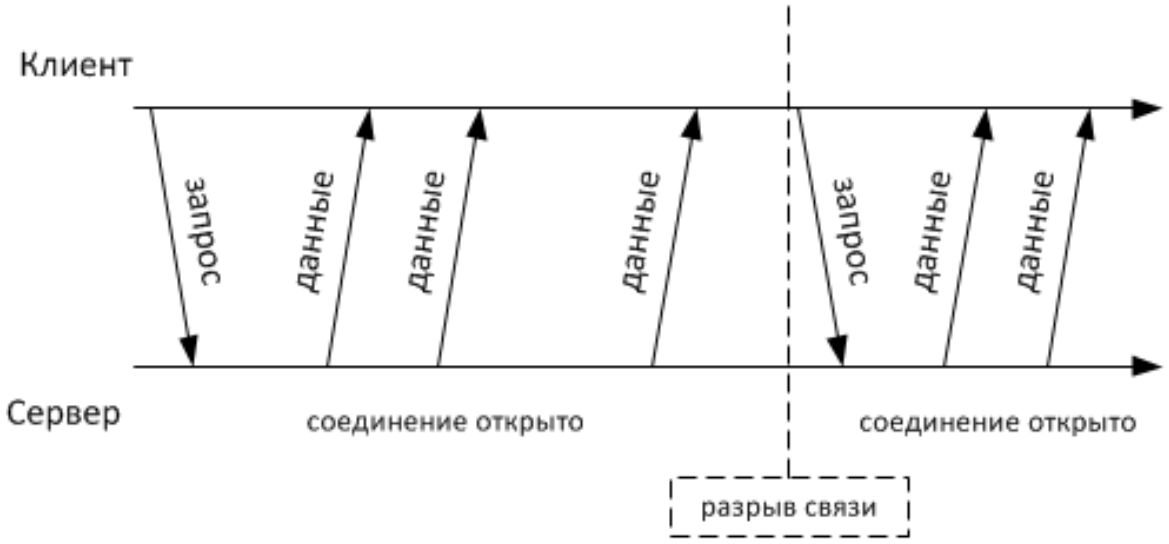

Рисунок 2 – Общая схема работы IFrame транспорта

может прислать скрипт, который будет содержать нужные данные.

С помощью данного способа можно запрашивать данные с любого сервера, в любом браузере, без каких-либо разрешений и дополнительных проверок.

Протокол JSONP — это своего рода надстройка над таким способом коммуникации.

3.4.1 Общая схема работы

1. На клиентской стороне создается тег `<script>` на специальный URL, в котором в качестве параметра передается имя функции обратного вызова, которая будет вызываться при получении данных;
2. В свою очередь сервер формирует ответ в виде вызова этой функции с данными переданными в ней в качестве параметров, и отправляет ответ клиенту;
3. Сразу после того как клиентская сторона получает ответ от сервера, полученный скрипт начинает немедленно выполняться, таким образом вызывая функцию обратного вызова, которая располагается на стороне клиента.

При использовании данного метода необходимо помнить про аспект безопасности, поскольку клиентский код должен доверять серверу при таком способе запроса данных. Ведь серверу ничего не стоит добавить в скрипт любые вредоносные команды.

COMET через протокол JSONP реализуется с использованием длинных запросов, то есть, создается тег `<script>`, браузер запрашивает скрипт у сер-

вера и сервер оставляет соединение висеть, пока не появятся данные, которые необходимо передать клиенту. Когда сервер хочет отправить сообщение — он формирует ответ с использованием формата JSONP, и тут же клиент отправляет новый запрос. [7]

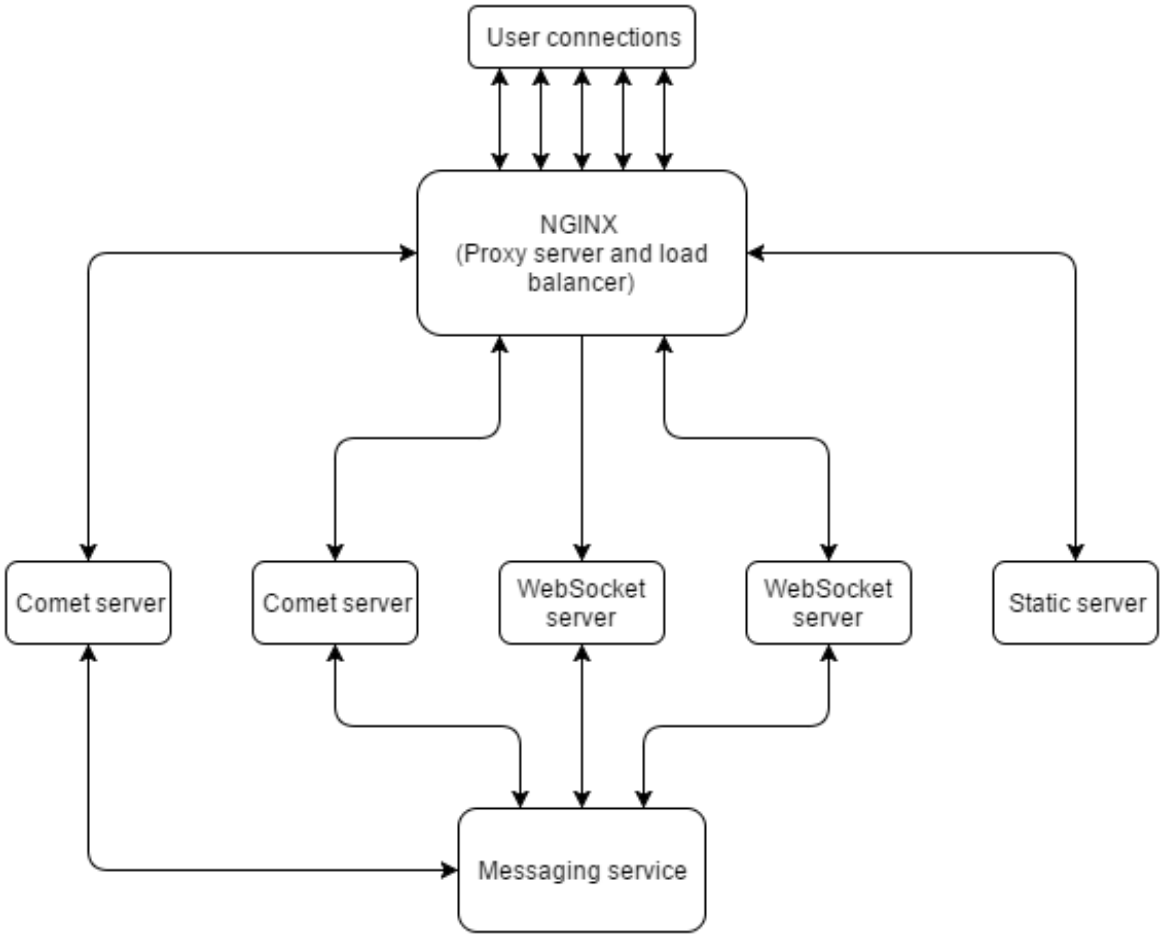

Рисунок 3 – Архитектура приложения

4 Описание архитектуры приложения

Архитектура приложения состоит из следующих компонентов:

— NGINX (прокси-сервер и балансировщик нагрузки)
— 2 WebSocket сервера
— 2 Comet сервера
— Messaging сервис

На рисунке 3 можно увидеть общую схему приложения.

4.1 Цикл работы приложения

После развертывания приложения, пользователи получают UI функционал (html + css + javascript), с помощью которого предоставляется возможность выбрать протокол, через который будет проходить обмен данными с сервером (в данном случае пользователь отправляет координаты своего местоположения с определенным интервалом). В ситуации, если пользователь выбрал WebSocket соединение, то NGINX, используя директиву `least_conn`, перенаправляет запрос на активацию соединения к тому WebSocket серверу,

который в текущий момент имеет наименьшее число подключений. Однако, если пользователь выбрал Comet соединение, то в этом случае NGINX, используя директиву `ip_hash`, создает специальный хеш для ip адреса пользователя, чтобы все последующие запросы шли на изначально определенный Comet сервер. В дальнейшем сообщение переотправляется всем подписчикам данного экземпляра сервера, а также передается в Messaging сервис, который в свою очередь пересылает сообщение другим экземплярам серверов кластера. Далее эти экземпляры пересылают это сообщение для своих подписчиков.

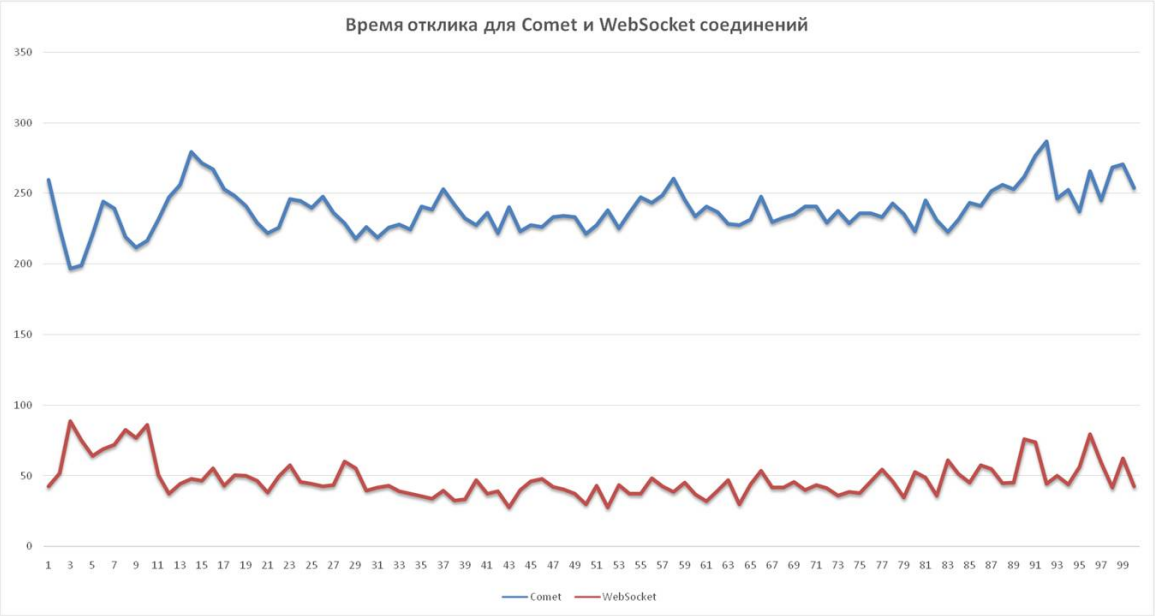

Рисунок 4 – Статистика по Comet и WebSocket соединениям

5 Нагрузочное тестирование Comet и WebSocket соединений

Для тестирования Comet и WebSocket соединений также были созданы 2 программы, которые можно запустить, открыв `gulp-test-comet.bat` или `gulp-test-ws.bat` для тестирования соответствующего типа соединения. Перед тестированием не забудьте открыть `gulp-serve.bat` для запуска приложения. Данные `.bat` файлы содержат команды `gulp test-comet --i 100` и `gulp test-ws --i 100`, запускающие 100 NodeJS процессов, которые в свою очередь подключаются к приложению и начинают получать сообщения от процесса-отправителя (`test-comet-send.js` и `test-ws-send.js` для comet и websocket соединения соответственно). Процесс-отправитель отправляет сообщения длиной 16 символов в кодировке $UTF-8$ с интервалом в несколько миллисекунд. На рисунке 4 можно увидеть усредненное время взятое со 100 экземпляров нагрузочного тестирования соответствующего типа за 100 итераций, выполнявшихся в одно и тоже время. По вертикали указано время в миллисекундах, в свою очередь, по горизонтали номер итерации.

Как можно видеть на графике, время между полученными сообщениями для Comet и WebSocket соединений существенно разнится. Данные результат объясняется тем, что для Comet соединения клиентская сторона после каждого полученного сообщения вынужденна переподписываться для получения следующих сообщений, тем самым снова и снова отправляя ...

6 Выводы

В результате пройденной практики были изучены основы архитектуры NodeJS и NGINX, также были рассмотрены основные способы асинхронного обмена данными между клиентом и сервером. На основе этих данных было создано приложение, позволяющее обмениваться данными между клиентами и сервером в режиме реального времени, а также в достаточной мере устойчивое к высоким нагрузкам (к большому числу одновременно подключенных клиентов).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Сухов, К. К.* Node.js. Путеводитель по технологии / К. К. Сухов. — М.: ДМК Пресс, 2015.

2 *Бартенев, В. В.* Nginx изнутри: рожден для производительности и масштабирования [Электронный ресурс]. — URL: `https://habrahabr.ru/post/260065/` (Дата обращения 25.05.2016) Загл. с экр. Яз. рус.

3 *Адаменко, И. И.* Websocket [Электронный ресурс]. — URL: `https://learn.javascript.ru/websockets` (Дата обращения 21.05.2016) Загл. с экр. Яз. рус.

4 *Мельников, А. А.* The websocket protocol [Электронный ресурс]. — URL: `https://tools.ietf.org/html/rfc6455` (Дата обращения 24.05.2016) Загл. с экр. Яз. англ.

5 *Адаменко, И. И.* Comet с xmlhttprequest: длинные опросы [Электронный ресурс]. — URL: `https://learn.javascript.ru/xhr-longpoll` (Дата обращения 25.05.2016) Загл. с экр. Яз. англ.

6 *Адаменко, И. И.* Iframe для ajax и comet [Электронный ресурс]. — URL: `https://learn.javascript.ru/ajax-iframe` (Дата обращения 26.05.2016) Загл. с экр. Яз. англ.

7 *Адаменко, И. И.* Протокол jsonp [Электронный ресурс]. — URL: `https://learn.javascript.ru/ajax-jsonp` (Дата обращения 25.05.2016) Загл. с экр. Яз. рус.