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.

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

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

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). Также важно отметить, что NodeJS используют в своих проектах такие крупные компании как Google, Yahoo, PayPal, Yammer и многие другие.

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;
- провести нагрузочное тестирование comet и websocket соединений и сделать выводы.

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 Основные моменты работы NGINX

В NGINX используется архитектура с предварительно заданным числом процессов, которая эффективней всего использует имеющиеся системные ресурсы:

- Мастер-процесс запускает команды, требующие повышенных прав доступа, такие как открытие портов и чтение конфигурации, после чего порождает несколько дочерних процессов (следующих трех типов).
- Загрузчик кэша начинает свою работу на старте, для того чтобы загрузить данные кэша, находящиеся на диске, в оперативную память, и затем завершается. Его работа рассчитана таким образом, чтобы потреблять как можно меньше ресурсов.
- Кэш-менеджер периодически активируется, чтобы удалить данные кэша с жесткого диска, таким образом, поддерживая его объем в заранее заданных границах.

— Рабочие процессы выполняют основную часть работы. Они работают с сетевыми соединениями, читая и записывая данные на диск, обмениваются данными с бэкенд-серверами. [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). [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). [6]

На рисунке 1 можно увидеть схему работы IFrame транспорта.

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 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.

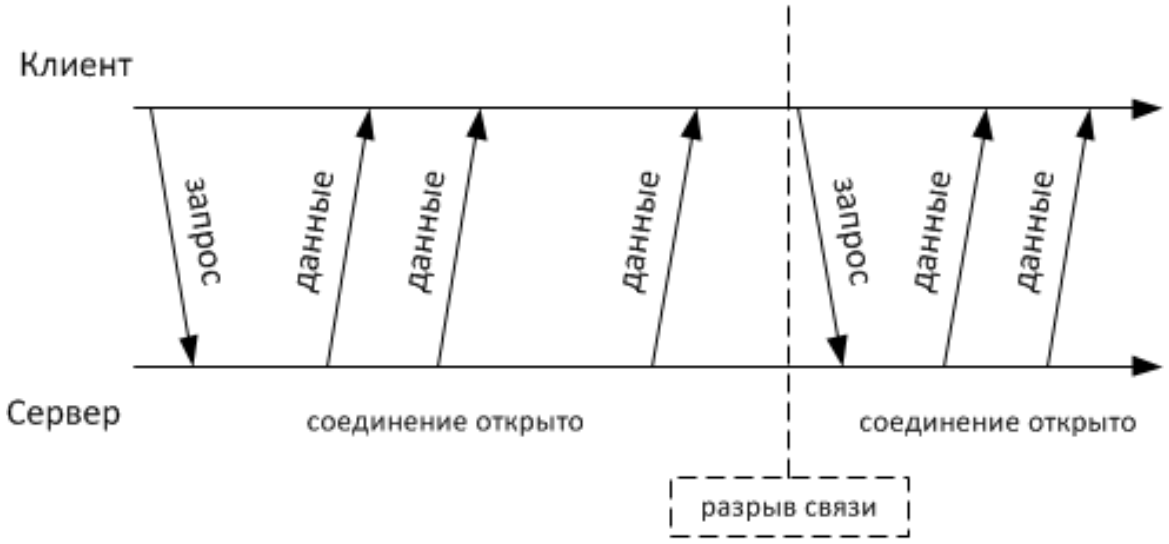

Рисунок 1 – Общая схема работы IFrame транспорта

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 сервера 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]

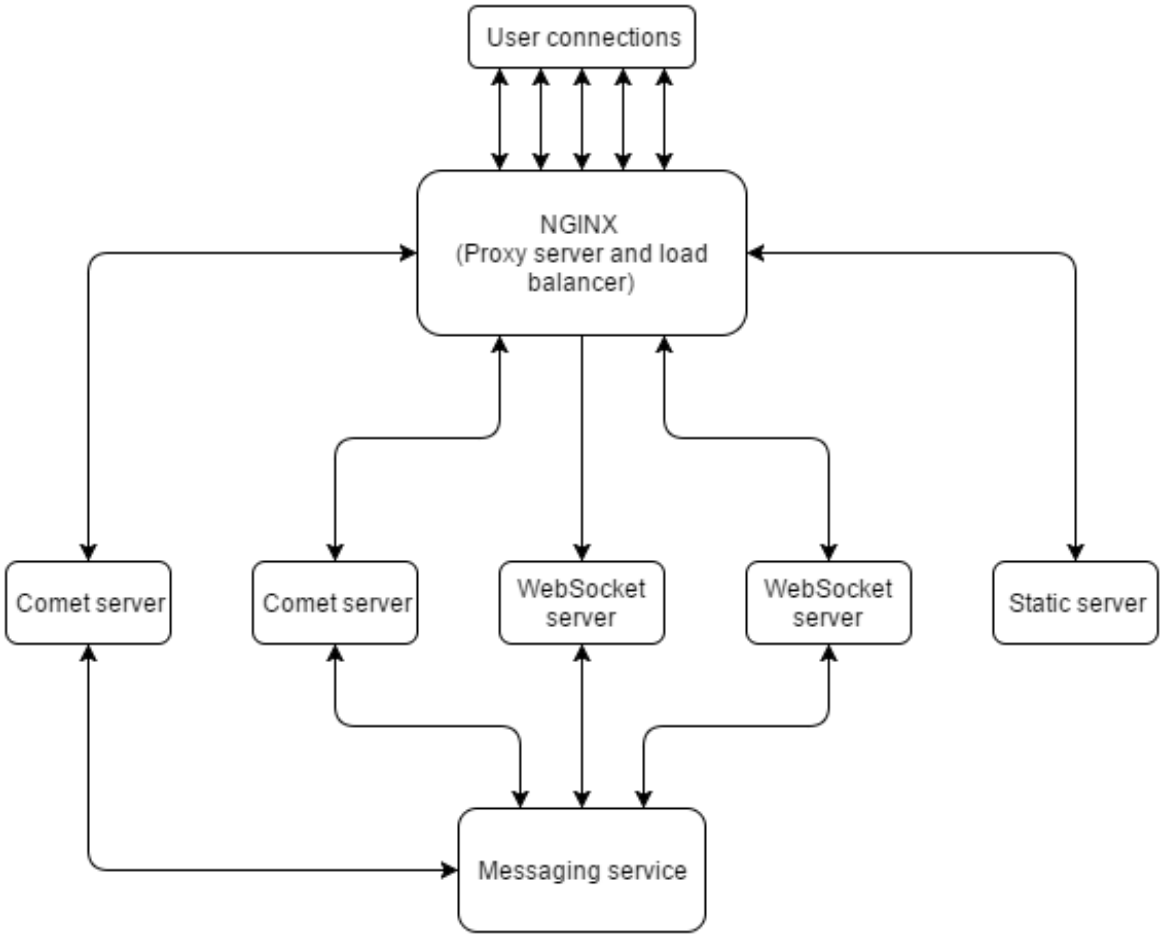

Рисунок 2 – Архитектура приложения

4 Описание архитектуры приложения

Архитектура приложения состоит из следующих компонентов:

- NGINX (прокси-сервер и балансировщик нагрузки)
- 2 WebSocket сервера
- 2 Comet сервера
- Static сервер
- Messaging сервис
- Gulp task runner предназначенный для запуска приложения

На рисунке 2 можно увидеть общую схему приложения.

4.1 Цикл работы приложения

Рабочий процесс приложения происходит в рамках следующих простых этапов:

1. После развертывания приложения, пользователи, в первый раз обращающиеся к нему, получают основной функционал для полноценной работы с приложением (html + css + javascript)

2. Получив интерфейс для работы с приложением, пользователю предоставляется возможность выбрать протокол, через который будет проходить обмен данными с сервером (в данном случае пользователь отправляет координаты своего местоположения с определенным интервалом).
3. 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.
4. 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.
5. 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.

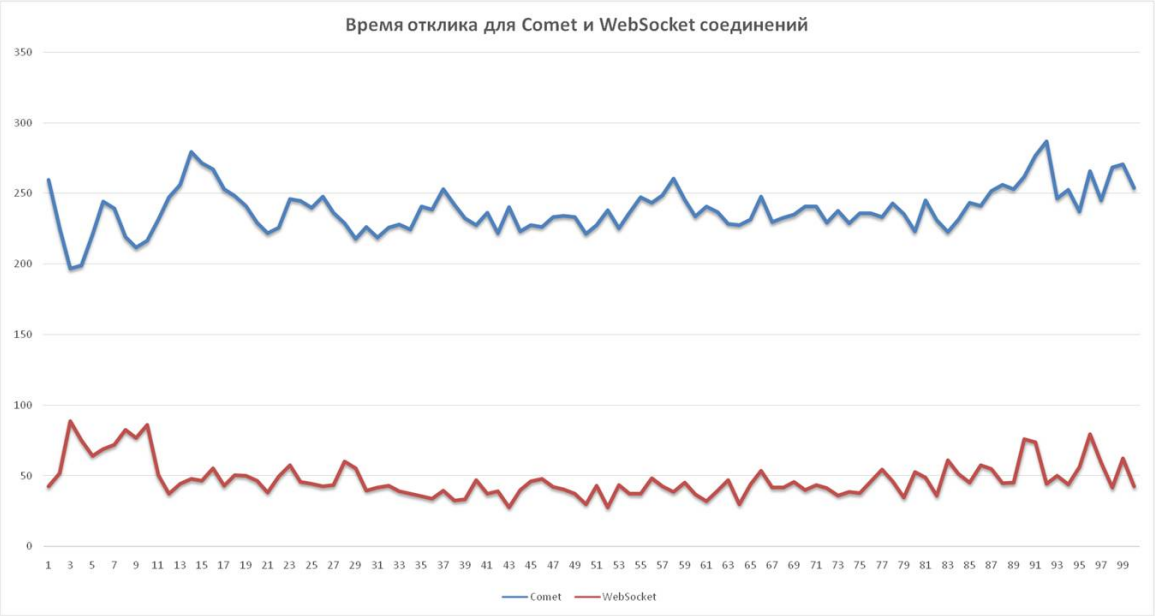

Рисунок 3 – Статистика по Comet и WebSocket соединениям

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 3 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. Данный 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, кроме необходимой информации, основные HTTP заголовки, тем самым увеличивая объем

передаваемых данных. Однако в случае использования WebSocket протокола, после установления двунаправленного соединения между сервером и клиентом, никаких переподписок не происходит, а также не происходит отправка тяжеловесных HTTP заголовков вместе с исходным сообщением, тем самым уменьшая объем передаваемых данных.

6 Заключение

В ходе данной работы 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). Кроме этого по результатам нагрузочного тестирования удалось выяснить, что передача данных с использованием WebSocket протокола является наиболее оптимальной для двунаправленной передачи данных.

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.