

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	3
1 Краткое описание архитектуры NodeJS	4
2 Краткое описание архитектуры NGINX	5
2.1 Несколько слов о важности архитектуры	5
2.2 Основные моменты работы NGINX.....	6
2.3 Внутри рабочего процесса	6
2.4 Устройство конечного автомата	6
2.5 Блокирующийся конечный автомат	7
3 Способы асинхронного обмена данными с сервером	8
3.1 WebSocket.....	8
3.1.1 Установление WebSocket соединения	8
3.1.2 Оформление и передача данных с использованием WebSocket протокола	9
3.1.3 Закрытие WebSocket соединения	10
3.2 Comet	10
3.3 IFrame	11
3.3.1 Общая схема работы	11
3.4 JSONP	11
3.4.1 Общая схема работы	12
4 Описание архитектуры приложения	14
4.1 Цикл работы приложения	20
5 Нагрузочное тестирование Comet и WebSocket соединений	22
6 Заключение	24
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	25
Приложение А Листинг Comet сервера, написанного на JavaScript.....	26
Приложение Б Листинг WebSocket сервера, написанного на JavaScript....	31
Приложение В Листинг Static сервера, написанного на JavaScript.....	35
Приложение Г Листинг Messaging сервиса, написанного на JavaScript	36
Приложение Д Листинг с кодом для Gulp task runner, написанного на JavaScript	39
Приложение Е Конфигурационный файл NGINX	43

ВВЕДЕНИЕ

Javascript является высокоуровневым прототипно-ориентированным языком программирования. Изначально созданный в 1995 году как язык для работы в среде браузера, уже в 1996 году в компании Netscape начали проводиться попытки по адаптиванию данного языка под нужды серверной стороны, однако данная технология не получила широкого распространения.

В 2009 году после нескольких лет экспериментирования над созданием серверных веб-компонентов на Javascript Райан Дал разработал NodeJS. NodeJS представляет собой платформу, основанную на JavaScript движке V8 (разработанной компанией Google), транслирующем JavaScript в машинный код, таким образом преобразующим 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 tessell). 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 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

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]

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 *keep-alive*-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: hsBlbuDTkk24srzEOTBUlZA1C2g=
```

Здесь заголовок Sec-WebSocket-Accept представляет собой перекодированный с использованием специального алгоритма ключ Sec-WebSocket-Key. Браузер в свою очередь использует ее для проверки, что ответ предназначен 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»).

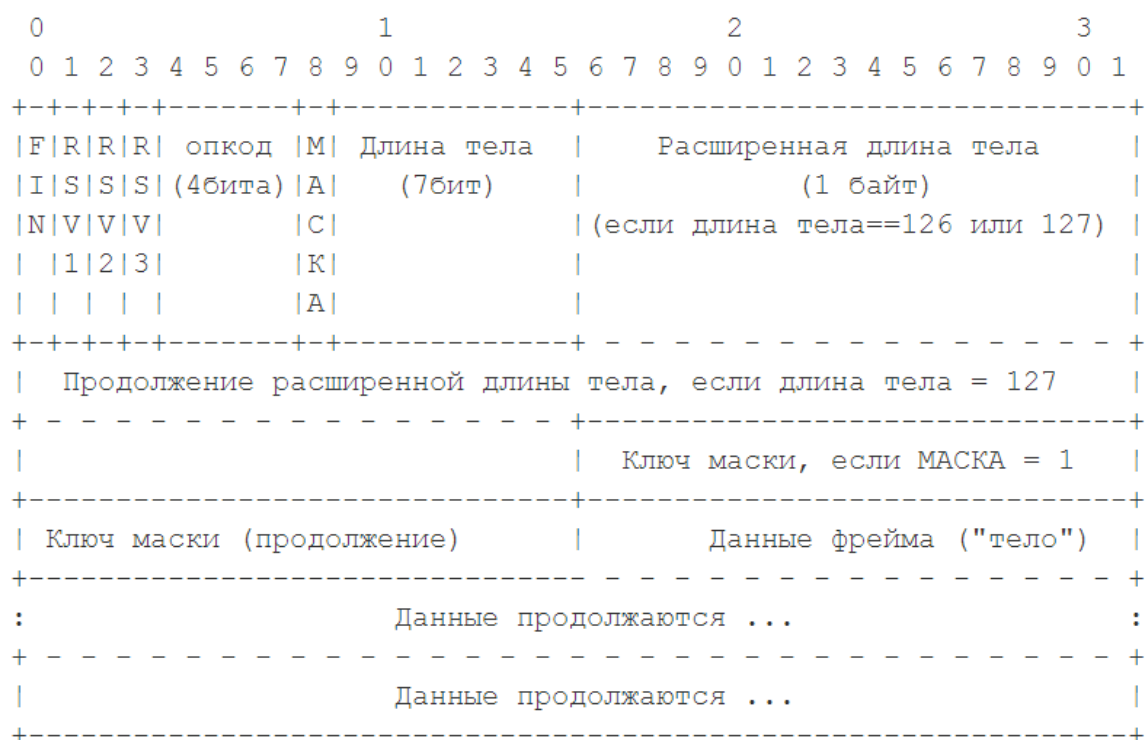


Рисунок 1 – Схема WebSocket фрейма

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 отправляет 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

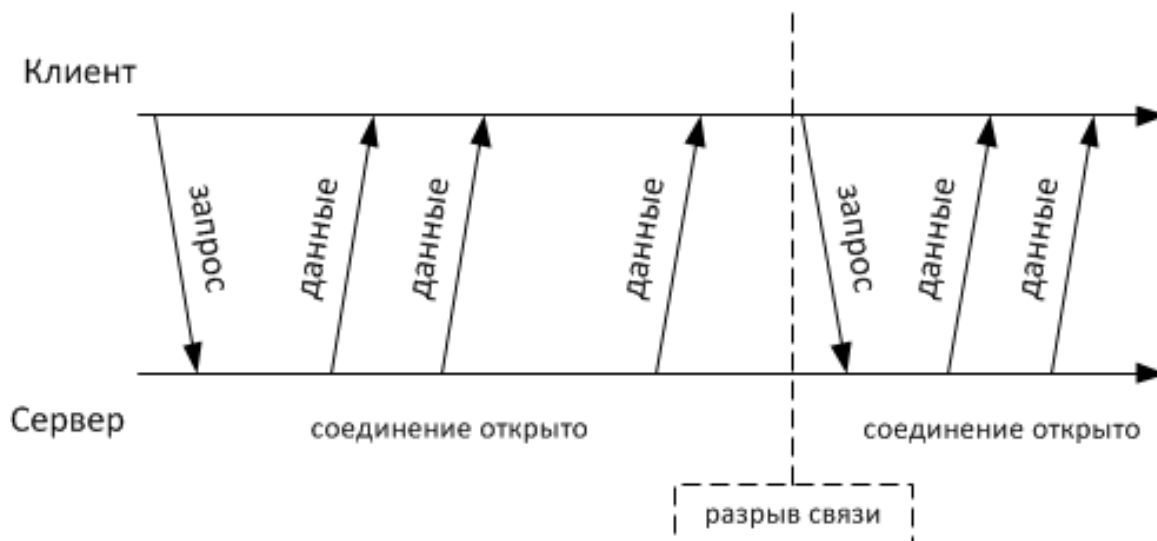


Рисунок 2 – Общая схема работы IFrame транспорта

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]

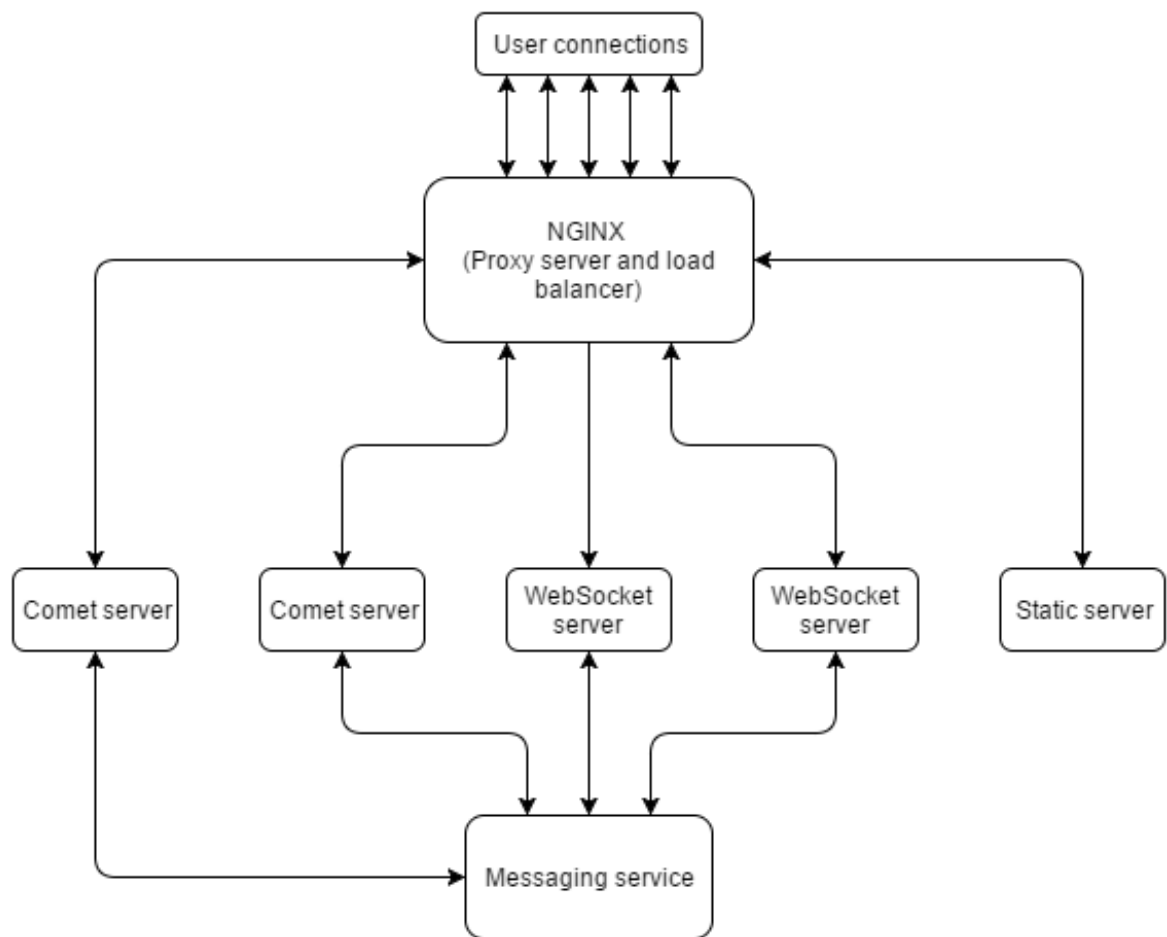


Рисунок 3 – Архитектура приложения

4 Описание архитектуры приложения

Архитектура приложения состоит из следующих компонентов:

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

На рисунке 3 можно увидеть общую схему приложения.

Итак, рассмотрим последовательно работу каждого компонента системы:

1. NGINX

Данный компонент используется в качестве прокси-сервера и балансировщика нагрузки. Важной частью NGINX является конфигурационный файл `nginx.conf`, позволяющий настроить поведение данного компонента. Инструкции

```
upstream websocket {
    least_conn;
    server localhost:8081;
    server localhost:8082;
}
```

```
upstream comet {
    ip_hash;
    server 127.0.0.1:8091;
    server 127.0.0.1:8092;
}
```

описывают две группы серверов: группа из двух websocket серверов и двух comet серверов соответственно. Для websocket группы используется директива балансировки нагрузки `least_conn`, которая передает новое соединение серверу с наименьшим числом активных подключений. В случае comet группы используется директива балансировки `ip_hash`, которая основывается на значении вычисленного хеша `ip` адреса пользователя, чтобы гарантировать, что все запросы от одного и того же клиента всегда перенаправлялись на один и тот же сервер. Для comet серверов это очень важно, поскольку в случае переподписки пользователя на разные сервера будет происходить постоянное удаление подписчиков на одном сервере и добавление их на другой, что в свою очередь будет пагубно сказываться на производительности приложения в целом.

Содержимое конфигурационного файла представлено в приложении Е.

2. WebSocket сервер

На этапе инициализации вызывается функция `init()`, в которой websocket сервер получает номер порта, на котором он будет развернут. После этого вызывается функция `initServer(port)`, в которой создается экземпляр websocket сервера, посредством следующей команды

```
var websocketServer = new WebSocketServer.Server({
    port: port
});
```

после которой он начинает прослушивать установленный порт. Далее происходит подписка на событие подключения сервера.

```

websocketServer.on('connection', function (ws) {
  //здесь код функции обратного вызова, обрабатывающий
  //каждое новое подключение
})

```

В коде выше указанной функции обратного вызова, в момент нового подключения генерируется уникальный идентификатор данного подключения, экземпляр подключения записывается в объект `clients`, а также происходит подписка на событие получения нового сообщения, а также на закрытие этого подключения.

```

//краткое содержание вышеописанной функции обратного вызова
//генерируем уникальный идентификатор
var id = _.uniqueId('ws_');
clients[id] = ws; // сохраняем экземпляр нового подключения
//в хранилище объекта clients

```

```

//подписка на событие получения нового сообщения
ws.on('message', function (message) {
  //в случае получения сообщения с неуказанным идентификатором
  //происходит отправка идентификатора клиенту
  //в противном случае происходит переотправка сообщения
  //всем подписчикам данного сервера,
  //а также отправка его в messaging сервис
  // с помощью метода sendOutWS(message);
})

```

```

//подписка на событие закрытия соединения
ws.on('close', function () {
  //удаляем подключение из хранилища подписчиков
  delete clients[id];
  //а также идет отправка сообщения для всех подписчиков
  //об уходе данного клиента
  sendOutWS({
    type: 'removeClient',
    removeClientId: wsMapIds[id]
  })

```



```
});  
}
```

Программный код websocket сервера представлен в приложении Б.

3. Comet сервер

Аналогично websocket серверу, на этапе инициализации также вызывается функция `init()`, в которой comet сервер получает номер порта, на котором он будет развернут.

```
\\парсим номер порта из строки в число  
serverPort = port = parseInt(port, 10);  
\\инициализируем http сервер  
\\прослушивающий заданный порт  
\\и вызывающий функцию обратного вызова ассерт  
\\в случае получения каких-либо запросов от клиентов  
http.createServer(accept).listen(port);
```

Далее, если пользователь обратился по пути `/subscribe`, то вызывается функция `onSubscribe(req, res, query)`; в которой происходит сохранение новых подключений в объекте-хранилище `subscribers`, а также подписка на событие закрытия соединения `close`.

```
req.on('close', function () {  
    //удаляем подключение из объекта-хранилища  
    delete subscribers[id];  
  
    //сообщаем всем подписчикам,  
    //что пользователь отсоединился  
    sendOutCOMET({  
        type: 'removeClient',  
        removeClientId: xhrMapIds[id]  
    });  
});
```

В случае обращения пользователя по пути `/publish` происходит отправка сообщения всем подписчикам с помощью метода `publish`, который в свою очередь вызывает метод `sendOutCOMET(message)`, который уже

делает переотправку сообщения подписчикам текущего сервера, а также передает сообщение messaging сервису. Программный код comet сервера представлен в приложении А.

4. Static сервер

Данный сервер используется для возврата UI (от user interface - пользовательский интерфейс) функционала (html + css + javascript), то есть все то, что необходимо клиенту для дальнейшей работы с приложением.

```
var express = require('express'), //подключаем модуль express
    app = express(), //получаем объект static сервера
    __dirname = './public'; //относительный путь к директории

//устанавливаем путь к директории,
//содержимое которой необходимо возвращать клиенту
app.use(express.static(__dirname));
app.listen(8088); //задаем порт, который нужно прослушивать
```

Программный код static сервера представлен в приложении В.

5. Messaging сервис

Данный сервис специализируется на выдаче портов для comet и websocket серверов, а также пересылает сообщения между экземплярами серверов. Для переотправки сообщений используется собственный websocket сервер, на который на старте инициализации подписываются основные comet и websocket сервера.

```
websocketServer = new ws.Server({
    port: 4010
}); //создается новый экземпляр websocket сервера

//происходит подписка на событие подключения
//экземпляров основных comet и websocket серверов.
websocketServer.on('connection', function (ws) {
    var id = urlToProcessId(ws.upgradeReq.url);

    //сохраняем подключение
    subscribers[id] = ws;
```

```

//подписываемся на получение нового сообщения
ws.on('message', function (message) {
    var key;

    //переотправляем сообщение для
    //остальных серверов
    for (key in subscribers) {
        if (subscribers.hasOwnProperty(key)
            && key !== id) {
            subscribers[key].send(message);
        }
    }
});

ws.on('close', function () {
    //удаляем соединение из объекта-хранилища
    //в случае его закрытия
    delete subscribers[id];
});
});

```

Программный код messaging сервиса представлен в приложении В.

6. Gulp task runner

Предназначен для запуска приложения и нагрузочных тестов. В случае запуска приложения, необходимо открыть консоль в директории с gulpfile.js и прописать команду gulp serve. Эта команда вызывает данный фрагмент кода.

```

gulp.task('serve', function () {
    var serverHelper;

    stopNginx = true; //флаг для остановки NGINX после
    //закрытия консоли

    //запускаем .bat файл с инструкциями для перезапуска
    //NGINX сервера

```

```

exec('start restart-nginx.bat', callback);

//создаем дочерний процесс,
//в котором будет работать messaging сервис
serverHelper = fork('server-helper-util.js',
{maxBuffer: maxBufferSize}, callback);

serverHelper.on('message', function (msg) {
//когда messaging сервис инициализировался
    if (msg === 'done') {
        //запускаем в дочерних процессах
        //2 websocket сервера
        //2 comet сервера
        //и 1 static сервер
        fork('static-server.js',
        {maxBuffer: maxBufferSize}, callback);

        fork('ws-server.js',
        {maxBuffer: maxBufferSize}, callback);
        fork('ws-server.js',
        {maxBuffer: maxBufferSize}, callback);

        fork('comet-server.js',
        {maxBuffer: maxBufferSize}, callback);
        fork('comet-server.js',
        {maxBuffer: maxBufferSize}, callback);
    }
});
});

```

Программный код gulp task runner представлен в приложении Д.

4.1 Цикл работы приложения

Итак, подведем краткие итоги вышесказанному.

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.

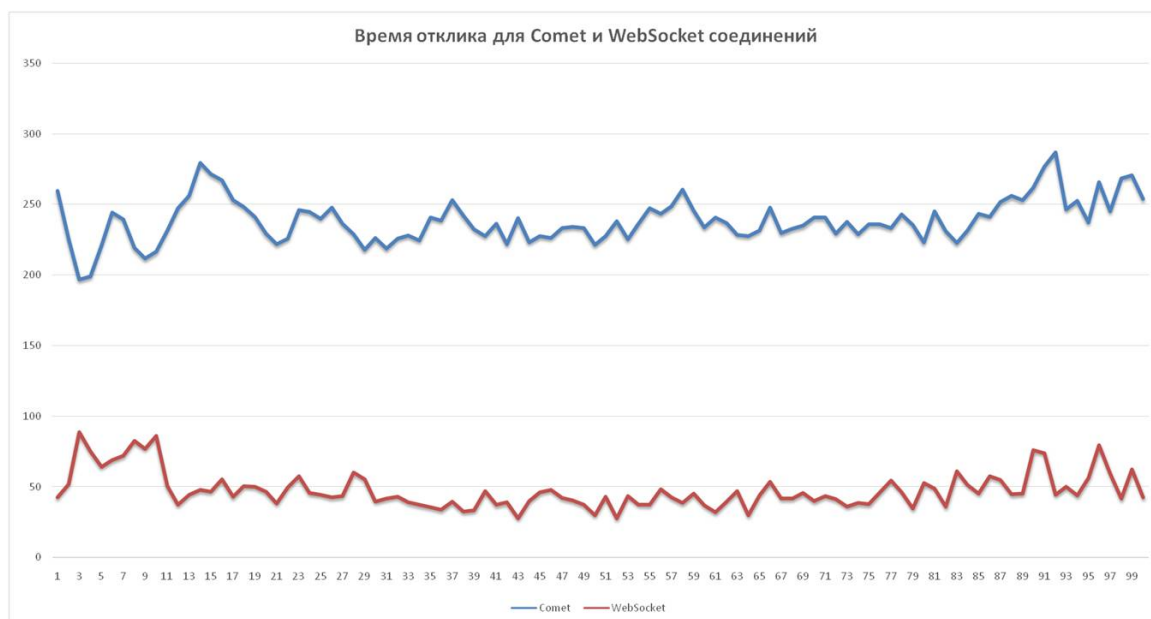


Рисунок 4 – Статистика по Comet и WebSocket соединениям

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. Данный результат объясняется тем, что для Comet соединения клиентская сторона после каждого полученного сообщения вынуждена 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 тяжелых 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.

ПРИЛОЖЕНИЕ А

Листинг Comet сервера, написанного на JavaScript

Код приложения comet-server.js.

```
1 var http = require('http'),
2     request = require('request'),
3     log4js = require('log4js'),
4     log4jsConfig = require('./config/config.js').log4js,
5     loggererror,
6     loggerlog,
7     WebSocketClient = require('websocket').client,
8     wsClient = new WebSocketClient(),
9     emitToCluster,
10    processId = process.pid,
11    url = require('url'),
12    _ = require('lodash'),
13    serverPort,
14    subscribers = {},
15    xhrMapIds = {};
16
17 init();
18
19 function init() {
20     log4js.configure(log4jsConfig);
21     loggererror = log4js.getLogger('error');
22     loggerlog = log4js.getLogger('info');
23
24     request({
25         uri: 'http://localhost:4000/getPort:comet',
26         method: 'GET'
27     }, function (error, message, port) {
28         serverPort = port = parseInt(port, 10);
29
30         http.createServer(accept).listen(port);
31         initServerInstancesCommunications();
32         initLogInterval(10000);
33         log('XHR Сервер запущен на порту ' + port);
34     });
35 }
36
37 function initLogInterval(time) {
```

```

38     time = parseInt(time, 10) || 10000;
39
40     log('Log interval has been started!');
41     setInterval(function () {
42         var messages = ['comet-server.js on port ',
43             serverPort,
44             ' Connections ',
45             Object.keys(subscribers).length,
46             ' Memory usage: ',
47             JSON.stringify(process.memoryUsage())].join('');
48
49         loggerlog.info(messages);
50         log('Logged info message: ' + messages);
51     }, time);
52 }
53
54
55 function onSubscribe(req, res, query) {
56     var id,
57         mapId;
58
59     res.setHeader('Content-Type', 'text/plain; charset=utf-8');
60     res.setHeader("Cache-Control", "no-cache, must-revalidate");
61
62     id = _.uniqueId('xhr_');
63
64     if (typeof query.clientId !== 'undefined') {
65         id = query.clientId;
66         subscribers[id] = res;
67         xhrMapIds[id] = query.clientId;
68     } else {
69         mapId = _.uniqueId('');
70         xhrMapIds[id] = mapId;
71         subscribers[id] = res;
72         subscribers[id].end(JSON.stringify({
73             type: 'newClient',
74             clientId: mapId
75         }));
76     }
77
78     log("новый XHR клиент " + id + ", клиентов: " +

```

```

79         Object.keys(subscribers).length);
80
81     req.on('close', function () {
82         delete subscribers[id];
83         log("XHR клиент " + id + " отсоединился, клиентов:" +
84             Object.keys(subscribers).length);
85
86         sendOutCOMET({
87             type: 'removeClient',
88             removeClientId: xhrMapIds[id]
89         });
90     });
91
92 }
93
94 function publish(message) {
95     log(JSON.stringify(message));
96     log("есть сообщение, клиентов:" + Object.keys(subscribers).length);
97
98     sendOutCOMET(message);
99 }
100
101 function accept(req, res) {
102     var urlParsed = url.parse(req.url, true),
103         query = urlParsed.query;
104
105     // новый клиент хочет получать сообщения
106     if (urlParsed.pathname == '/subscribe') {
107         onSubscribe(req, res, query); // собственно, подписка
108         return;
109     }
110
111     log('XHR: ' + req.data);
112     log('XHR urlParsed.pathname ' + urlParsed.pathname);
113
114     // отправка сообщения
115     if (urlParsed.pathname == '/publish' && req.method == 'POST') {
116         // принять POST-запрос
117         req.setEncoding('utf8');
118
119         req.on('data', function (message) {

```

```

120         log("XHR!");
121         publish(JSON.parse(message)); // собственно, отправка
122         res.end("ok");
123     });
124 }
125 }
126
127 function sendOutCOMET(message) {
128     var id,
129         res;
130     emitToCluster(message);
131     message = JSON.stringify(message);
132     for (id in subscribers) {
133         if (subscribers.hasOwnProperty(id)) {
134             res = subscribers[id];
135             res.end(message);
136         }
137     }
138 }
139
140 function sendOutCOMET_withoutCluster(message) {
141     var id,
142         res;
143     message = JSON.stringify(message);
144     for (id in subscribers) {
145         if (subscribers.hasOwnProperty(id)) {
146             res = subscribers[id];
147             res.end(message);
148         }
149     }
150 }
151
152
153 function initServerInstancesCommunications() {
154     wsClient.on('connectFailed', function (error) {
155         log('Connect Error: ' + error.toString());
156     });
157
158     wsClient.on('connect', function (connection) {
159         log('WebSocket Client Sender Connected');
160

```

```

161     emitToCluster = function (data) {
162         if (connection.connected) {
163             connection.send(JSON.stringify(data));
164         }
165     };
166
167     connection.on('message', function (message) {
168         log('(' + processId + ') Got message ');
169         console.dir(message);
170         sendOutCOMET_withoutCluster(message);
171     });
172 });
173
174 wsClient.connect('ws://localhost:4010/?processId=' + processId,
175                 'echo-protocol');
176 }
177
178 function log(msg) {
179     console.log('\n\ncomet-server.js on port ' + serverPort + ' : ' + msg);
180 }
181
182 process.on('uncaughtException', function (err) {
183     var messages = ['comet-server.js on port ' + serverPort,
184                   'uncaughtException: ',
185                   err.message, '\r\n',
186                   err.stack].join('');
187
188     loggererror.info(messages);
189
190     console.error('uncaughtException: ', err.message);
191     console.error(err.stack);
192 });

```

ПРИЛОЖЕНИЕ Б

Листинг WebSocket сервера, написанного на JavaScript

Код приложения ws-server.js.

```
1 var WebSocketServer = new require('ws'),
2   WebSocketClient = require('websocket').client,
3   log4js = require('log4js'),
4   log4jsConfig = require('./config/config.js').log4js,
5   loggererror,
6   loggerlog,
7   wsClient = new WebSocketClient(),
8   processId = process.pid,
9   request = require('request'),
10  emitToCluster,
11  _ = require('lodash'),
12  serverPort,
13  clients = {},
14  wsMapIds = {};
15
16 init();
17
18 function init() {
19   log4js.configure(log4jsConfig);
20   loggererror = log4js.getLogger('error');
21   loggerlog = log4js.getLogger('info');
22
23   request({
24     uri: 'http://localhost:4000/getPort:ws',
25     method: 'GET'
26   }, function (error, message, port) {
27     serverPort = port = parseInt(port, 10);
28
29     initServer(port);
30
31     initLogInterval(10000);
32     initServerInstancesCommunications();
33   });
34 }
35
36 function initLogInterval(time) {
37   time = parseInt(time, 10) || 10000;
```

```

38
39 log('Log interval has been started!');
40 setInterval(function () {
41     var messages = ['ws-server.js on port ',
42         serverPort,
43         ' Connections ',
44         Object.keys(clients).length,
45         ' Memory usage: ',
46         JSON.stringify(process.memoryUsage())].join('');
47
48     loggerlog.info(messages);
49     log('Logged info message: ' + messages);
50 }, time);
51 }
52
53 function initServer(port) {
54     var websocketServer = new WebSocketServer.Server({
55         port: port
56     });
57
58     websocketServer.on('connection', function (ws) {
59         var id = _.uniqueId('ws_');
60         clients[id] = ws;
61         log("новое WS соединение " + id);
62
63         ws.on('message', function (message) {
64             var mapId;
65             log('получено WS сообщение ' + message);
66             message = JSON.parse(message);
67             if (message.type === 'ping' && typeof message.id === 'undefined') {
68                 mapId = _.uniqueId('');
69                 wsMapIds[id] = mapId;
70                 clients[id].send(JSON.stringify({
71                     type: 'newClient',
72                     clientId: mapId
73                 }));
74             } else {
75                 wsMapIds[id] = message.id;
76                 log("WS!");
77                 sendOutWS(message);
78             }
79         });
80     });
81 }

```



```

79
80     });
81
82     ws.on('close', function () {
83         log('соединение WS закрыто ' + id);
84         delete clients[id];
85         sendOutWS({
86             type: 'removeClient',
87             removeClientId: wsMapIds[id]
88         });
89     });
90
91 });
92
93 }
94
95
96 function sendOutWS(message) {
97     var key;
98     emitToCluster(message);
99     message = JSON.stringify(message);
100    for (key in clients) {
101        if (clients.hasOwnProperty(key)) {
102            clients[key].send(message);
103        }
104    }
105 }
106
107
108 function sendOutWS_withoutCluster(message) {
109     var key;
110     message = JSON.stringify(message);
111     for (key in clients) {
112         if (clients.hasOwnProperty(key)) {
113             clients[key].send(message);
114         }
115     }
116 }
117
118 function initServerInstancesCommunications() {
119     wsClient.on('connectFailed', function (error) {

```

```

120     log('Connect Error: ' + error.toString());
121 });
122
123 wsClient.on('connect', function (connection) {
124     log('WebSocket Client Sender Connected');
125
126     emitToCluster = function (data) {
127         if (connection.connected) {
128             connection.send(JSON.stringify(data));
129         }
130     };
131
132     connection.on('message', function (message) {
133         log('(' + processId + ') Got message ');
134         console.dir(message);
135         sendOutWS_withoutCluster(message);
136     });
137 });
138
139 wsClient.connect('ws://localhost:4010/?processId=' + processId,
140                 'echo-protocol');
141 }
142
143 function log(msg) {
144     console.log('\n\nws-server.js on port ' + serverPort + ' : ' + msg);
145 }
146
147 process.on('uncaughtException', function (err) {
148     var messages = ['comet-server.js on port ' + serverPort,
149                   'uncaughtException: ',
150                   err.message, '\r\n',
151                   err.stack].join('');
152
153     loggererror.info(messages);
154
155     console.error('uncaughtException: ', err.message);
156     console.error(err.stack);
157 });

```

ПРИЛОЖЕНИЕ В

Листинг Static сервера, написанного на JavaScript

Код приложения static-server.js.

```
1 var express = require('express'),
2     app = express(),
3     __dirname = './public';
4
5 app.use(express.static(__dirname));
6 console.info('\n\nStatic server started! Port 8088!');
7 app.listen(8088);
```

ПРИЛОЖЕНИЕ Г

Листинг Messaging сервиса, написанного на JavaScript

Код приложения server-helper-util.js.

```
1 var express = require('express'),
2   bodyParser = require('body-parser'),
3   ws = new require('ws'),
4   url = require('url'),
5   websocketServer,
6   subscribers = {},
7   app = express(),
8   maxId = 0,
9   cometNum = 0,
10  wsNum = 0;
11
12 app.use(bodyParser.json());
13
14 app.get('/getId', function (req, res) {
15   res.send('' + maxId);
16   maxId++;
17 });
18
19 app.get('/getPort:type', function (req, res) {
20
21   switch (req.params.type) {
22     case ':comet':
23       cometNum++;
24       log('Taken comet port 809' + oneOrTwo(cometNum));
25       res.send('809' + cometNum);
26       break;
27     case ':ws':
28       wsNum++;
29       log('Taken ws port 808' + oneOrTwo(wsNum));
30       res.send('808' + wsNum);
31       break;
32     default:
33       break;
34   }
35 });
36
37 app.listen(4000, function () {
```

```

38     log('Server helper util listening on port 4000!');
39     process.send('done');
40 });
41
42
43 //===== WebSocket helper for client servers =====
44
45 websocketServer = new ws.Server({
46     port: 4010
47 });
48
49 websocketServer.on('connection', function (ws) {
50     var id = urlToProcessId(ws.upgradeReq.url); //saving id in js closure
51
52     subscribers[id] = ws;
53
54     ws.on('message', function (message) {
55         var key;
56
57         log('New message for cluster');
58         console.dir(Object.keys(subscribers));
59         console.dir(JSON.parse(message));
60         for (key in subscribers) { //here id is available
61             if (subscribers.hasOwnProperty(key) && key !== id) {
62                 subscribers[key].send(message);
63             }
64         }
65     });
66
67     ws.on('close', function () {
68         delete subscribers[id];
69     });
70 });
71
72 function urlToProcessId(url) {
73     return url.substring(url.indexOf('processId=') + 10);
74 }
75
76 function oneOrTwo(number) {
77     return number % 2 + 1;
78 }

```

```
79
80 function log(msg) {
81     console.log('\n\nserver-helper-util.js : ' + msg);
82 }
83
84 function dir(msg) {
85     console.log('\n\nserver-helper-util.js : ');
86     console.dir(msg);
87 }
```

ПРИЛОЖЕНИЕ Д

Листинг с кодом для Gulp task runner, написанного на JavaScript

Код приложения gulpfile.js.

```
1 var gulp = require('gulp'),
2     exec = require('child_process').exec,
3     fork = require('child_process').fork,
4     tcpPortUsed = require('tcp-port-used'),
5     maxBufferSize = 1024 * 50000,
6     stopNginx = false;
7
8 //require('require-dir')('./gulp');
9
10 gulp.task('serve', function () {
11     var serverHelper;
12     stopNginx = true;
13
14     exec('start restart-nginx.bat', callback);
15     serverHelper = fork('server-helper-util.js',
16                        {maxBuffer: maxBufferSize}, callback);
17
18     serverHelper.on('message', function (msg) {
19         if (msg === 'done') {
20             fork('static-server.js', {maxBuffer: maxBufferSize}, callback);
21             fork('ws-server.js', {maxBuffer: maxBufferSize}, callback);
22             fork('ws-server.js', {maxBuffer: maxBufferSize}, callback);
23             fork('comet-server.js', {maxBuffer: maxBufferSize}, callback);
24             fork('comet-server.js', {maxBuffer: maxBufferSize}, callback);
25         }
26     });
27
28 });
29
30 gulp.task('test-comet', function () {
31     var numberOfSubscribers = parseInt(process.argv.slice(4)[0], 10) || 1,
32         initCometSubscribers = initSubscribers.bind({}, numberOfSubscribers,
33                                                    'test-comet.js',
34                                                    'test-comet-send.js'),
35         testHelper;
36
37     console.log('Start of ' + numberOfSubscribers + ' test instances...');
```

```

38
39     check3000Port(function (inUse) {
40         if (inUse) {
41             initCometSubscribers();
42         } else {
43             testHelper = fork('test-helper-util.js');
44
45             testHelper.on('message', function (msg) {
46                 if (msg === 'done') {
47                     initCometSubscribers();
48                 }
49             });
50         }
51     });
52
53
54 });
55
56 gulp.task('test-ws', function () {
57     var numberOfSubscribers = parseInt(process.argv.slice(4)[0], 10) || 1,
58         testHelper,
59         initWsSubscribers = initSubscribers.bind({}, numberOfSubscribers,
60                                                     'test-ws.js',
61                                                     'test-ws-send.js'),
62         i;
63
64     console.log('Start of ' + numberOfSubscribers + ' test instances...');
65
66     check3000Port(function (inUse) {
67         if (inUse) {
68             initWsSubscribers();
69         } else {
70             testHelper = fork('test-helper-util.js');
71
72             testHelper.on('message', function (msg) {
73                 if (msg === 'done') {
74                     initWsSubscribers();
75                 }
76             });
77         }
78     });

```



```

79
80
81 });
82
83 function initSubscribers(numberOfSubscribers, listenerJS, publisherJS) {
84     var activeSubscribers = 0,
85         i;
86
87     for (i = 0; i < numberOfSubscribers; i++) {
88         console.log('Starting ' + i + ' ' + listenerJS);
89
90         fork(listenerJS).on('message', function (message) {
91             if (message === 'done') {
92                 activeSubscribers++;
93                 console.log('Active subscribers: ' + activeSubscribers);
94
95                 if (activeSubscribers === numberOfSubscribers) {
96                     console.log('Starting ' + publisherJS);
97
98                     if (publisherJS !== 'test-ws-send.js') {
99                         fork(publisherJS);
100                     }
101                 }
102             }
103         });
104     }
105 }
106
107 function check3000Port(callback) {
108     tcpPortUsed.check(3000, '127.0.0.1')
109         .then(callback, function (err) {
110             console.error('Error on check:', err.message);
111         });
112 }
113
114 function callback(error, stdout, stderr) {
115     if (error) {
116         console.log('Error!');
117         console.dir(error);
118     } else {
119         if (this && this.msg) {

```

```
120         console.log(this.msg);
121     }
122 }
123 }
124
125 process.on('SIGINT', function () {
126     console.log("\nGracefully shutting down from SIGINT (Ctrl-C)");
127     if (stopNginx) {
128         exec('start stop-nginx.bat', function () {
129             console.log('Done!');
130             process.exit(0);
131         });
132     } else {
133         process.exit(0);
134     }
135 });
```

ПРИЛОЖЕНИЕ Е

Конфигурационный файл NGINX

Код приложения nginx.conf.

```
1 worker_processes  auto;
2
3
4 events {
5     worker_connections  1024;
6 }
7
8
9 http {
10     include          mime.types;
11     default_type      application/octet-stream;
12     sendfile          on;
13     keepalive_timeout  65;
14     proxy_read_timeout 950s;
15
16     map $http_upgrade $connection_upgrade {
17         default upgrade;
18         '' close;
19     }
20
21     map $http_x_requested_with $nocache {
22         default 0;
23         XMLHttpRequest 1;
24     }
25
26     upstream websocket {
27         least_conn;
28         server localhost:8081;
29         server localhost:8082;
30     }
31
32     upstream comet {
33         ip_hash;
34         server 127.0.0.1:8091;
35         server 127.0.0.1:8092;
36     }
37
```

```

38  server {
39      listen      10000;
40      server_name  localhost;
41
42      location /xhr/publish/ {
43          proxy_pass http://comet/publish;
44          proxy_cache_bypass $nocache;
45          proxy_no_cache     $nocache;
46          proxy_buffering    off;
47          proxy_connect_timeout      6000;
48          proxy_send_timeout         6000;
49          proxy_read_timeout         6000;
50      }
51
52      location /xhr/subscribe/ {
53          proxy_pass http://comet/subscribe;
54          proxy_cache_bypass $nocache;
55          proxy_no_cache     $nocache;
56          proxy_buffering    off;
57          proxy_connect_timeout      6000;
58          proxy_send_timeout         6000;
59          proxy_read_timeout         6000;
60      }
61
62      location /ws/ {
63          proxy_pass http://websocket;
64          proxy_http_version 1.1;
65          proxy_set_header Upgrade $http_upgrade;
66          proxy_set_header Connection $connection_upgrade;
67          proxy_redirect off;
68      }
69
70      location / {
71          proxy_pass http://localhost:8088/;
72      }
73  }
74
75 }

```