РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

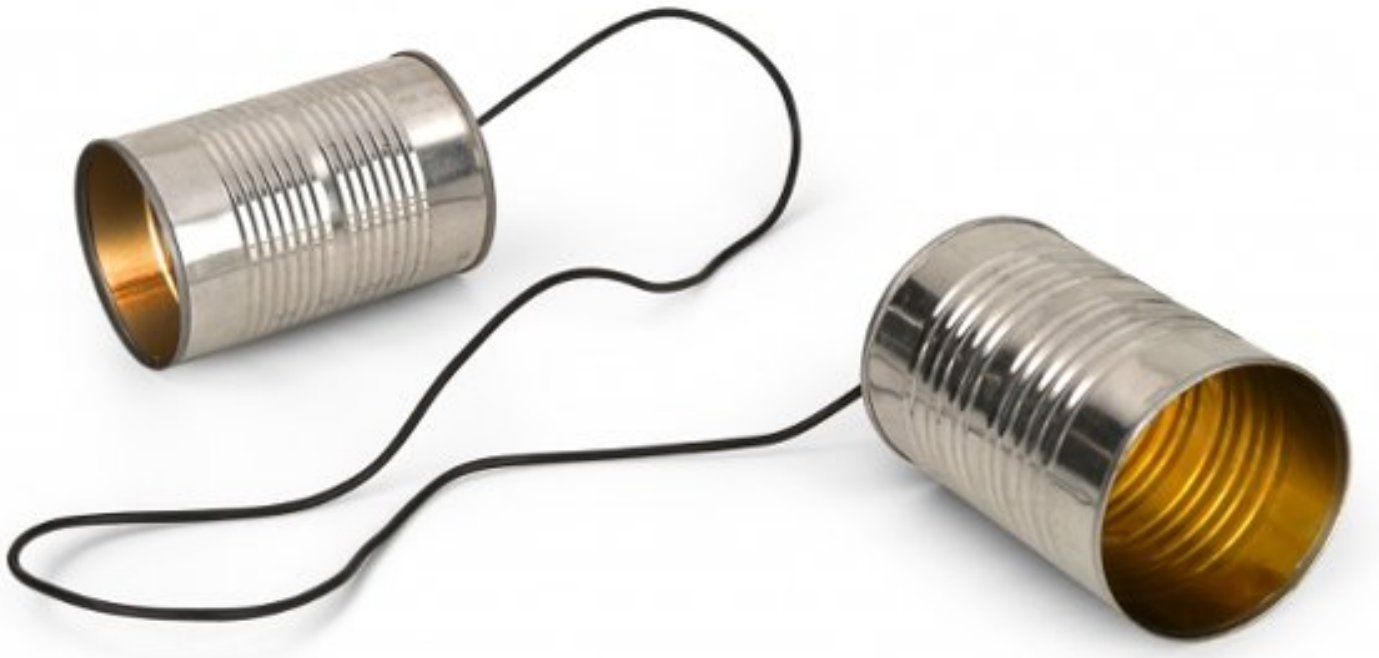
Как работает JS: WebSocket и HTTP/2+SSE. Что выбрать?

ПЕРЕВОД

ru_vds 14 ноября 2017 в 13:49 20k

Оригинал: Alexander Zlatkov

Перед вами — перевод пятого материала из серии, посвящённой особенностям JS-разработки. В предыдущих статьях мы рассматривали основные элементы экосистемы JavaScript, возможностями которых пользуются разработчики серверного и клиентского кода. В этих материалах, после изложения основ тех или иных аспектов JS, даются рекомендации по их использованию. Автор статьи говорит, что эти принципы применяются в ходе разработки приложения SessionStack. Современный пользователь библиотек и фреймворков может выбирать из множества возможностей, поэтому любому проекту, для того, чтобы достойно смотреться в конкурентной борьбе, приходится выжимать из технологий, на которых он построен, всё, что можно.



В этот раз мы поговорим о коммуникационных протоколах, сопоставим и обсудим их особенности и составные части. Тут мы займёмся технологиями WebSocket и HTTP/2, в частности, поговорим о безопасности и поделимся советами, касающимися выбора подходящих протоколов в различных ситуациях.

Введение

В наши дни сложные веб-приложения, обладающие насыщенными динамическими пользовательскими интерфейсами, воспринимаются как нечто само собой разумеющееся. А ведь интернету пришлось пройти долгий путь для того, чтобы достичь его сегодняшнего состояния.

В самом начале интернет не был рассчитан на поддержку подобных приложений. Он был задуман как коллекция HTML-страниц, как «паутина» из связанных друг с другом ссылками

документов. Всё было, в основном, построено вокруг парадигмы HTTP «запрос/ответ». Клиентские приложения загружали страницы и после этого ничего не происходило до того момента, пока пользователь не щёлкнул мышью по ссылке для перехода на очередную страницу.

Примерно в 2005-м году появилась технология AJAX и множество программистов начало исследовать возможности двунаправленной связи между клиентом и сервером. Однако, все сеансы HTTP-связи всё ещё инициировал клиент, что требовало либо участия пользователя, либо выполнения периодических обращений к серверу для загрузки новых данных.

«Двунаправленный» обмен данными по HTTP

Технологии, которые позволяют «упреждающе» отправлять данные с сервера на клиент существуют уже довольно давно. Среди них — Push и Comet.

Один из наиболее часто используемых приёмов для создании иллюзии того, что сервер самостоятельно отправляет данные клиенту, называется «длинный опрос» (long polling). С использованием этой технологии клиент открывает HTTP-соединение с сервером, который держит его открытым до тех пор, пока не будет отправлен ответ. В результате, когда у сервера появляются данные для клиента, он их ему отправляет.

Вот пример очень простого фрагмента кода, реализующего

технологию длинного опроса:

```
(function poll(){
   setTimeout(function(){
      $.ajax({
        url: 'https://api.example.com/endpoint',
        success: function(data) {
          // Делаем что-то с `data`
          // ...

          // Рекурсивно выполняем следующий запрос
          poll();
        },
        dataType: 'json'
      });
  }, 10000);
})();
```

Эта конструкция представляет собой функцию, которая сама себя вызывает после того, как, в первый раз, будет запущена автоматически. Она задаёт 10-секундный интервал для каждого асинхронного Ajax-обращению к серверу, а после обработки ответа сервера снова выполняется планирование вызова функции.

Ещё одна используемая в подобной ситуации техника — это Flash или составной запрос HXR, и так называемые htmlfiles.

У всех этих технологий одна и та же проблема: дополнительная нагрузка на систему, которую создаёт использование HTTP, что делает всё это неподходящим для организации работы приложений, где требуется высокая скорость отклика. Например, это что-то вроде многопользовательской браузерной «стрелялки» или любой другой онлайн-игры, действия в которой выполняются в

режиме реального времени.

Введение в технологию WebSocket

Спецификация WebSocket определяет API для установки соединения между веб-браузером и сервером, основанного на «сокете». Проще говоря, это — постоянное соединение между клиентом и сервером, пользуясь которыми клиент и сервер могут отправлять данные друг другу в любое время.

Клиент устанавливает соединение, выполняя процесс так называемого рукопожатия WebSocket. Этот процесс начинается с того, что клиент отправляет серверу обычный HTTP-запрос. В этот запрос включается заголовок `Upgrade`, который сообщает серверу о том, что клиент желает установить WebSocket-соединение.

Посмотрим, как установка такого соединения выглядит со стороны клиента:

```
// Создаём новое WebSocket-соединение.
var socket = new WebSocket('ws://websocket.example.com');
```

URL, применяемый для WebSocket-соединения, использует схему `ws`. Кроме того, имеется схема `wss` для организации защищённых WebSocket-соединений, что является эквивалентом HTTPS.

В данном случае показано начало процесса открытия WebSocket-соединения с сервером `websocket.example.com`.

Вот упрощённый пример заголовков исходного запроса.

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

Если сервер поддерживает протокол WebSocket, он согласится перейти на него и сообщит об этом в заголовке ответа `Upgrade`. Посмотрим на реализацию этого механизма с использованием Node.js:

```
// Будем использовать реализацию WebSocket из
//https://github.com/theturtle32/WebSocket-Node
var WebSocketServer = require('websocket').server;
var http = require('http');

var server = http.createServer(function(request, response) {
  // обработаем HTTP-запрос.
});
server.listen(1337, function() { });

// создадим сервер
wsServer = new WebSocketServer({
  httpServer: server
});
```

```
// WebSocket-сервер
wsServer.on('request', function(request) {
  var connection = request.accept(null, request.origin);

  // Это - самый важный для нас коллбэк, где обрабатываются
  // сообщения от клиента.
  connection.on('message', function(message) {
      // Обработаем сообщение WebSocket
  });

  connection.on('close', function(connection) {
    // Закрытие соединения
  });
});
```

После установки соединения в ответе сервера будут сведения о переходе на протокол WebSocket:

```
HTTP/1.1 101 Switching Protocols
Date: Wed, 25 Oct 2017 10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

После этого вызывается событие `open` в экземпляре WebSocket на клиенте:

```
var socket = new WebSocket('ws://websocket.example.com');

// Выводим сообщение при открытии WebSocket-соединения.
socket.onopen = function(event) {
  console.log('WebSocket is connected.');
};
```

Теперь, после завершения фазы рукопожатия, исходное HTTP-соединение заменяется на WebSocket-соединение, которое использует то же самое базовое TCP/IP-соединение. В этот

момент и клиент и сервер могут приступать к отправке данных.

Благодаря использованию WebSocket можно отправлять любые объёмы данных, не подвергая систему ненужной нагрузке, вызываемой использованием традиционных HTTP-запросов. Данные передаются по WebSocket-соединению в виде сообщений, каждое из которых состоит из одного или нескольких фреймов, содержащих отправляемые данные (полезную нагрузку). Для того, чтобы обеспечить правильную сборку исходного сообщения по достижению им клиента, каждый фрейм имеет префикс, содержащий 4-12 байтов данных о полезной нагрузке. Использование системы обмена сообщениями, основанной на фреймах, помогает сократить число служебных данных, передаваемых по каналу связи, что значительно уменьшает задержки при передаче информации.

Стоит отметить, что клиенту будет сообщено о поступлении нового сообщения только после того, как будут получены все фреймы и исходная полезная нагрузка сообщения будет реконструирована.

Различные URL протокола WebSocket

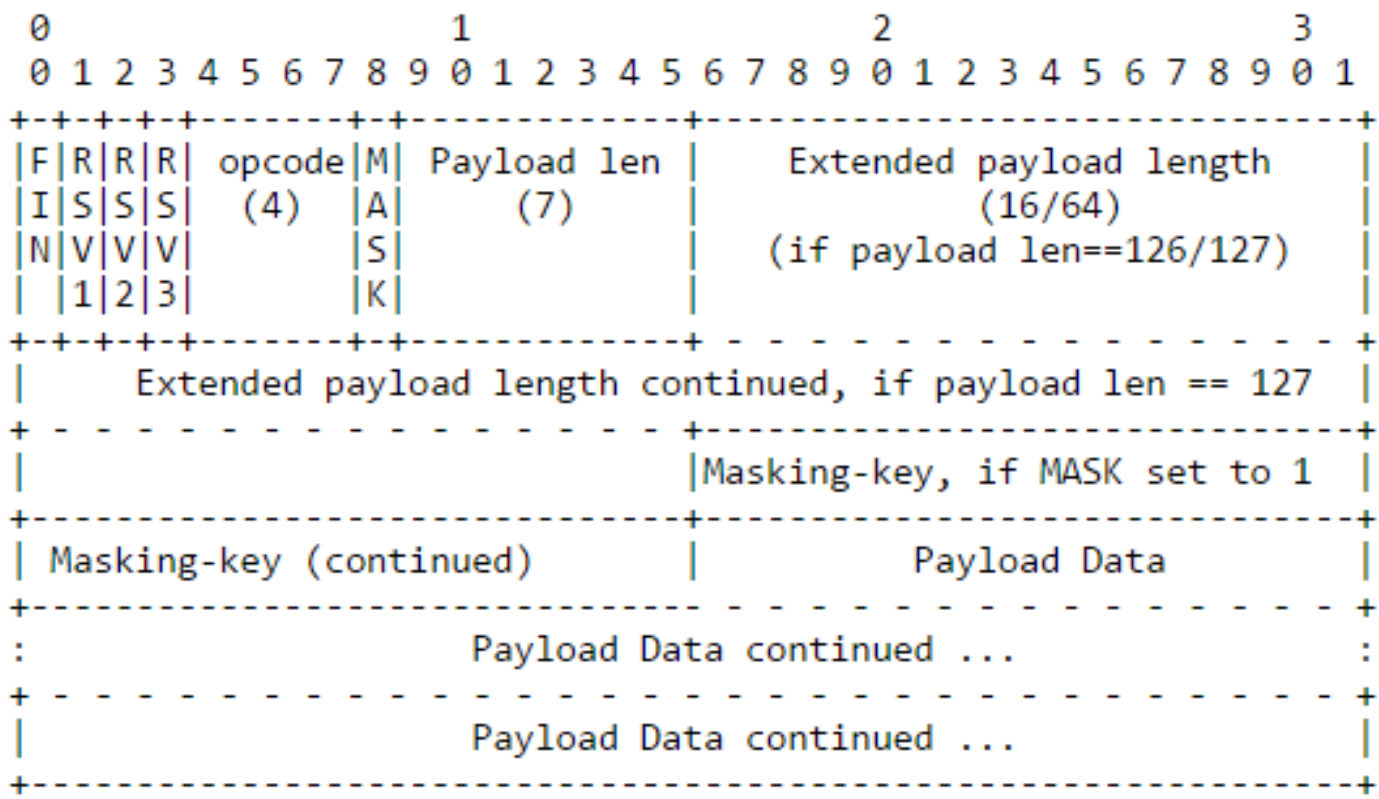
Выше мы упоминали о том, что в WebSocket используется новая схема URL. На самом деле — их две: `ws://` и `wss://`.

При построении URL-адресов используются определённые правила. Особенностью URL WebSocket является то, что они не поддерживают якоря (`#sample_anchor`).

В остальном к URL WebSocket применяются те же правила, что и к URL HTTP. При использовании ws-адресов соединение оказывается незашифрованным, по умолчанию применяется порт 80. При использовании wss требуется TLS-шифрование и применяется порт 443.

Протокол работы с фреймами

Взглянем поближе на протокол работы с фреймами WebSocket. Вот что можно узнать о структуре фрейма из соответствующего RFC:



Если говорить о версии WebSocket, стандартизированной RFC, то можно сказать, что в начале каждого пакета имеется небольшой заголовок. Однако, устроен он довольно сложно. Вот описание его

составных частей:

- `fin` (1 бит): указывает на то, является ли этот фрейм последним фреймом, завершающим передачу сообщения. Чаще всего для передачи сообщения достаточно одного фрейма и этот бит всегда оказывается установленным. Эксперименты показали, что Firefox создаёт второй фрейм после того, как размер сообщения превысит 32 Кб.
- `rsv1`, `rsv2`, `rsv3` (каждое по 1-му биту): эти поля должны быть установлены в 0, только если не было достигнуто договорённости о расширениях, которая и определит смысл их ненулевых значений. Если в одном из этих полей будет установлено ненулевое значение и при этом не было достигнуто договорённости о смысле этого значения, получатель должен признать соединение недействительным.
- `opcode` (4 бита): здесь закодированы сведения о содержимом фрейма. В настоящее время используются следующие значения:

  - `0x00`: в этом фрейме находится следующая часть передаваемого сообщения.
  - `0x01`: в этом фрейме находятся текстовые данные.
  - `0x02`: в этом фрейме находятся бинарные данные.
  - `0x08`: этот фрейм завершает соединение.
  - `0x09`: это ping-фрейм.
  - `0x0a`: это pong-фрейм.

Как видите, здесь достаточно неиспользуемых значений. Они

зарезервированы на будущее.

- `mask` (1 бит): указывает на то, что фрейм замаскирован. Сейчас дело обстоит так, что каждое сообщение от клиента к серверу должно быть замаскировано, в противном случае спецификации предписывают разрывать соединения.
- `payload_len` (7 битов): длина полезной нагрузки. Фреймы WebSocket поддерживают следующие методы указания размеров полезной нагрузки. Значение 0-125 указывает на длину полезной нагрузки. 126 означает, что следующие два байта означают размер. 127 означает, что следующие 8 байтов содержат сведения о размере. В результате длина полезной нагрузки может быть записана примерно в 7 битах, или в 16, или в 64-х.
- `masking-key` (32 бита): все фреймы, отправленные с клиента на сервер, замаскированы с помощью 32-битного значения, которое содержится во фрейме.
- `payload`: передаваемые во фрейме данные, которые, наверняка, замаскированы. Их длина соответствует тому, что задано в `payload_len`.

Почему протокол WebSocket основан на фреймах, а не на потоках? Если вы знаете ответ на этот вопрос — можете поделиться им в комментариях. Кроме того, вот интересное обсуждение на эту тему на HackerNews.

Данные во фреймах

Как уже было сказано, данные могут быть разбиты на множество

фреймов. В первом фрейме, с которого начинается передача данных, в поле `opcode`, задаётся тип передаваемых данных. Это необходимо, так как в JavaScript, можно сказать, не было поддержки бинарных данных во время начала работы над спецификацией WebSockets. Код `0x01` указывает на данные в кодировке UTF-8, код `0x02` используется для бинарных данных. Часто в пакетах WebSocket передают JSON-данные, для которых обычно устанавливают поле `opcode` как для текста. При передаче бинарных данных они будут представлены в виде Blob-сущностей, специфичных для веб-браузера.

API для передачи данных по протоколу WebSocket устроено очень просто:

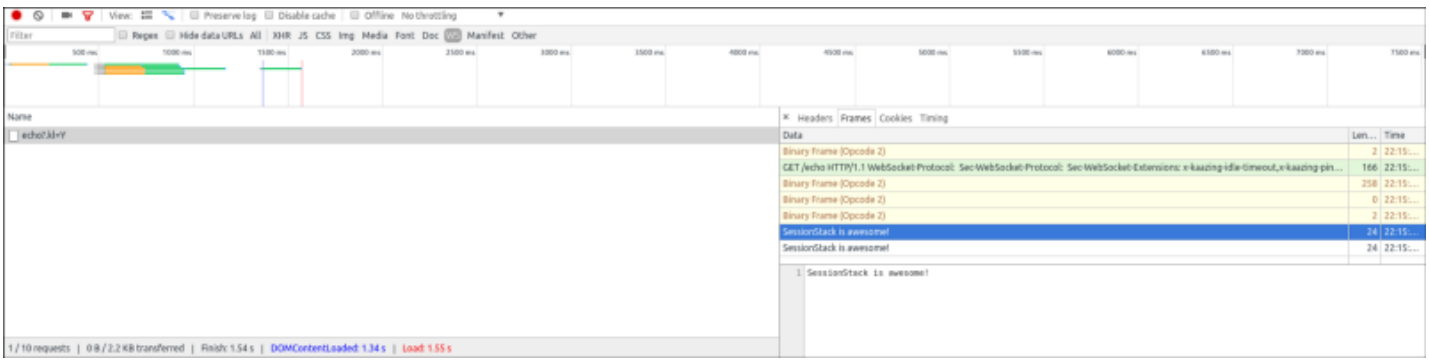
```
var socket = new WebSocket('ws://websocket.example.com');
socket.onopen = function(event) {
  socket.send('Some message'); // Отправка данных на сервер.
};
```

Когда, на клиентской стороне, WebSocket принимает данные, вызывается событие `message`. Это событие имеет свойство `data`, которое можно использовать для работы с содержимым сообщения.

```
// Обработка сообщений, отправленных сервером.
socket.onmessage = function(event) {
  var message = event.data;
  console.log(message);
};
```

Узнать, что находится внутри фреймов WebSocket-соединения,

можно с помощью вкладки Network (Сеть) инструментов разработчика Chrome:



Фрагментация данных

Полезные данные могут быть разбиты на несколько отдельных фреймов. Предполагается, что получающая сторона будет буферизовать фреймы до тех пор, пока не поступит фрейм с установленным полем заголовка `fin`. В результате, например, сообщение «Hello World» можно передать в 11 фреймах, каждый из которых несёт 1 байт полезной нагрузки и 6 байтов заголовочных данных. Фрагментация управляющих пакетов запрещена. Однако, спецификация даёт возможность обрабатывать чередующиеся управляющие фреймы. Это нужно в том случае, если TCP-пакеты прибывают в произвольном порядке.

Логика объединения фреймов, в общих чертах, выглядит так:

- Принять первый фрейм.
- Запомнить значение поля `opcode`.
- Принимать другие фреймы и объединять полезную нагрузку фреймов до тех пор, пока не будет получен фрейм с установленным битом `fin`.

- Проверить, чтобы поле `opcode` у всех фреймов, кроме первого, было установлено в ноль.

Основная цель фрагментации заключается в том, чтобы позволить отправку сообщений, размер которых неизвестен на момент начала отправки данных.

Благодаря фрагментации сервер может подобрать буфер разумного размера, а, когда буфер заполняется, отправлять данные в сеть. Второй вариант использования фрагментации заключается в мультиплексировании, когда нежелательно, чтобы сообщение занимало весь логический канал связи. В результате для целей мультиплексирования нужно иметь возможность разбивать сообщения на более мелкие фрагменты для того, чтобы лучше организовать совместное использование канала.

О heartbeat-сообщениях

В любой момент после процедуры рукопожатия, либо клиент, либо сервер, может решить отправить другой стороне ping-сообщение. Получая такое сообщение, получатель должен отправить, как можно скорее, pong-сообщение. Это и есть heartbeat-сообщения. Их можно использовать для того, чтобы проверить, подключён ли ещё клиент к серверу.

Сообщения «ping» и «pong» — это всего лишь управляющие фреймы. У ping-сообщений поле `opcode` установлено в значение `0x9`, у pong-сообщений — в `0xA`. При получении ping-сообщения, в ответ надо отправить pong-сообщение, содержащее ту же

полезную нагрузку, что и ping-сообщение (для таких сообщений максимальная длина полезной нагрузки составляет 125). Кроме того, вы можете получить pong-сообщение, не отправляя перед этим ping-сообщение. Такие сообщения можно просто игнорировать.

Подобная схема обмена сообщениями может быть очень полезной. Есть службы (вроде балансировщиков нагрузки), которые останавливают простаивающие соединения.

Вдобавок, одна из сторон не может, без дополнительных усилий, узнать о том, что другая сторона завершила работу. Только при следующей отправке данных вы можете выяснить, что что-то пошло не так.

Обработка ошибок

Обрабатывать ошибки в ходе работы с WebSocket-соединениями можно, подписавшись на событие `error`. Выглядит это примерно так:

```
var socket = new WebSocket('ws://websocket.example.com');

// Обработка ошибок.
socket.onerror = function(error) {
  console.log('WebSocket Error: ' + error);
};
```

Закрытие соединения

Для того, чтобы закрыть соединение, либо клиент, либо сервер,

должен отправить управляющий фрейм с полем `opcode`, установленным в `0x8`. При получении подобного фрейма другая сторона, в ответ, отправляет фрейм закрытия соединения. Первая сторона затем закрывает соединение. Таким образом, данные, полученные после закрытия соединения, отбрасываются.

Вот как инициируют операцию закрытия WebSocket-соединения на клиенте:

```
// Закрыть соединение, если оно открыто.
if (socket.readyState === WebSocket.OPEN) {
    socket.close();
}
```

Кроме того, для того, чтобы произвести очистку после завершения закрытия соединения, можно подписаться на событие `close`:

```
// Выполнить очистку.
socket.onclose = function(event) {
  console.log('Disconnected from WebSocket.');
};
```

Серверу нужно прослушивать событие `close` для того, чтобы, при необходимости, его обработать:

```
connection.on('close', function(reasonCode, description) {
    // Соединение закрывается.
});
```

Сравнение технологий WebSocket и HTTP/2

Хотя HTTP/2 предлагает множество возможностей, эта технология не может полностью заменить существующие push-технологии и потоковые способы передачи данных.

Первое, что важно знать об HTTP/2, заключается в том, что это — не замена всего, что есть в HTTP. Виды запросов, коды состояний и большинство заголовков остаются такими же, как и при использовании HTTP. Новшества HTTP/2 заключаются в повышении эффективности передачи данных по сети.

Если сравнить HTTP/2 и WebSocket, мы увидим много общих черт.

Показатель	HTTP/2	WebSocket
Сжатие заголовков	Да (HPACK)	Нет
Передача бинарных данных	Да	Да (бинарные или текстовые)
Мультиплексирование	Да	Да
Приоритизация	Да	Нет
Сжатие	Да	Да
Направление	Клиент/Сервер и Server Push	Двунаправленная передача данных
Полнодуплексный режим	Да	Да

Как уже было сказано, HTTP/2 вводит технологию Server Push, которая позволяет серверу отправлять данные в клиентский кэш по собственной инициативе. Однако, при использовании этой технологии данные нельзя отправлять прямо в приложение. Данные, отправленные сервером по своей инициативе, обрабатывает браузер, при этом нет API, которые позволяют, например, уведомить приложение о поступлении данных с сервера и отреагировать на это событие.

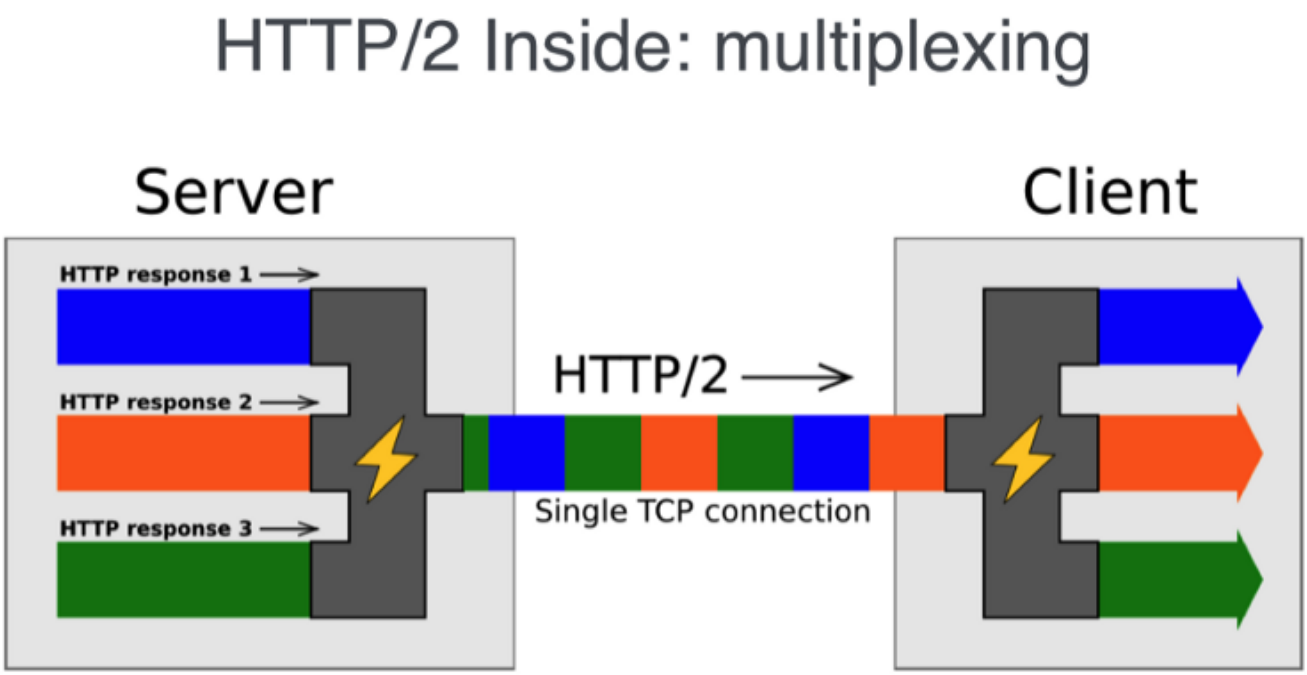
Именно в подобной ситуации весьма полезной оказывается технология Server-Sent Events (SSE). SSE — это механизм, который позволяет серверу асинхронно отправлять данные клиенту после установления клиент-серверного соединения.

После соединения сервер может отправлять данные по своему усмотрению, например, когда окажется готовым к передаче очередной фрагмент данных. Этот механизм можно представить себе как одностороннюю модель издатель-подписчик. Кроме того, в рамках этой технологии существует стандартное клиентское API для JavaScript, называемое `EventSource`, реализованное в большинстве современных браузеров как часть стандарта HTML5 W3C. Обратите внимание на то, что для браузеров, которые не поддерживают API EventSource, существуют полифиллы.

Так как технология SSE основана на HTTP, она отлично сочетается с HTTP/2. Её можно скомбинировать с некоторыми возможностями HTTP/2, что открывает дополнительные перспективы. А именно, HTTP/2 даёт эффективный транспортный уровень, основанный на мультиплексированных каналах, а SSE даёт приложениям API для передачи данных с сервера.

Для того, чтобы в полной мере понять возможности мультиплексирования и потоковой передачи данных, взглянем на определение IETF: *«поток» — это независимая, двунаправленная последовательность фреймов, передаваемых между клиентом и сервером в рамках соединения HTTP/2. Одна из его основных характеристик заключается в том, что одно HTTP/2-*

соединение может содержать несколько одновременно открытых потоков, причём, любая конечная точка может обрабатывать чередующиеся фреймы из нескольких потоков.



Технология SSE основана на HTTP. Это означает, что с использованием HTTP/2 не только несколько SSE-потоков могут передавать данные в одном TCP-соединении, но то же самое может быть сделано и с комбинацией из нескольких наборов SSE-потоков (отправка данных клиенту по инициативе сервера) и нескольких запросов клиента (уходящих к серверу).

Благодаря HTTP/2 и SSE теперь имеется возможность организации двунаправленных соединений, основанных исключительно на возможностях HTTP, и имеется простое API, которое позволяет обрабатывать в клиентских приложениях данные, поступающие с серверов. Недостаточные возможности в сфере двунаправленной передачи данных часто рассматривались как основной недостаток при сравнении SSE и WebSocket.

Благодаря HTTP/2 подобного недостатка больше не существует. Это открывает возможности по построению систем обмена данными между серверными и клиентскими частями приложений исключительно с использованием возможностей HTTP, без привлечения технологии WebSocket.

WebSocket и HTTP/2. Что выбрать?

Несмотря на чрезвычайно широкое распространение связки HTTP/2+SSE, технология WebSocket, совершенно определённо, не исчезнет, в основном из-за того, что она отлично освоена и из-за того, что в весьма специфических случаях у неё есть преимущества перед HTTP/2, так как она была создана для обеспечения двустороннего обмена данными с меньшей дополнительной нагрузкой на систему (например, это касается заголовков).

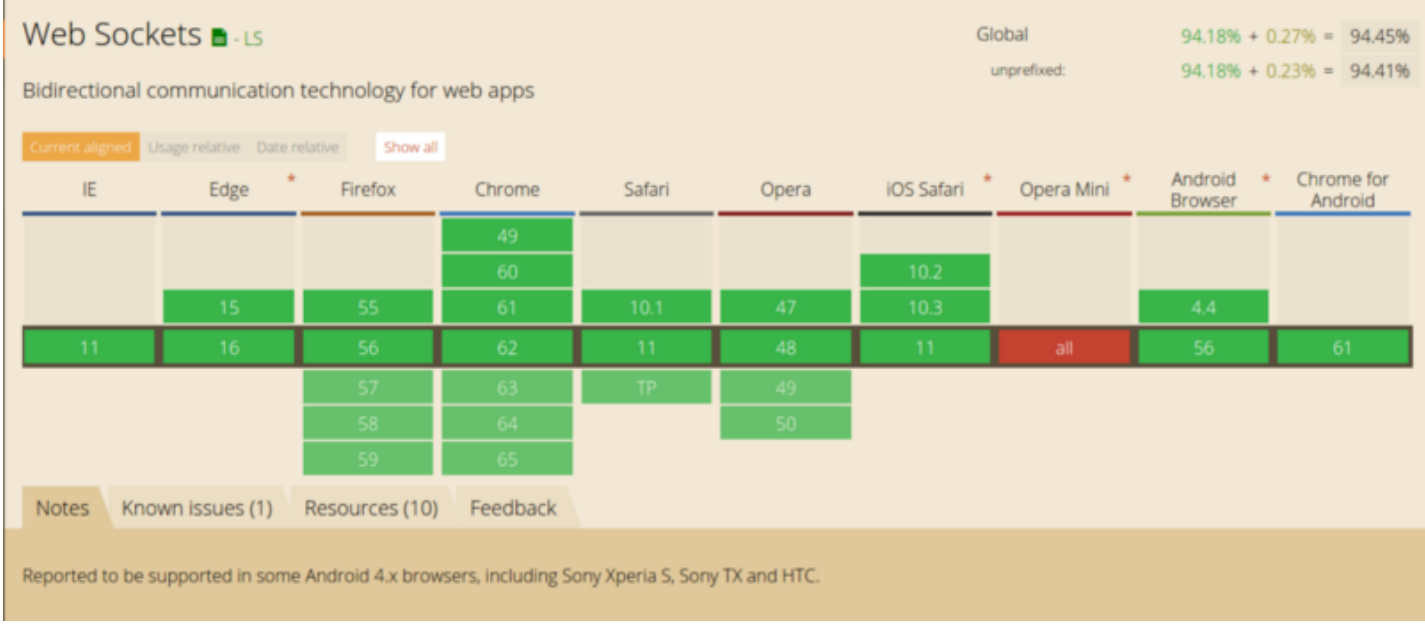
Предположим, вы хотите создать онлайн-игру, которая нуждается в передаче огромного количества сообщений между клиентами и сервером. В подобном случае WebSocket подойдёт гораздо лучше, чем комбинация HTTP/2 и SSE.

В целом, можно порекомендовать использование WebSocket для случаев, когда нужен по-настоящему низкий уровень задержек, приближающийся, при организации связи между клиентом и сервером, к обмену данными в реальном времени. Помните, что такой подход может потребовать переосмысления того, как строится серверная часть приложения, а также то, что тут может потребоваться обратить внимание на другие технологии, вроде

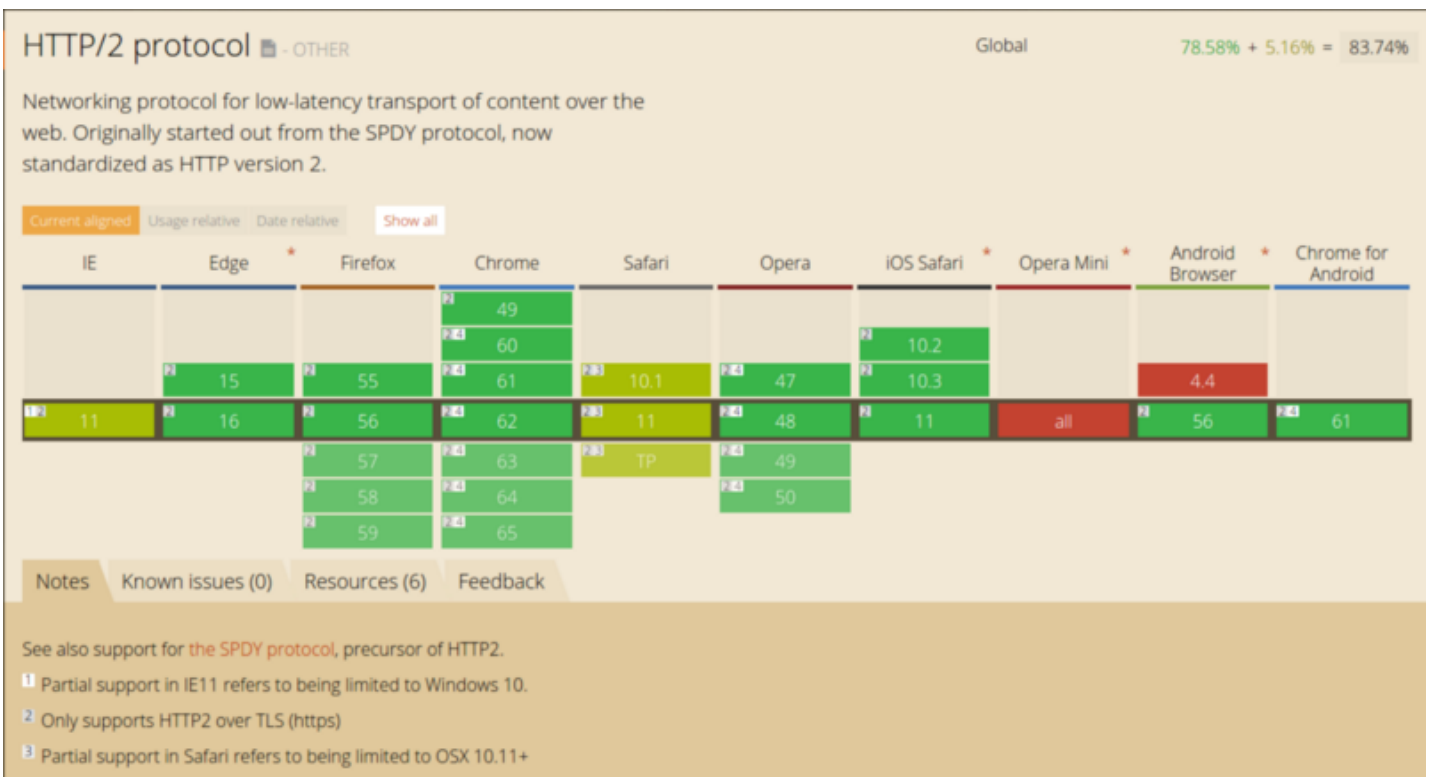
очередей событий.

Если вам нужно, например, показывать пользователям в реальном времени новости или рыночные данные, или вы создаёте чат-приложение, использование связки HTTP/2+SSE даст вам эффективный двунаправленный канал связи, и, в то же время — преимущества работы с технологиями из мира HTTP. А именно, технология WebSocket нередко становится источником проблем, если рассматривать её с точки зрения совместимости с существующей веб-инфраструктурой, так как её использование предусматривает перевод HTTP-соединения на совершенно другой протокол, ничего общего с HTTP не имеющий. Кроме того, тут стоит учесть соображения масштабируемости и безопасности. Компоненты веб-систем (файрволы, средства обнаружения вторжений, балансировщики нагрузки) создают, настраивают и поддерживают с оглядкой на HTTP. В результате, если говорить об отказоустойчивости, безопасности и масштабируемости, для больших или очень важных приложений лучше подойдёт именно HTTP-среда.

Кроме того, во внимание стоит принять и поддержку технологий браузерами. Посмотрим, как с этим дела обстоят у WebSocket:



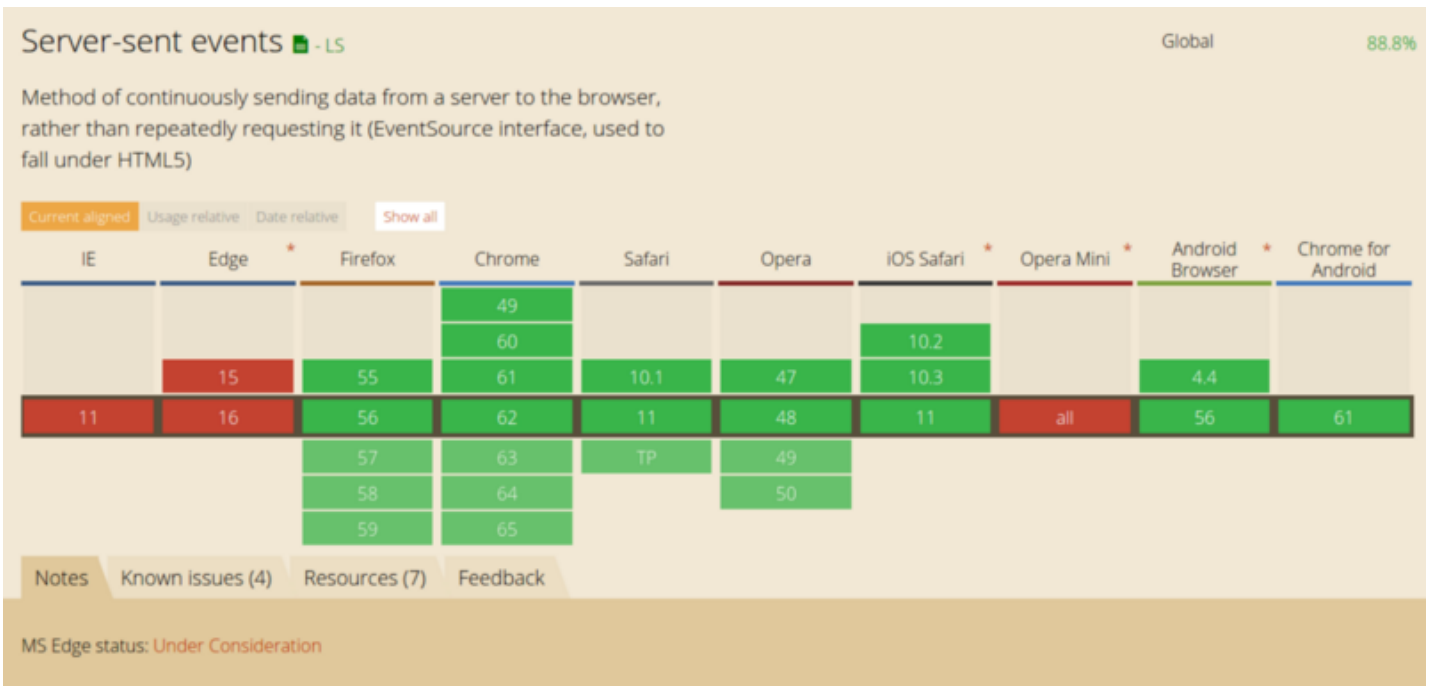
Тут всё выглядит очень даже прилично. Однако, в случае с HTTP/2 всё уже не так:



Тут можно отметить следующие особенности поддержки HTTP/2 в разных браузерах:

- Поддержка HTTP/2 только с использованием TLS (что, на самом деле, не так уж и плохо).
- Частичная поддержка в IE 11, но только в Windows 10.
- Поддержка только в OSX 10.11+ для Safari.
- Поддержка HTTP/2 только в том случае, если есть возможность пользоваться ALPN (а сервер это должен поддерживать явно).

Поддержка SSE, однако, выглядит лучше:



Не поддерживают эту технологию лишь IE/Edge. (Да, Opera Mini не поддерживает ни SSE, ни WebSocket, поэтому поддержку в этом браузере мы можем, сравнивая эти технологии, и не учитывать.) Однако, для IE/Edge существуют достойные полифиллы.

Итоги

Как видите, у технологий WebSockets и HTTP/2+SSE есть, в сравнении друг с другом, и преимущества, и недостатки. Что же

всё-таки выбрать? Пожалуй, на этот вопрос поможет ответить лишь анализ конкретного проекта и всесторонний учёт его требований и особенностей. Возможно, помощь при принятии решения окажет знание того, как эти технологии используют в уже существующих проектах. Так, автор этого материала говорит, что они, в SessionStack, используют, в зависимости от ситуации, и WebSockets, и HTTP.

Когда библиотеку SessionStack интегрируют в веб-приложение, она начинает собирать и записывать все изменения DOM, события, возникающие при взаимодействии с пользователем, JS-исключения, результаты трассировки стека, неудачные сетевые запросы, отладочные сообщения, позволяя воспроизводить проблемные ситуации и наблюдать за всем, что происходит при работе пользователя с приложением. Всё это происходит в режиме реального времени и не должно влиять на производительность веб-приложения. Администратор может наблюдать за сеансом работы пользователя прямо в процессе работы этого пользователя. В этом сценарии в SessionStack решили использовать HTTP, так как двунаправленный обмен данными тут не нужен (сервер просто передаёт данные в браузер). Использование в подобной ситуации WebSocket было бы неоправданно, привело бы к усложнению поддержки и масштабирования решения. Однако, библиотека SessionStack, интегрируемая в веб-приложение, использует WebSocket, и, только если организовать обмен данными по WebSocket невозможно, переходит на HTTP.

Библиотека собирает данные в пакеты и отправляет на сервера

SessionStack. В настоящее время реализуется лишь передача данных с клиента на сервер, но не наоборот, однако, некоторые возможности библиотеки, которые появятся в будущем, требуют двунаправленного обмена данными, именно поэтому здесь и используется технология WebSocket.

Уважаемые читатели! Пользовались ли вы технологиями WebSocket и HTTP/2+SSE? Если да — просим рассказать о том, какие задачи вы с их помощью решали, и о том, как вам понравилось то, что получилось.

Похожие публикации

Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true?

95

ПЕРЕВОД

ru_vds • 25 января в 16:08

JavaScript и ужасы мутаций

48

ПЕРЕВОД

ru_vds • 19 января в 11:42

Машины состояний и разработка веб-приложений

8

ПЕРЕВОД

ru_vds • 18 января в 12:02

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

alizar • вчера в 15:14

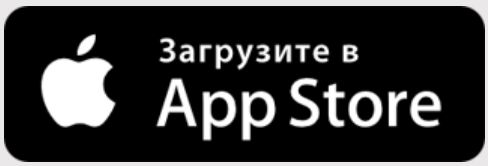
Стивен Хокинг и его работа — что дал ученый человечеству? 8

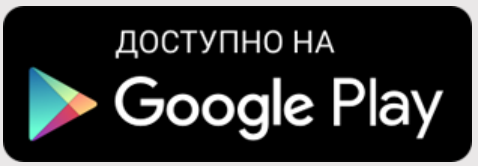
marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения 17

AlexeyNadezhin • вчера в 20:32

Мобильное приложение





Полная версия