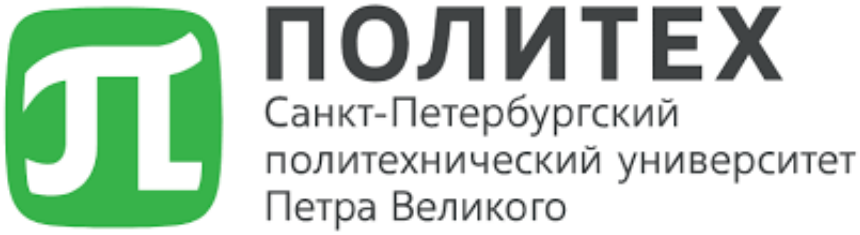
Федеральное государственное автономное образовательное
учреждение высшего образования



Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Курсовая работа

по дисциплине "Сетевая безопасность"

на тему

Исследование коммуникационного протокола WebSocket

Выполнил студент гр. 3540901/21501 _____ С.А.Мартынов
(подпись)

Преподаватель _____ В.Э. Шмаков
(подпись)

«_____» _____ 2023 г.

Санкт-Петербург
2023

Оглавление

Введение

Технология WebSocket представляет собой протокол двунаправленной связи между клиентом и сервером, который позволяет устанавливать постоянное соединение и обмениваться данными в реальном времени. В отличие от традиционного подхода, где клиент отправляет запрос на сервер, а сервер отвечает, WebSockets позволяют обоим сторонам инициировать передачу данных без задержек и лишних запросов.

Если сравнивать WebSocket с традиционными сокетами (Berkeley Socket), можно отметить что эта технология имеет более короткую историю. Формальный стандарт, описывающий протокол WebSocket и его реализацию, был опубликован рабочей группой IETF (Internet Engineering Task Force) в документе "The WebSocket Protocol"(RFC 6455) только в июле 2011 года.

WebSockets основан на протоколе HTTP и использует специальный заголовок для установки соединения между клиентом и сервером. После установки соединения, клиент и сервер могут отправлять сообщения друг другу в виде бинарных или текстовых данных. Это делает WebSockets идеальным выбором для различных приложений, где требуется обновление информации в реальном времени, таких как чаты, онлайн-игры, мониторинг и др.

Основные преимущества WebSockets включают:

- Низкая задержка: WebSockets позволяют устанавливать постоянное соединение, что устраняет необходимость в повторных запросах и сокращает задержку передачи данных.
- Эффективное использование ресурсов: Постоянное соединение WebSockets требует меньше ресурсов сервера и сети по сравнению с частыми запросами и ответами HTTP.
- Двунаправленная связь: WebSockets позволяют обмениваться данными между клиентом и сервером в обоих направлениях, что открывает возможности для интерактивного взаимодействия и обновления информации.
- Расширяемость: WebSockets поддерживают расширение протокола, что позволяет

добавлять собственные функции и возможности поверх стандартного протокола.

На сегодня, WebSockets остаются популярной технологией веб-разработки и широко применяются для создания интерактивных и реального времени веб-приложений. В данной работе мы проведём исследование основных возможностей этого протокола, рассмотрим его устройство и приведём практические примеры использования.

Глава 1

Возможности протокола WebSocket

1.1 Основные принципы WebSocket

Основные принципы WebSockets основаны на преодолении ограничений, существующих в протоколе HTTP.

1.1.1 Двунаправленная связь

Традиционный подход, основанный на протоколе HTTP, предполагает, что клиент инициирует запрос на сервер, а сервер отвечает на этот запрос. После ответа соединение закрывается. Это означает, что для каждой операции требуется инициировать новое соединение и отправить новый запрос, что может привести к значительным задержкам и накладным расходам на сеть. Кроме того, такой подход не применим для задач интерактивного взаимодействия между клиентом и сервером.

Протокол WebSocket позволяет установить постоянное двунаправленное соединение между сторонами. Двунаправленная связь означает возможность обмена данными в обоих направлениях - от клиента к серверу и от сервера к клиенту - без необходимости повторных запросов со стороны клиента. Технически, это означает что после первого запроса соединение между клиентом и сервером не завершается, а остается открытым в течение всей сессии. Таким образом обеспечивается возможность отправлять данные с сервера на клиент в любой момент без ожидания запроса от клиента. Клиент также может инициировать передачу данных серверу в режиме реального времени.

1.1.2 Инициация соединения сервером

В протоколе HTTP только клиент может инициировать соединение, отправляя запрос на сервер, а сервер может только отвечать на запросы (полудуплексное соединение). Это ограничение может быть проблематичным в ситуациях, когда серверу необходимо отправить данные клиенту без предварительного запроса от него. Сервер вынужден ожидать запроса от клиента, и только после этого отправить ответ.

Позволяя серверу инициировать передачу данных клиенту, WebSocket решает эту проблему. После установки постоянного соединения между клиентом и сервером с помощью протокола WebSocket, сервер может в любой момент отправлять данные клиенту без ожидания запроса. Это открывает возможности для мгновенного обновления информации на клиентской стороне. Такой подход может использоваться, к примеру, в системах мониторинга, когда нужно уведомить пользователя о наступлении какого-то события.

1.1.3 Поддержка реального времени

Протокол HTTP не поддерживает передачу в режиме реального времени, поскольку данные передаются только в ответ на запросы. Это затрудняет создание веб-приложений, которые требуют постоянного обновления данных в реальном времени. Чтобы клиент узнал об обновлении на сервере, ему приходилось реализовывать паттерн active waiting. Его суть сводится к тому, что клиентский код в бесконечном цикле отправляет запросы на сервер, ожидая получить обновления. Помимо избыточной нагрузки на обе стороны процесса, это порождало проблему оптимального выбора периодичности запросов: слишком частые запросы приводили с генерации мусорных пакетов в сети, а слишком редкие не позволяли достаточно оперативно забирать с сервера данные.

Благодаря WebSocket, сервер может отправить данные клиенту в момент их готовности, не ожидая запроса, обеспечивая поддержку реального времени в веб-приложениях что особенно полезно для чатов, многопользовательских игры, финансовых приложений и т. д.

1.1.4 Эффективное использование ресурсов

Основная оптимизация производительности, предоставляемая WebSocket, происходит из установлении постоянного соединения между клиентом и сервером. Повторяющиеся операции установления и завершения соединений являются довольно дорогой операцией. Особенно это чувствительно в мобильных устройствах, где есть повышенные требования к экономии ресурсов.

Следующей оптимизаций производительности является использование бинарных данных в заголовках. В традиционном HTTP все заголовки в текстовом формате ASCII и могут занимать ощутимую часть сетевого пакета. Переход к бинарным заголовкам позволяет снизить нагрузку на обработку данных и сократить объем передаваемой информации, что способствует общему ускорению работы системы.

Третья оптимизация производительности WebSocket заключается в поддерживает сжатие данных, что позволяет уменьшить размер передаваемых данных. Сервер и клиент могут договориться о методе сжатия, таком как `GZIP` или `DEFLATE`, и сжимать данные перед отправкой. Это помогает уменьшить объем данных, улучшая производительность и сокращая использование пропускной способности сети.

1.1.5 Прокси-серверы и брандмауэры

Прокси-сервера и брандмауэры играют важную роль в современных компьютерных сетях, поскольку способны обеспечить безопасность, контроль и масштабируемость, что особенно важно в корпоративных сетях. Однако использование этих технологий влечет ряд проблем для сетевого взаимодействия.

Одна из основных проблем, возникающих при из использовании, связана с поддержкой разных протоколов и стандартов. Отсутствие поддержки (или явно описанных правил) может вызывать конфликты и нежелательное поведение. Другой возможной проблемой может быть кеширование и буферизация данных. Традиционные соединения по своей природе обладают низким уровнем совместимости с прокси (к примеру, FTP), результатом чего является замедление передачи данных, а в худшем случае потеря или повреждение информации.

Протокол WebSocket разработан таким образом, что его работа осуществляется поверх протокола HTTP, что делает его совместимым с большинством прокси-серверов и фаерволов, избавляя пользователя от проблем с посредниками в коммуникации.

1.2 Архитектура WebSocket

1.2.1 Установка и завершение соединения

Протокол WebSocket предоставляет возможность установки двунаправленной связи между клиентом и сервером над одним устойчивым TCP-соединением. Большинство распростра-

нённых протоколов предполагают отправку первого пакета для рукопожатия на слушающий сокет сервера. Однако с WebSocket ситуация выглядит несколько сложнее, т.к. фактически отправляется стандартный HTTP запрос, но содержащий специальный заголовок.

В данном разделе мы пройдем через процесс установки и разрыва связи по WebSocket, а также рассмотрим суть и порядок отправления пакетов с каждой стороны.

Установка связи между клиентом и сервером начинается с выполнения HTTP-запроса с использованием метода "GET"и параметра "Upgrade". WebSocket использует этот параметр, чтобы сообщить серверу о своем желании переключиться на WebSocket-протокол.

HTTP-запрос от клиента выглядит примерно так:

```
GET /my-websocket-path HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin: http://example.com
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

Заголовок "Sec-WebSocket-Key"это уникальный ключ, который будет использоваться сервером для формирования ответа "Sec-WebSocket-Accept". Он нужен для того, чтобы сервер подтвердил, что он корректно обрабатывает именно WebSocket-запрос. Заголовок "Sec-WebSocket-Version"указывает версию протокола WebSocket. В этом примере используется версия 13, которая является наиболее актуальной на текущий момент.

После получения запроса сервер отправляет ответ (так же по HTTP), подтверждающий согласие на переход к WebSocket-протоколу. Ответ содержит "101 Switching Protocols"статус, "Upgrade: websocket"и "Connection: Upgrade"заголовки, а также "Sec-WebSocket-Accept который создается на основе ключа "Sec-WebSocket-Key"из запроса клиента.

HTTP-ответ от сервера выглядит следующим образом:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: dGhlIHNhbXBsZSBub25jZQ==
```

Подпротоколы

Помимо стандартного набора полей, существуют ещё подпротоколы (subprotocol), которые позволяют задать дополнительные заголовки `Sec-WebSocket-Extensions` и `Sec-WebSocket-Protocc` для управления потоком передачи.

Заголовок `Sec-WebSocket-Extensions: deflate-frame` указывает, что браузер поддерживает модификацию протокола, позволяющую сжимать данные. Обычно, этот заголок формируется браузером.

Заголовок `Sec-WebSocket-Protocol: soap, wamp` указывает, что браузер планирует передавать данные через WebSocket с использованием протоколов SOAP (Simple Object Access Protocol) или WAMP (WebSocket Application Messaging Protocol). Стандартные подпротоколы зарегистрированы в специальном каталоге IANA. При наличии таких заголовков сервер может использовать расширения и отправить соответствующий ответ.

Предположим, что в запросе клиент сообщил о поддержке сжатия, и готовности использовать протоколы SOAP и WAMP:

```
GET /my-websocket-path HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin: http://example.com
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap, wamp
```

Сервер в ответе сообщил о поддержке расширения deflate-frame, но из запрошенных подпротоколов только SOAP:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap
```

После успешного переключения на протокол WebSocket между клиентом и сервером становится доступным двунаправленный обмен данными через набор WebSocket-фреймов (или сообщений). Фреймы можно отправлять в любых допустимых форматах:

1. кадры, которые передают текстовую информацию представленную в кодировке utf-8;
2. кадры, которые передают данные в двоичном виде;
3. кадры, которые содержат управляющие команды (ping, pong, close).

Одна из сторон может инициировать разрыв соединения, отправляя управляющий фрейм "close"с опциональным статус кодом и причиной закрытия.

Завершение соединения

Процесс разрыва связи между клиентом и сервером следующий:

1. Одна из сторон (клиент или сервер) отправляет управляющий фрейм "close"с определенным статусом и возможным сообщением о закрытии;
2. Принимающая сторона подтверждает получение фрейма "close отправляя свой фрейм "close"с любым желаемым статусом и сообщением;
3. После успешного обмена управляющими фреймами "close"TCP-соединение закрывается;
4. Каждая из сторон выполняет нужные действия по уведомлению об отключении, очистке ресурсов и прочему.

Вебсокеты используют четырехзначные коды закрытия (event code), которые отличаются от HTTP-кодов:

- 1000: Соединение закрыто нормальным образом.
- 1001: Противоположная сторона "пропала". Это может произойти, когда серверный процесс был завершен или браузер перешел на другую страницу.
- 1002: Соединение закрыто противоположной стороной из-за ошибки в протоколе.
- 1003: Соединение закрыто, потому что противоположная сторона не может принять полученные данные. Например, если сторона, которая работает только с текстовыми данными, получает бинарное сообщение, она может закрыть соединение с данным кодом.

Таким образом, протокол WebSocket обеспечивает поддержку установки и разрыва связи между клиентом и сервером через процесс обмена HTTP-запросами и ответами, а затем переключением на двунаправленный обмен данными. Отправка пакетов (фреймов) между сторонами происходит согласно спецификации WebSocket, которая определяет формат и последовательность отправляемой информации и управляющих команд.

1.2.2 Фреймы и их структура

Формат пакета WebSocket был разработан для минимизации сложности парсинга и накладных расходов, связанных с обменом данных. Все фреймы можно разделить на две категории: data frames (фреймы с данными) и control frames (фреймы управления). Первые предназначены для передачи полезной нагрузки, а вторые обеспечивают задачи поддержки связи (PING) и закрытия соединения.

На рисунке 1.1 приведена схема фрейма из RFC6455 раздел 5.2 [2].

Пакет состоит из заголовка и полезной нагрузки. Заголовок пакета имеет длину от 2 до 14 октетов, в зависимости от модификаторов в полях заголовка. Начинатается заголовок пакета с поля `FIN` (вертикальная надпись на рисунке), затем поля `RSV1`, `RSV2`, `RSV3`. Все они занимают всего один бит. После идёт поле `oppcode` (4 бита), `MASK` (1 бит) и дальше `Payload len` (7 бит), которое показывает размер "длинны тела"пакета. Затем, если "длинна тела"равна 126 или 127, идёт "расширенная длина тела и потом (на следующей строке, то есть после первых 32 бит) будет её продолжение. После идёт ключ маски и дальше сами данные.

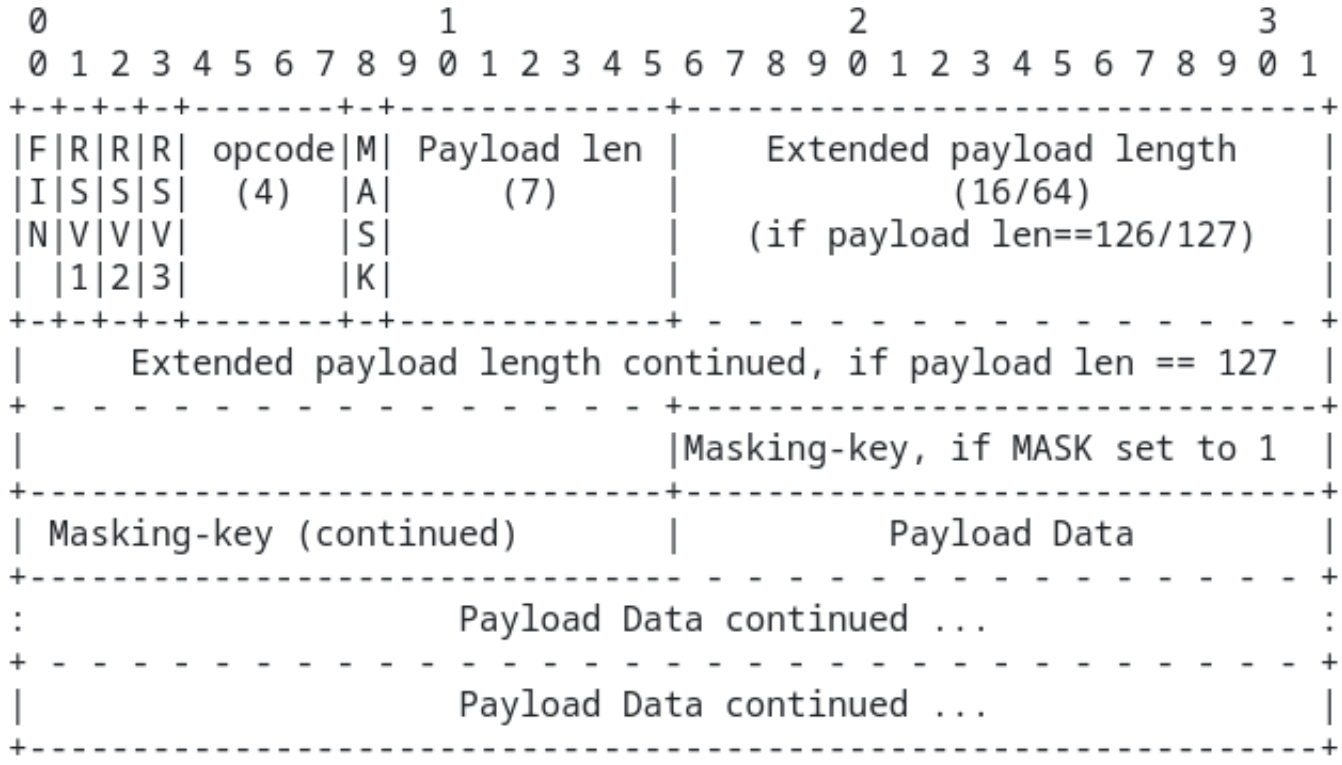


Рис. 1.1: Фрейм WebSocket протокола

Теперь подробнее рассмотрим назначение флагов и возможные значения:

- `FIN` – это 1-битовый флаг, указывающий, является ли обрабатываемый пакет последним фрагментом в сообщении. Значение 0 означает промежуточный пакет, а 1 – последний пакет. Таким образом, если передаётся длинное сообщение, то 1 будет только у последнего пакета, а если короктое – то у одного единственного (первого и последнего) пакета.
- `RSV (Reserved bits)` – это 3 битовых флага, отведенный для будущих расширений протокола. RSV1, RSV2 и RSV3 должны иметь значение 0, если они не используются каким-либо расширением.
- `OpCode` –это 4-битовый код, определяющий вид полезной нагрузки. Значение OpCode может быть:
  - `0x1` обозначает текстовый фрейм.
  - `0x2` обозначает двоичный фрейм.
  - `0x3` зарезервированы для будущих фреймов с данными.
  - `0x4` зарезервированы для будущих фреймов с данными.
  - `0x5` зарезервированы для будущих фреймов с данными.

  - `0x6` зарезервированы для будущих фреймов с данными.
  - `0x7` зарезервированы для будущих фреймов с данными.
  - `0x8` обозначает закрытие соединения этим фреймом.
  - `0x9` обозначает PING.
  - `0xA` обозначает PONG.
  - `0xB` зарезервированы для будущих управляющих фреймов.
  - `0xC` зарезервированы для будущих управляющих фреймов.
  - `0xD` зарезервированы для будущих управляющих фреймов.
  - `0xE` зарезервированы для будущих управляющих фреймов.
  - `0xF` зарезервированы для будущих управляющих фреймов.
  - `0x0` обозначает фрейм-продолжение для фрагментированного сообщения. Он интерпретируется, исходя из ближайшего предыдущего ненулевого типа.

- `MASK` – это 1-битовый флаг, указывающий, применяется ли маска к полезной нагрузке. Для всех отправляемых клиентом сообщений `MASK` должен быть равен 1, а для всех сообщений сервера — равен 0.

- `Payload len` – это поле может занимать 7 битов, 7+16 битов, или 7+64 битов и хранит длинну тела, т.е. размер передаваемых данных.

  - "0-125"указывает длину полезной нагрузки в октетах;
  - "126"указывает, что следующие 2 октета интерпретируются как 16-битное беззнаковое целое число, определяющее длину полезной нагрузки;
  - "127"указывает, что следующие 8 октетов интерпретируются как 64-битное беззнаковое целое число, определяющее длину полезной нагрузки.

  Эта сложная схема необходима для минимизации издержек: рассмотрим сообщений длиной 125 байт (и менее), тогда для хранения длины потребует всего 7 битов; для больших сообщений (в пределах 65536) – 7 битов + 2 байта; а для еще ещё больших – 7 битов и 8 байт. Этого будет достаточно для хранения длины сообщения размером в гигабайт и более.

- `Masking-key` – если флаг `MASK` установлено 0, то этого полня нет; иначе, это 4-байтное поле содержат маску, которая налагается на тело сообщения.

- `Payloat data` – После заголовка следует полезная нагрузка, которая может включать текстовые или двоичные данные. Маскировка предотвращает возникновение проблем на уровне более низких слоев сетевой инфраструктуры, связанных с неконтролируемым форматом бинарных данных.

Примеры пакетов

Не фрагментированное текстовое сообщение "Hello"без маски может выглядеть так:

```
0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f
```

Состояние флагов будет следующим:

- `FIN = 1` – это короткое сообщение, первый пакет является и последним;
- `OpCode = 0x1` – простой текстовый фрейм.

Таким образом, мы получаем `10000001` в двоичной системе счисления, или `0x81` в шестнадцатеричной. Полосе заголовка идёт размер длинны тела (`0x5`) и само тело с текстом.

Фрагментированное текстовое сообщение "Hello , "World"(из трёх частей), без маски может выглядеть так:

```
0x01 0x05 0x48 0x65 0x6c 0x6c 0x6f
0x00 0x01 0x20
0x80 0x05 0x57 0x6f 0x72 0x6c 0x64
```

Состояние флагов будет следующим:

- У первого фрейма `FIN = 0` и текстовый опкод `OpCode = 0x1`;
- у второго фрейма `FIN = 0` и `OpCode = 0x0` (при фрагментации сообщения, у всех фреймов, кроме первого, опкод пустой);
- У третьего (завершающего) фрейма `FIN = 1`.

1.2.3 WebSocket API

WebSocket объект обеспечивает API для установки и контроля веб-сокет соединения с сервером, а также для передачи и приема данных через данное соединение. На данный момент полная спецификация поддерживается всеми браузерами [3].

Конструктор WebSocket имеет один обязательный параметр и один необязательный параметр:

```
WebSocket WebSocket(
  in DOMString url,
```

```
  in optional DOMString protocols
);

WebSocket WebSocket(
  in DOMString url,
  in optional DOMString[] protocols
);
```

- `url` – ссылка для подключения; сервер по данному адресу должен откликнуться на запрос websocket.
- `protocols` (опционально) – протокол представлен в виде строки или списка строк протоколов. Эти строки служат для определения клиентских подпротоколов, поскольку один сервер может поддерживать различные WebSocket-подпротоколы (например, чтобы один сервер мог обрабатывать разные виды коммуникаций на основе указанного протокола). Если значение протокола не указано, по умолчанию используется пустая строка.

Если порт, на который устанавливается соединение заблокирован, конструктор может выбросить исключение `"SECURITY_ERR"`.

Константы состояния готовности

Константы используются атрибутом `readyState` для описания состояния WebSocket подключения

- `CONNECTING` – Соединение ещё не открыто.
- `OPEN` – Соединение открыто и готово к обмену данными.
- `CLOSING` – Соединение в процессе закрытия.
- `CLOSED` – Соединение закрыто или не может открыться.

Методы

`close()`. Закрывает WebSocket - подключение или заканчивает попытку подключения. Если подключение уже закрыто, этот метод не делает ничего.

`send()`. Передаёт данные на сервер через WebSocket - соединение.

1.3 Сравнение WebSocket с другими технологиями

1.3.1 HTTP-поллинг

Длинные опросы (long polling) – это технология, которая была разработана для решения проблемы латентности при использовании обычных опросов (веб-запросов с задержками), вызванных ограничениями стандартных HTTP-соединений. Вместо того чтобы постоянно отправлять новые запросы на сервер для проверки обновлений и тем самым создавать большую нагрузку на сервер, длинный опрос позволяет клиенту отправить один запрос, который будет "держаться"сервером, пока не появится новая информация для отправки клиенту. После отправки этой новой информации сервер закрывает соединение, и клиент немедленно запрашивает новое подключение.

Главным преимуществом длинных опросов является его совместимость с большинством существующих веб-серверов и HTTP-инфраструктуры, что упрощает его интеграцию в уже развернутые системы. Однако, длинные опросы могут столкнуться с ограничениями производительности из-за потребности в постоянной отправке новых запросов и создания новых подключений, что может вызвать задержку при передаче данных. Особенно это заметно при большом количестве активных клиентов.

В то же время, WebSocket обеспечивает более высокую производительность, так как постоянное соединение устраняет задержку и снижает расходы на обработку запросов. Кроме того, WebSocket обеспечивают лучшую поддержку двоичных данных и сжатия, что позволяет сократить объем передаваемых данных и ускорить их обмен. Недостатки WebSocket связанные с поддержкой браузерами уже в прошлом.

В зависимости от конкретного проекта и требований, можно использовать как длинные опросы, так и WebSocket. Если важно сохранение совместимости с существующими клиентами (к примеру, устаревшие версии Android) и инфраструктурой, и потребность в активной связи между клиентом и сервером не является критичной, то можно использовать длинные опросы. Однако, если проект требует быстрой и эффективной передачи данных с минимальными задержками, WebSocket является более оптимальным выбором для реализации взаимодействия между клиентом и сервером.

1.3.2 Server-Sent Events (SSE)

Server-Sent Events (SSE) представляет собой технологию, позволяющую серверу отправлять обновления клиенту через односторонний HTTP-канал без необходимости инициирования

новых запросов со стороны клиента. SSE обычно используется для отправки уведомлений или обновлений статуса устройств, что в свою очередь позволяет снизить нагрузку на сервер и повысить скорость обмена данными.

Основное отличие между Server-Sent Events (SSE) и WebSocket состоит в режиме коммуникации. SSE предоставляет только одностороннее подключение (от сервера к клиенту), в то время как WebSocket обеспечивает двустороннее общение. Это означает, что SSE идеально подходит для ситуаций, где серверу необходимо отправлять информацию клиенту, но клиент не обязан отсылать данные серверу. Однако, если клиенту требуется отправить данные серверу, например, для обработки пользовательского ввода, WebSocket будет предпочтительнее.

С точки зрения производительности и надежности, Server-Sent Events и WebSocket можно считать сравнительно равными. Обе технологии обладают низкими задержками и малым оверхедом, что делает их подходящими для использования в интерактивных приложениях. Однако стоит обратить внимание на то, что некоторые функции, доступные в WebSocket, отсутствуют в SSE. К таким функциям относится, например, возможность отправки бинарных данных.

Выбор между Server-Sent Events и WebSocket зависит от конкретных задач, которые ставятся перед разработчиками. SSE подходит для ситуаций с однонаправленным обменом данными, особенно если серверу необходимо отправлять частые обновления клиенту. WebSocket же является идеальным решением для двусторонних взаимодействий и приложений, где важна реакция на пользовательский ввод в режиме реального времени.

1.3.3 WebRTC

Технология WebRTC (Web Real-Time Communications) была разработана для обмена данными в режиме реального времени между двумя браузерами без участия централизованного сервера. Она поддерживает передачу потоковых данных, таких как аудио, видео и единственные наборы данных бинарных и обычных текстовых форм. WebRTC состоит из трех основных API: MediaStream, RTCPeerConnection и RTCDataChannel. Эти API работают совместно, чтобы установить пару pub/sub (поведенческий шаблон проектирования передачи сообщений "Издатель - подписчик") с использованием протоколов SCTP и ICE (включая STUN и TURN). Ключевым аспектом WebRTC является возможность значительно снижать задержку обмена данными и работать в условиях реального времени. Это делает технологию WebRTC чрезвычайно полезной для видео- и аудиоконференций, P2P-игры и других решений, требующих быстрого взаимодействия между участниками.

WebSocket, с другой стороны, представляет собой универсальный протокол для обмена данными между клиентом и сервером, построенный поверх протокола HTTP, который поддерживает инициацию соединения на сервере. WebSocket работает через TCP, что может привести к задержкам из-за перегрузки сети или медленной передачи данных. В отличие от WebRTC, WebSocket требует промежуточного сервера для обработки запросов от клиента, что увеличивает задержки и зависимость от сервера.

Сравнивая технические характеристики, можно отметить, что WebRTC имеет больше возможностей, включая адаптивное кодирование, снижение задержки и аппаратные возможности ускорения через GPU. WebSocket имеет более простую структуру, что облегчает развертывание и снижает нагрузку на сервер. Однако необходимость использования серверов делает WebSocket более затратной технологией по сравнению с децентрализованным подходом WebRTC.

В итоге можно сказать, что WebRTC и WebSocket представляют разные классы технологий, каждая из которых имеет свои достоинства и недостатки. WebRTC подходит для видеочатов, тогда как WebSocket решает проблему обмена сообщениями. В некоторых ситуациях возможно использование обеих технологий в одной системе, с учетом уникальных преимуществ каждой из них.

Глава 2

Реализация клиент-серверного взаимодействия по протоколу WebSocket

2.1 Пример простого чата (node.js)

Рассмотрим пример приложения, созданного на node.js.

Сервер запускается на порту 8080, и ожидает подключения клиента (листинг 2.1). В данном случае мы не обеспечиваем никакого шифрования и должной обработки ошибок, наша задача просто продемонстрировать принцип работы.

Листинг 2.1: Исходный код сервера node.js приложения

```
1  // Import required modules
2  const http = require('http');
3  const fs = require('fs');
4  const WebSocket = require('ws');
5
6  // Create a WebSocket Server instance with no server argument
7  const wss = new WebSocket.Server({ noServer: true });
8
9  // Set for keeping track of connected clients
10 const clients = new Set();
11
12 // Function for handling incoming HTTP requests and WebSocket upgrade
       ↪ requests
13 function accept(req, res) {
14   // Check if the request is a WebSocket upgrade request on the "/ws" path
15   if (req.url === '/ws' && req.headers.upgrade && req.headers.upgrade.
         ↪ toLowerCase() === 'websocket' && req.headers.connection.match(/\
```

```
      ↪ bupgrade\b/i)) {
16      wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onSocketConnect);
17    } else if (req.url === '/') { // Serve the index.html file for requests to
        ↪  the root path
18      fs.createReadStream('./index.html').pipe(res);
19    } else { // Return a 404 Not Found response for other request URLs
20      res.writeHead(404);
21      res.end();
22    }
23  }
24
25  // Function for handling new WebSocket connections
26  function onSocketConnect(ws) {
27    clients.add(ws); // Add the connected WebSocket to the clients set
28    console.log('New connection');
29
30    ws.on('message', function (message) { // Event handler for incoming
        ↪ WebSocket messages
31      console.log(`Received message: ${message}`);
32
33      message = message.slice(0, 50); // Limit message length to 50 characters
34
35      // Broadcast the message to all connected clients
36      for (let client of clients) {
37        client.send(message);
38      }
39    });
40
41    ws.on('close', function () { // Event handler for closed WebSockets
42      console.log('Connection closed');
43      clients.delete(ws); // Remove the closed WebSocket from the clients set
44    });
45  }
46
47  // Create an HTTP server that listens on port 8080 and calls the accept
      ↪ function for each request
48  http.createServer((req, res) => {
49    accept(req, res);
50  }).listen(8080);
51
52  // Log to the console that the server has started on port 8080
53  console.log("Server started on port 8080");
```

Как только клиент подключился, сервер отдаёт ему HTML страницу с кодом на JavaScript.

Этот выполняет WebSocket подключение к серверу для обмена сообщениями (листинг 2.2).

Листинг 2.2: Исходный код клиента

```
<!--
This code creates a simple real-time messaging app using HTML, CSS, and
    ↪ JavaScript. It allows users to send and receive messages.

-->

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>Message App</title>
</head>

<body>
  <!-- Create a form for sending messages -->
  <form id="publish-form">
    <input type="text" id="message-input" maxlength="50" /> <!-- input field
        ↪ to type messages -->
    <input type="submit" value="Send" /> <!-- send button -->
  </form>

  <!-- Create a div to display messages -->
  <div id="messages"></div>

  <script>
    // Set WebSocket URL
    const wsUrl = 'ws://localhost:8080/ws';
    // Create WebSocket connection
    const ws = new WebSocket(wsUrl);

    // Get HTML elements
    const form = document.getElementById('publish-form');
    const inputField = document.getElementById('message-input');
    const msgDiv = document.getElementById('messages');

    // Add event listener to send messages upon form submission
    form.addEventListener('submit', (event) => {
      event.preventDefault();
      const msgToSend = inputField.value;

      ws.send(msgToSend);
```

```
41      });
42
43      // Add event listener to listen for incoming messages
44      ws.addEventListener('message', async (event) => {
45        const msgReceived = await new Response(event.data).text()
46        displayMessage(msgReceived);
47      });
48
49      // Add event listener to listen for WebSocket closing
50      ws.addEventListener('close', (event) => {
51        console.log(`Closed ${event.code}`);
52      });
53
54      // Function to display messages in the div#messages
55      function displayMessage(message) {
56        const msgElement = document.createElement('div');
57        msgElement.textContent = message;
58        console.log(message);
59        msgDiv.prepend(msgElement);
60      }
61    </script>
62  </body>
63
64  </html>
```

Если открыть это соединение из двух разных браузеров, можно устроить простой чат (рисунок 2.1).

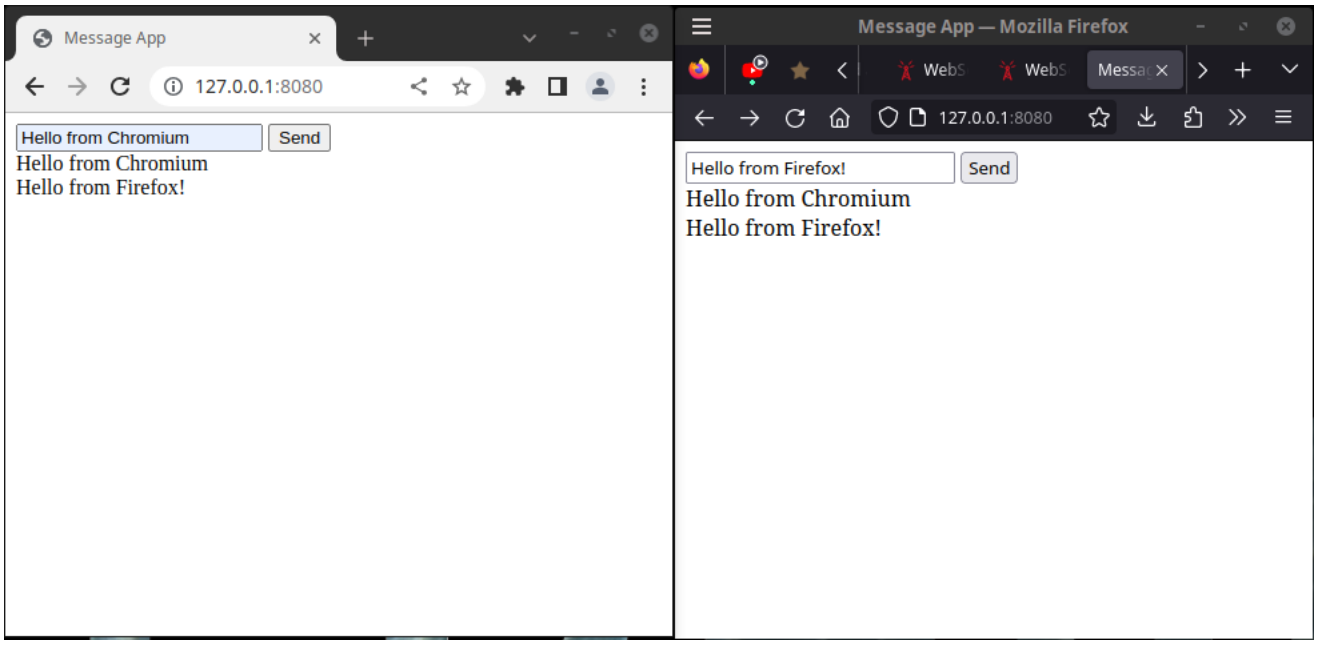


Рис. 2.1: Взаимодействие между двумя разными браузерами

В это время в терминале можно видеть сообщения, которыми обмениваются клиенты (рисунок 2.2).

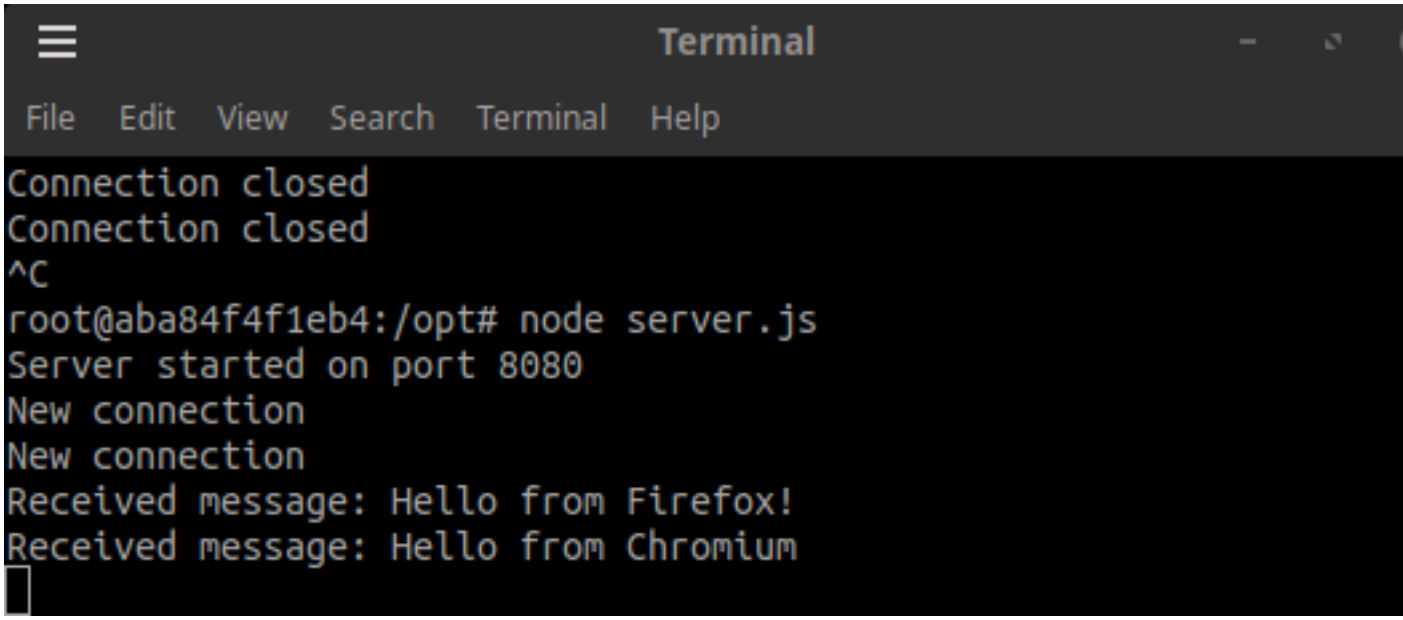


Рис. 2.2: Консоль серверного приложения

2.2 Пример тестера скорости соединения (Rust)

Второй пример представляет из себя немного более сложный код. Идея проекта в том, чтобы передать через WebSocket соединение файл объёмом 10 мб и замерить скорость передачи.

Сервер так же запускается на порту 8080 (листинг 2.3), и готов отдать пользователю статичный HTML файл.

Листинг 2.3: Исходный код сервера Rust приложения

```rust
// Import necessary packages and modules
use actix_files::NamedFile;
use actix_web::{
    middleware, web, App, Error, HttpRequest, HttpResponse, HttpServer,
        ↪ Responder,
};
use actix_web_actors::ws;
use env_logger;

// Import local server module
mod server;
use server::MyWebSocket;

// Serve index.html asynchronously
async fn serve_index() -> impl Responder {
    NamedFile::open_async("./static/index.html").await.unwrap()
}

// Serve WebSocket asynchronously
async fn serve_websocket(req: HttpRequest, stream: web::Payload) -> Result<
    ↪ HttpResponse, Error> {
    ws::start(MyWebSocket::new(), &req, stream)
}

// Entry point of the application with async main
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    // Initialize logger using environment variables, default level: "info"
    env_logger::init_from_env(env_logger::Env::new().default_filter_or("info
        ↪ "));

    // Log a message about the server starting
    log::info!("starting HTTP server at http://localhost:8080");

```

```
32      // Create and configure the HTTP server
33      HttpServer::new(|| {
34          App::new()
35              .route("/", web::get().to(serve_index)) // Serve index.html
36              .route("/ws", web::get().to(serve_websocket)) // Serve WebSocket
37              .wrap(middleware::Logger::default()) // Add logging middleware
38      })
39      .workers(2) // Set the number of worker threads
40      .bind("127.0.0.1:8080")? // Bind server to address
41      .run() // Start the server
42      .await // Wait for the server to finish gracefully
43  }
```

Управление раутами и временем жизни соединения в этот раз вынесено в отдельный файл (листинг 2.4)

Листинг 2.4: Управление WebSocket соединением

```
1  // Import necessary modules from std and external crates
2  use std::fs::File;
3  use std::io::BufReader;
4  use std::io::Read;
5  use std::time::{Duration, Instant};
6
7  use actix::prelude::*;
8  use actix_web::web::Bytes;
9  use actix_web_actors::ws;
10
11 // Configure constants for heartbeat (how often it sends) and client timeout
12 const HEARTBEAT_INTERVAL: Duration = Duration::from_secs(5);
13 const CLIENT_TIMEOUT: Duration = Duration::from_secs(10);
14
15 // WebSocket actor
16 pub struct MyWebSocket {
17     hb: Instant, // Store the time when the last heartbeat message was
       ↪ received
18 }
19
20 // Implement MyWebSocket
21 impl MyWebSocket {
22     // Constructor
23     pub fn new() -> Self {
24         Self { hb: Instant::now() }
25     }
26
```

```
27      // Schedule regular heartbeat messages, and stop the WebSocket
            ↪ connection if timeout
28      fn schedule_heartbeat(&self, ctx: &mut <Self as Actor>::Context) {
29          ctx.run_interval(HEARTBEAT_INTERVAL, |act, ctx| {
30              if Instant::now().duration_since(act.hb) > CLIENT_TIMEOUT {
31                  ctx.stop();
32                  return;
33              }
34
35              ctx.ping(b"");
36          });
37      }
38  }
39
40  // Implement Actor for MyWebSocket
41  impl Actor for MyWebSocket {
42      type Context = ws::WebsocketContext<Self>;
43
44      // When the actor starts, schedule the heartbeat messages
45      fn started(&mut self, ctx: &mut Self::Context) {
46          self.schedule_heartbeat(ctx);
47      }
48  }
49
50  // Implement StreamHandler with Result type for handling WebSocket events
51  impl StreamHandler<Result<ws::Message, ws::ProtocolError>> for MyWebSocket {
52      // Handle incoming WebSocket messages (ping, pong, text, close)
53      fn handle(&mut self, msg: Result<ws::Message, ws::ProtocolError>, ctx: &
            ↪ mut Self::Context) {
54          match msg {
55              Ok(ws::Message::Ping(msg)) => {
56                  self.hb = Instant::now(); // Update the heartbeat time
57                  ctx.pong(&msg); // Respond with a pong message
58              }
59              Ok(ws::Message::Pong(_)) => {
60                  self.hb = Instant::now(); // Update the heartbeat time
61              }
62              Ok(ws::Message::Text(_)) => {
63                  let file = File::open("./static/Sample10mb").unwrap();
64                  let mut reader = BufReader::new(file);
65                  let mut buffer = Vec::new();
66
67                  reader.read_to_end(&mut buffer).unwrap();
68                  ctx.binary(Bytes::from(buffer)); // Send the binary data
                        ↪ from a file
```

```
69              }
70              Ok(ws::Message::Close(reason)) => {
71                  ctx.close(reason); // Close the WebSocket connection
72                  ctx.stop(); // Stop the WebSocket Actor
73              }
74              _ => ctx.stop(), // Stop the WebSocket Actor for other scenarios
75          }
76      }
77  }
```

Клиентское приложение представляет из себя HTML страницу (листинг 2.5). Её код тривиален, поэтому в листинге обратим внимание только на JavaScript код.

Листинг 2.5: Исходный код сервера node.js приложения

```
1           // Anonymous function that runs on page load
2           (function () {
3               // Select DOM elements
4               const startButton = document.querySelector("#start");
5               const logElement = document.querySelector("#log");
6
7               // Calculate the average of an array
8               const average = (array) => array.reduce((a, b) => a + b) / array
                    ↪ .length;
9               const totalTests = 10;
10              let startTime, endTime, testResults = [];
11              let socket = null;
12
13              // Log messages to the log element
14              function log(msg, type = "status") {
15                  logElement.innerHTML += `<p class="msg msg--${type}">${msg
                        ↪ }</p>`;
16                  logElement.scrollTop += 1000;
17              }
18
19              // Start the test by creating a WebSocket connection
20              function start() {
21                  complete();
22
23                  const wsUri = (location.protocol.startsWith("https") ? "wss"
                        ↪  : "ws") + "://" + location.host + "/ws";
24                  let testsRun = 0;
25
26                  log("Starting...");
27                  socket = new WebSocket(wsUri);
```

```
28
29                  socket.onopen = () => {
30                      log("Started.");
31                      updateTestStatus();
32                      testsRun++;
33                      startTime = performance.now();
34                      socket.send("start");
35                  };
36
37                  socket.onmessage = (ev) => {
38                      endTime = performance.now();
39                      log(`Completed Test: ${testsRun}/${totalTests}. Took ${
                            ↪ endTime - startTime} milliseconds.`);
40                      testResults.push(endTime - startTime);
41
42                      if (testsRun < totalTests) {
43                          testsRun++;
44                          startTime = performance.now();
45                          socket.send("start");
46                      } else complete();
47                  };
48
49                  socket.onclose = () => {
50                      log("Finished.", "success");
51                      socket = null;
52                      updateTestStatus();
53                  };
54              }
55
56              // Complete the test and cleanup
57              function complete() {
58                  if (socket) {
59                      log("Cleaning up...");
60                      socket.close();
61                      socket = null;
62
63                      const testAverage = average(testResults) / 1000;
64                      const mbps = 10 / testAverage;
65                      const status = mbps < 10 ? "bad" : mbps < 50 ? "" : "
                            ↪ good";
66
67                      log(`Average speed: ${mbps.toFixed(2)} MB/s or ${(mbps *
                            ↪  8).toFixed(2)} Mbps`, status);
68                      updateTestStatus();
69                  }
```

```
70              }
71
72              // Update the button status based on test state
73              function updateTestStatus() {
74                  if (socket) {
75                      startButton.disabled = true;
76                      startButton.innerHTML = "Running";
77                  } else {
78                      startButton.disabled = false;
79                      startButton.textContent = "Start";
80                  }
81              }
82
83              // Handle button click
84              startButton.addEventListener("click", () => {
85                  if (socket) complete();
86                  else start();
87                  updateTestStatus();
88              });
89
90              // Initialize the test status
91              updateTestStatus();
92              log('Click "Start" to begin.');
93          })();
```

В панели веб-разработчика можно видеть, что файл действительно был передан именно через WebSocket (рисунок 2.3).

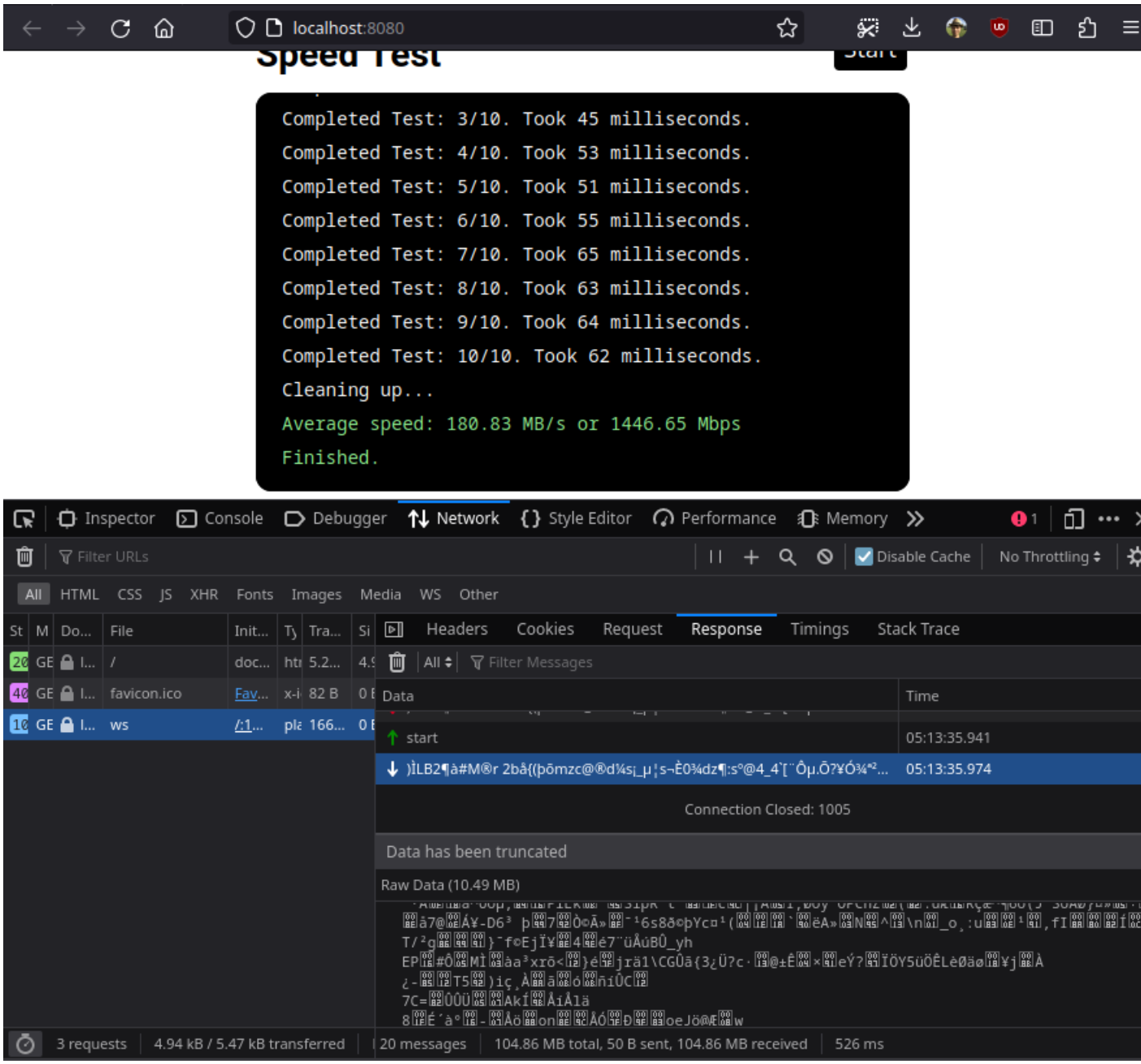


Рис. 2.3: Панель веб-разработчика

Глава 3

Применение WebSocket в Web-среде

Использование WebSocket на производственных серверах в условиях реального веба требует повышенного внимания к безопасности и производительности.

3.1 Безопасность WebSocket

Хотя WebSockets предлагают мощное средство для разработки приложений коммуникации в реальном времени, они также создают определенные угрозы безопасности, которые следует принимать во внимание. Вот некоторые распространенные проблемы безопасности, связанные с WebSockets, а также стратегии их устранения:

Перехват WebSocket между сайтами (CSWSH): возникает, когда злоумышленник использует уязвимость XSS для кражи соединения WebSocket и связи с сервером. Для предотвращения этого, разработчику важно убедиться, что соединение WebSocket устанавливается исключительно между доверенными сторонами. Внедрение аутентификации на стороне сервера поможет предотвратить такие атаки.

Подделка межсайтовых запросов (CSRF): возникает, когда злоумышленник отправляет запрос WebSocket на сервер от имени другого пользователя. Для предотвращения этого, важно использовать CSRF-токены, чтобы обеспечить прием запроса WebSocket только от доверенных источников.

Инъекционные атаки: они возникают, когда злоумышленник внедряет вредоносный код или данные в запрос или ответ WebSocket. Для предотвращения атак с использованием инъекций, важно скрупулезно проверять и фильтровать все данные, отправляемые через соединение WebSocket. Реализация проверки входящих данных и кодирования исходящих

данных может помочь избежать инъекционных атак.

Атаки "отказ в обслуживании"(DoS): возникают, когда злоумышленник перегружает сервер большим числом запросов WebSocket, что может привести к сбою сервера. Для предотвращения DoS-атак важно внедрить ограничение скорости и дросселирование, чтобы снизить количество запросов, которые сервер способен обрабатывать.

Атаки "человек посередине"(MitM): возникают, когда злоумышленник перехватывает трафик WebSocket и просматривает или изменяет данные, передаваемые между клиентом и сервером. Чтобы предотвратить атаки MitM, важно внедрить меры шифрования и аутентификации, такие как SSL/TLS, чтобы обеспечить безопасность трафика WebSocket и защиту от перехвата или изменения.

В общем, важно надлежащим образом обезопасить соединение WebSocket, внедряя такие меры, как проверка вхождений, кодирование выходных данных, аутентификация, шифрование, ограничение скорости и CSRF-токены. Следуя рекомендациям по обеспечению безопасности WebSocket, разработчики смогут предотвратить типичные проблемы безопасности и обеспечить надежность и безопасность их приложений для обмена данными в реальном времени.

3.1.1 Шифрование и обеспечение конфиденциальности

Протокол WebSocket поддерживается практически всеми современными браузерами включая Chrome, Firefox, Internet Explorer, Opera, и Safari. Однако по спецификации, этот протокол не обеспечивает шифрования, а значит может быть перехвачен, прочитан и даже изменён. Поскольку WebSocket это централизованный протокол (т.е. предполагает наличие сервера) то проблему защиты данных нужно решать на стороне серверных платформ начинает поддерживать этот протокол.

NGINX с одной стороны умеет терминировать SSL/TLS траффик, с другой – поддерживает WebSocket, позволяя установить туннель между клиентом и сервером бэкэнда[4]. Чтобы NGINX отправил запрос на обновление от клиента к серверу бэкэнда, заголовки Upgrade и Connection должны быть установлены явно, как в этом примере:

```
location /wsapp/ {
    proxy_pass http://wsbackend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
    proxy_set_header Host $host;
```

```
}
```

После этого NGINX обрабатывает это как соединение WebSocket.

3.1.2 Атака “отравленный кэш“

В ранних реализациях WebSocket существовала уязвимость, известная как «cache poisoning» (отравленный кэш)[1]. Эта уязвимость позволяла атаковать кэширующие прокси-серверы, особенно корпоративные системы.

Атака выполнялась следующим образом:

1. Злоумышленник привлекает доверчивого пользователя (далее - Жертва) на свой сайт.
2. На сайте Злоумышленника, веб-страница открывает WebSocket-соединение из браузера Жертвы к серверу Злоумышленника. Предполагается, что жертва использует прокси-сервер, который является целью атаки.
3. Веб-страница создает особый WebSocket-запрос, который некоторые прокси-сервера не могут распознать (и в этом есть условие для успешной атаки).
4. Прокси-сервер Жертвы пропускают начальный запрос (содержащий Connection: upgrade) и полагает, что следующий HTTP-запрос уже поступил. Однако на самом деле вместо этого данные передаются через WebSocket! Обе стороны WebSocket (веб-страница и сервер) контролируются Злоумышленником!
5. Злоумышленник может передать что-то, похожее на GET-запрос к известному ресурсу, например: http://code.jquery.com/jquery.js. Тогда сервер ответит, имитируя код jQuery с кэширующими заголовками. Прокси-сервер получит этот ответ и закэширует искажённую версию jQuery.
6. В результате, при загрузке новых страниц, другие пользователи, использующие тот же прокси-сервер, что и Жертва, получат хакерский код вместо настоящего jQuery (это и объясняет название атаки - «отравленный кэш».).

Эта атака не работает для всех прокси-серверов, однако при исследовании уязвимости было доказано, что реальные уязвимые прокси-серверы существуют.

Для предотвращения такой атаки разработали метод защиты - использование «маски».

Маска для защиты от атаки

Маска, которую мы рассматривали ранее при изучении полей WebSocket пакета, создана для предотвращения атаки отравленного кэша.

Ключ маски – это случайное 32-битное значение, которое меняется для каждого пакета данных.

Тело сообщения проходит через `XOR` $\oplus$ с маской, а получатель восстанавливает данные, применяя повторно `XOR` с маской (можно легко доказать, что $(x \oplus a) \oplus a == x$).

Маска служит двум целям:

1. Генерируется браузером. Таким образом, злоумышленник больше не может контролировать реальное содержимое тела сообщения. После наложения маски данные превращаются в бинарный беспорядок.
2. Полученный пакет данных уже не может быть воспринят промежуточным прокси как HTTP-запрос.

Наложение маски требует дополнительных ресурсов, поэтому в протоколе WebSocket оно не является обязательным.

Если используется два клиента (не обязательно браузеры), которые доверяют друг другу и посредникам, можно установить бит маски в 0, и тогда не потребуется указывать ключ маски.

3.2 Производительность и масштабируемость

Когда речь заходит о промышленном применении и высоких нагрузках, встаёт вопрос масштабируемости.

Вертикальное и горизонтальное масштабирование

Первый тип - вертикальное масштабирование, что является простым путем увеличения производительности приложения, хотя и имеет свои ограничения. Это, в перву очередь, ресурсы: берётся один экземпляр приложения и улучшаем аппаратное обеспечение (процессор, объем памяти, ввод-вывод и т.д.). Этот метод не требует дополнительных затрат с точки зрения разработки, однако не является самым эффективным для масштабирования.

Главным недостатком вертикального масштабирования является то, что время исполнения кода не изменяется линейно с улучшением оборудования. Кроме того, существует ограничение на возможности аппаратных усовершенствований - нет возможности неограниченно повышать производительность процессора.

Значит, есть другая альтернатива - горизонтальное масштабирование, которое предполагает создание дополнительных экземпляров приложения вместо добавления ресурсов к существующим. Это позволяет практически неограниченно масштабировать систему с возможностью динамического масштабирования при изменении нагрузок.

С точки зрения разработки, у нас появляется дополнительный элемент в системе – балансировщик нагрузки. А в некоторых случаях, придётся внести изменения в архитектуру, добавив реализацию pub/sub.

Проблема состояния и липкая сессия

Простые REST API сервера не хранят состояние. И горизонтальное масштабирование для таких серверов превращается в тривиальную задачу: мы можем случайным образом раскидывать запросы клиентов, и нам не важно какой экземпляр приложения этот запрос обработает.

Вот так может выглядеть конфиг для балансировщика HAProxy [5]

```
defaults
  mode http
  option http-server-close
  timeout connect 5s
  timeout client 30s
  timeout client-fin 30s
  timeout server 30s
  timeout tunnel 1h
  default-server inter 1s rise 2 fall 1 on-marked-down shutdown-sessions
  option forwardfor

frontend all
  bind 127.0.0.1:8080
  default_backend backends

backend backends
  server srv1 127.0.0.1:8081 check
  server srv2 127.0.0.1:8082 check
```

с WebSocket ситуация иначе, так как каждое сокетное соединение привязано к определенному экземпляру. В таком случае, нужно гарантировать, что все запросы отдельных пользователей направляются к нужному бэкэнду.

Для решения этой проблемы можно использовать липкие сессии. Балансировщик нагрузки HAProxy вместо циклического перебора будет использовать стратегию с минимальным количеством подключений и добавим куки для привязки запросов пользователей к определенному бэкэнду. Тогда запрос от одного пользователя будет приходить всегда на один и тот же сервер.

```
backend backends
  balance leastconn
  cookie serverid insert
  server srv1 127.0.0.1:8081 check cookie srv1
  server srv2 127.0.0.1:8082 check cookie srv2
```

Рассылка широковещательных сообщений

Предположим, что возникает необходимость отправить сообщение сразу всем пользователям системы. Либо пользователи подписаны на один канал, при этом подключены к разным физическим серверам. Очевидно, что решить эту задачу в рамках одного сервера невозможно, т.к. он просто не знает о всех пользователях.

Решение этой задачи заключается в организации связи между разными экземплярами приложения на разных серверах. Для этого все экземпляры могут быть подписаны на определенный канал и обрабатывать входящие сообщения. Это применение паттерна pub/sub о котором мы уже говорили ранее. Существует большое количество готовых продуктов подобного рода, таких как Redis, Kafka или Nats.

Заключение

В ходе проведенного исследования были рассмотрены основные возможности протокола WebSocket, его устройство и приведены практические примеры использования. Целью работы было изучить протокол WebSocket и понять его преимущества и особенности по сравнению с другими технологиями для обмена данными в режиме реального времени.

Основные принципы WebSocket, такие как двунаправленная связь, инициация соединения сервером, поддержка реального времени, эффективное использование ресурсов и совместимость с прокси-серверами и брандмауэрами, делают его идеальным выбором для приложений, требующих непрерывного обмена данными между клиентом и сервером.

Архитектура WebSocket, включающая установку и завершение соединения, фреймы и их структуру, а также WebSocket API, предоставляет гибкую платформу для разработки приложений реального времени. Протокол WebSocket также был сравнен с другими технологиями, такими как HTTP-поллинг, Server-Sent Events (SSE) и WebRTC, и показал высокий уровень в области производительности и эффективности.

В работе были представлены практические примеры использования WebSocket, включая реализацию простого чата на платформе node.js и создание тестера скорости соединения на языке программирования Rust. Эти примеры демонстрируют, как легко и эффективно можно использовать протокол WebSocket для различных типов приложений.

Выводы по работе:

1. Протокол WebSocket - полезная и востребованная технология для реализации обмена данными между клиентом и сервером в режиме реального времени. Это достигается благодаря двунаправленной связи, низкому времени задержки и эффективному использованию ресурсов.

2. WebSocket обладает рядом преимуществ перед другими технологиями, что делает его актуальным выбором для решения задач, связанных с обменом данными в режиме реального времени.

3. Несмотря на свои преимущества, важно учесть возможные риски, связанные с безопасностью и производительностью при использовании WebSocket в конкретном проекте, и применить соответствующие меры для смягчения этих рисков.

4. Практические примеры использования WebSocket успешно применяются для решения различных задач, свидетельствующих об универсальности и вариативности этой технологии.

Таким образом, в результате исследования и анализа данного протокола можно сделать вывод о его значимости для обеспечения непрерывного и эффективного обмена данными в режиме реального времени. Его преимущества, такие как низкая задержка, масштабируемость, поддержка расширений и простота использования, делают его идеальным выбором для разработки современных веб-приложений. Протокол WebSocket открывает новые возможности в области интерактивных веб-приложений, многопользовательских игр, финансовых торговых платформ и других приложений, где важна своевременная доставка данных и низкая задержка взаимодействия.

Литература

[1] JavaScript.ru: Справочник по работе с WebSocket. URL: https://learn.javascript.ru/websockets (дата обращения: 16.04.2023).

[2] Internet Engineering Task Force (IETF) / Request for Comments: 6455: The WebSocket Protocol (December 2011). URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата обращения: 20.04.2023).

[3] MDN Web Docs / The WebSocket API (WebSockets). URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (дата обращения: 20.04.2023).

[4] Nginx blog / NGINX as a WebSocket Proxy (Rick Nelson, 2014). URL: https://www.nginx.com/blog/websocket-nginx/ (дата обращения: 23.04.2023).

[5] How to scale WebSocket – horizontal scaling with WebSocket tutorial (2020). URL: https://tsh.io/blog/how-to-scale-websocket/ (дата обращения: 26.04.2023).