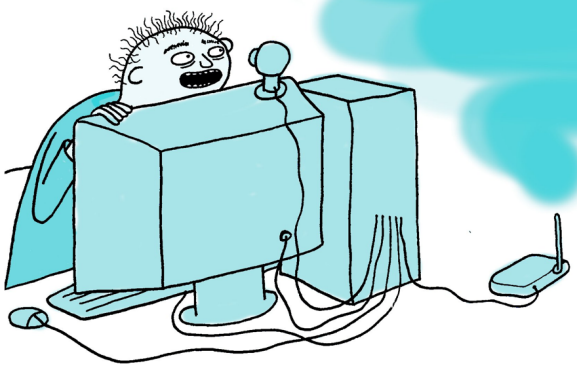
WebRTC
для любопытных



ЧЕРНОВИК

Sean DuBois
Claes Mogren
Alex Zhukov
и команда webrtcforthecurious.com

https://github.com/webrtc-for-the-curious/webrtc-for-the-curious

Contents

WebRTC для любопытных

Книга написана разработчиками WebRTC(Web Real-Time Communication) с целью поделиться с миром полученными знаниями из опыта. *WebRTC для любопытных* это - книга с открытым исходным кодом для тех, кто любит копнуть глубже. Это книга не нацелена на описание абстракции.

Книга полностью покрывает описание протоколов и API.Также для удобства собраны выводы из RFCs и все недокументированные знания в одном месте. Эта книга не является заменой учебных пособии и не является таковым и тут не много кода.

WebRTC является прекрасной технологией, но она также сложна в использовании. Книга не несет в себе конфликт интересов(vendor agnostic).

Для кого это книга:

- Для разработчиков, которые не знают, какие проблемы решает WebRTC, и хотят узнать об этом побольше.
- Для тех, кто уже имел опыт использование WebRTC, но хочет углубить свои знания.
- Для опытных разработчиков, кому нужно помощь с отладкой.
- WebRTC разработчику, который нуждается в подробном описаний в специфической части WebRTC.

Система чтения

Книга написана удобным для многократного чтения способом.Вы можете переходить к главам как вам хочется и вы не потеряетесь, каждая глава написана как отдельная статья.

Каждая глава дает ответы на 3 вопроса:

- Какую проблему нужно решить?
- Как это решить? (Включая технические подробности о решений)
- Где можно узнать о теме побольше?

Главы написаны как отдельные статьи и не требуют предварительных знаний по теме. Вы можете перейти в любую точку книги и сразу же начать изучать тему. В книге также будут рекомендации и ссылки где можно будет узнать по каждой теме побольше. Другие книги охватывают отдельные темы гораздо глубже. Эта книга призвана научить вас основам и тонкостям WebRTC, написанная экспертами.

В свободном доступе и с соблюдением конфиденциальности

Книга доступна на GitHub и WebRTCforTheCurious.com. Лицензия книги дает вам свободу использовать ее как вы считаете нужным. Вы также можете скачать текущую русскую версию книги как ePub или PDF файл.

Книга написана обычными людьми для обычных людей. Она не несет в себе конфликт интересов(vendor agnostic).

Данный вебсайт не использует аналитику или систему отслеживания.

Попробуй и ты!

Мы нуждаемся в твоей помощи! Эта книга полностью разработана на GitHub и еще пишется. Мы лояльны и поощеряем пользователей активно участвовать в разработке и открывать issue по темам, которые, по вашему мнению, требуют более подробного разъяснения.

Лицензия

Эта книга доступна по лицензий CC0. Авторы отказались от всех своих авторских и смежных прав на свои труды в этой книге в максимальной степени, разрешенной законом. Вы можете использовать эту книгу, как хотите, без указания авторства.

Что, Зачем и Как

Что такое WebRTC?

WebRTC, сокращение от Web Real-Time Communication, это одновременно и API, и Протокол. Протокол WebRTC - это набор правил для двух агентов WebRTC для согласования двунаправленной безопасной связи в реальном времени. API WebRTC позволяет разработчикам использовать протокол WebRTC. API WebRTC определен только для JavaScript.

Похожие отношения можно провести между HTTP и Fetch API. WebRTC как протокол будет HTTP, а WebRTC как API будет Fetch API.

Протокол WebRTC доступен в других API и языках помимо JavaScript. Вы также можете найти серверы и специализированные инструменты для WebRTC. Все эти реализации используют протокол WebRTC, чтобы они могли взаимодействовать друг с другом.

Протокол WebRTC поддерживается в IETF в рабочей группе rtcweb. API WebRTC документирован в W3C как webrtc.

Почему я должен изучать WebRTC?

Вот некоторые вещи, которые даст вам WebRTC:

- Открытый стандарт
- Множество реализаций
- Доступность в браузерах
- Обязательное шифрование
- Обход NAT
- Повторное использование существующих технологий
- Контроль перегрузки
- Задержка менее секунды

Этот список не исчерпывающий, а всего лишь пример некоторых вещей, которые вы можете оценить в процессе изучения. Не волнуйтесь, если вы пока не знаете всех этих терминов, эта книга научит вас им по мере прочтения.

Протокол WebRTC - это набор других технологий

Протокол WebRTC - это огромная тема, которая потребовала бы целой книги для объяснения. Однако, для начала мы разделим его на четыре шага:

1. Сигнализация
2. Подключение
3. Защита
4. Коммуникация

Эти шаги последовательны, что означает, что предыдущий шаг должен быть на 100% успешным, прежде чем начнется следующий.

Любопытный факт о WebRTC заключается в том, что каждый шаг на самом деле состоит из многих других протоколов! Чтобы создать WebRTC, мы объединяем многие существующие технологии. В этом смысле, вы можете думать о WebRTC скорее как о комбинации и конфигурации хорошо понятных технологий, восходящих к началу 2000-х, чем как о совершенно новом процессе.

Каждому из этих шагов посвящена отдельная глава, но сначала полезно понять их на высоком уровне. Поскольку они зависят друг от друга, это поможет объяснить далее цель каждого из этих шагов.

Сигнализация: Как пиры находят друг друга в WebRTC

Когда агент WebRTC начинает работу, он не знает, с кем будет общаться и о чем. Шаг *Сигнализации* решает эту проблему! Сигнализация используется для начальной загрузки вызова, позволяя двум независимым агентам WebRTC начать общение.

Сигнализация использует существующий текстовый протокол SDP (Session Description Protocol). Каждое SDP-сообщение состоит из пар “ключ-значение” и содержит список “медиа-секций”. SDP, который обмениваются два агента WebRTC, содержит такие детали как:

- IP-адреса и порты, на которых агент доступен (кандидаты).
- Количество аудио- и видеотреков, которые агент хочет отправить.
- Аудио- и видеокодеки, поддерживаемые каждым агентом.
- Значения, используемые при подключении (`uFrag/uPwd`).
- Значения, используемые при защите (отпечаток сертификата).

Очень важно отметить, что сигнализация обычно происходит “вне полосы”, что означает, что приложения обычно не используют сам WebRTC для обмена сигнальными сообщениями. Должен существовать другой канал связи между двумя сторонами перед началом подключения WebRTC. Тип используемого канала не является заботой WebRTC. Любая архитектура,

подходящая для отправки сообщений, может передавать SDP между подключающимися пирами, и многие приложения просто используют свою существующую инфраструктуру (например, REST-эндпоинты, WebSocket-соединения или прокси-серверы аутентификации) для обмена SDP между соответствующими клиентами.

Подключение и обход NAT с помощью STUN/TURN

После того, как два агента WebRTC обменялись SDP, у них есть достаточно информации для попытки подключения друг к другу. Для осуществления этого подключения WebRTC использует другую устоявшуюся технологию, называемую ICE (Interactive Connectivity Establishment).

ICE - это протокол, который существует до WebRTC и позволяет установить прямое соединение между двумя агентами без центрального сервера. Эти два агента могут находиться в одной сети или на противоположных концах земного шара.

ICE обеспечивает прямое подключение, но настоящая магия процесса подключения заключается в концепции “обхода NAT” и использовании STUN/TURN-серверов. Эти две концепции, которые мы подробнее исследуем позже, - все, что вам нужно для связи с ICE-агентом в другой подсети.

Когда два агента успешно установили ICE-соединение, WebRTC переходит к следующему шагу: установлению зашифрованного транспорта для обмена аудио, видео и данными между ними.

Защита транспортного уровня с помощью DTLS и SRTP

Теперь, когда у нас есть двунаправленная связь (через ICE), нам нужно сделать нашу коммуникацию безопасной! Это делается с помощью еще двух протоколов, которые также существовали до WebRTC: DTLS (Datagram Transport Layer Security) и SRTP (Secure Real-Time Transport Protocol). Первый протокол, DTLS, - это просто TLS поверх UDP (TLS - криптографический протокол, используемый для защиты связи через HTTPS). Второй протокол, SRTP, используется для обеспечения шифрования пакетов данных RTP (Real-time Transport Protocol).

Сначала WebRTC подключается, выполняя DTLS-рукопожатие поверх соединения, установленного ICE. В отличие от HTTPS, WebRTC не использует центральный орган для сертификатов. Он просто утверждает, что сертификат, обмененный через DTLS, соответствует отпечатку, предоставленному через

сигнализацию. Это DTLS-соединение затем используется для сообщений DataChannel.

Далее, WebRTC использует протокол RTP, защищенный с помощью SRTP, для передачи аудио/видео. Мы инициализируем нашу SRTP-сессию, извлекая ключи из согласованной DTLS-сессии.

Мы обсудим, почему передача медиа и данных имеют свои собственные протоколы, в следующей главе, но пока достаточно знать, что они обрабатываются отдельно.

Коммуникация с пирами через RTP и SCTP

Теперь, когда у нас есть два WebRTC-агента, подключенные и защищенные, с установленной двунаправленной связью, давайте начнем общаться! Снова WebRTC будет использовать два существующих протокола: RTP (Real-time Transport Protocol) и SCTP (Stream Control Transmission Protocol). Мы используем RTP для обмена медиа, зашифрованными с помощью SRTP, и SCTP для отправки и получения сообщений DataChannel, зашифрованных с помощью DTLS.

RTP - это довольно минимальный протокол, но он предоставляет необходимые инструменты для реализации потоковой передачи в реальном времени. Самое важное в RTP - это гибкость для разработчика, позволяющая обрабатывать задержку, потерю пакетов и перегрузки по своему усмотрению. Мы обсудим это подробнее в главе о медиа.

Последний протокол в стеке - SCTP. Важная особенность SCTP заключается в том, что вы можете отключить надежную и упорядоченную доставку сообщений (среди многих других опций). Это позволяет разработчикам обеспечить необходимую задержку для систем реального времени.

Теперь мы закончили! Мы успешно установили двунаправленную и безопасную связь. Если у вас есть стабильное соединение между вашими WebRTC-агентами, это вся сложность, которая вам нужна. В следующем разделе мы обсудим, как WebRTC справляется с реальными проблемами потери пакетов и ограничениями пропускной способности.

WebRTC - набор протоколов

WebRTC решает множество проблем. На первый взгляд технология может показаться чрезмерно сложной, но гениальность WebRTC заключается в его скромности. Он был создан не с

предположением, что сможет решить все лучше. Вместо этого он объединил многие существующие технологии с единственной целью в единый, широко применимый пакет.

Это позволяет нам изучать и исследовать каждую часть индивидуально, не испытывая перегрузки. Хороший способ визуализировать это - 'WebRTC-агент' на самом деле является просто оркестратором многих различных протоколов.

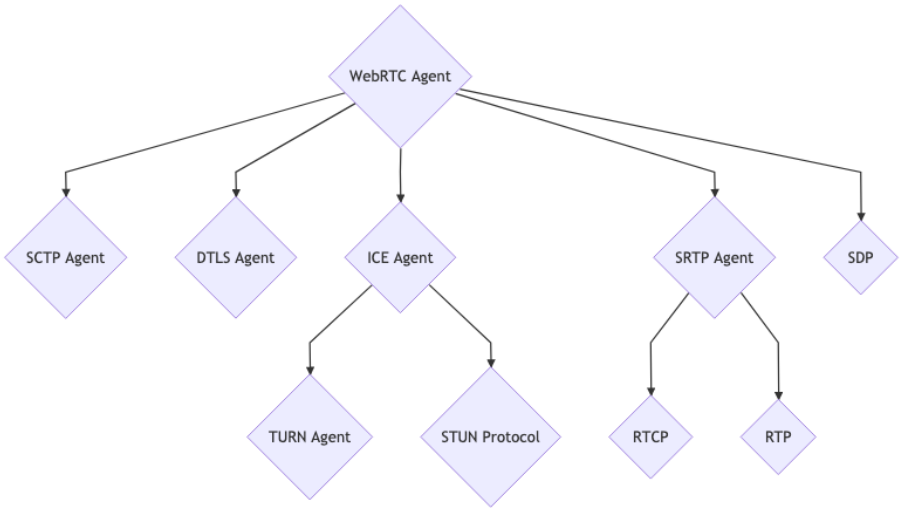


Figure 1: WebRTC Agent

Как работает API WebRTC?

Этот раздел описывает, как JavaScript API WebRTC соотносится с протоколом WebRTC, описанным выше. Он не предназначен для подробной демонстрации API WebRTC, а скорее для создания ментальной модели того, как все связано. Если вы не знакомы ни с протоколом, ни с API, не волнуйтесь. Этот раздел будет интересно перечитать, когда вы узнаете больше!

`new RTCPeerConnection`

`RTCPeerConnection` - это верхнеуровневый "сеанс WebRTC". Он содержит все упомянутые выше протоколы. Подсистемы все выделены, но пока ничего не происходит.

`addTrack`

`addTrack` создает новый RTP-поток. Для этого потока будет сгенерирован случайный Synchronization Source (SSRC). Этот поток будет находиться внутри Session Description, сгенерированной `createOffer` в медиа-секции. Каждый вызов `addTrack` создаст новый SSRC и медиа-секцию.

Сразу после установления SRTP-сессии эти медиа-пакеты начнут шифроваться с помощью SRTP и отправляться через ICE.

`createDataChannel`

`createDataChannel` создает новый SCTP-поток, если SCTP-ассоциация не существует. SCTP не включен по умолчанию. Он запускается только когда одна сторона запрашивает канал данных.

Сразу после установления DTLS-сессии SCTP-ассоциация начнет отправлять пакеты, зашифрованные с помощью DTLS через ICE.

`createOffer`

`createOffer` генерирует Session Description локального состояния для обмена с удаленным пиром.

Акт вызова `createOffer` не меняет ничего для локального пира.

`setLocalDescription`

`setLocalDescription` фиксирует все запрошенные изменения. Вызовы `addTrack`, `createDataChannel` и подобные являются временными до этого вызова. `setLocalDescription` вызывается со значением, сгенерированным `createOffer`.

Обычно после этого вызова вы отправите предложение удаленному пиру, который будет использовать его для вызова `setRemoteDescription`.

`setRemoteDescription`

`setRemoteDescription` - это способ информирования локального агента о состоянии удаленных кандидатов. Так акт ‘Сигнализации’ выполняется с помощью JavaScript API.

Когда `setRemoteDescription` был вызван на обеих сторонах, WebRTC-агенты теперь имеют достаточно информации для начала связи “Peer-To-Peer” (P2P)!

addIceCandidate

`addIceCandidate` позволяет WebRTC-агенту добавлять дополнительные удаленные ICE-кандидаты в любое время. Этот API отправляет ICE-кандидата прямо в подсистему ICE и не имеет другого эффекта для большего WebRTC-соединения.

ontrack

`ontrack` - это обратный вызов, срабатывающий при получении RTP-пакета от удаленного пира. Входящие пакеты были объявлены в Session Description, переданном через `setRemoteDescription`.

WebRTC использует SSRC и ищет связанный `MediaStream` и `MediaStreamTrack`, и вызывает этот обратный вызов с этими деталями.

oniceconnectionstatechange

`oniceconnectionstatechange` - это обратный вызов, который срабатывает при изменении состояния ICE-агента. Когда происходит изменение сетевой связности, вы получаете об этом уведомление.

onconnectionstatechange

`onconnectionstatechange` - это комбинация состояний ICE-агента и DTLS-агента. Вы можете наблюдать за этим, чтобы быть уведомленным, когда ICE и DTLS успешно завершены.

Сигнализация

Что такое сигнализация WebRTC?

Когда вы создаете агента WebRTC, он ничего не знает о другом пире. У него нет представления, с кем он будет подключаться или что будет отправляться! Сигнализация - это начальная загрузка, которая делает вызов возможным. После обмена этими значениями WebRTC-агенты могут напрямую общаться друг с другом.

Сигнальные сообщения - это просто текст. WebRTC-агенты не заботятся о том, как они передаются. Обычно они передаются через Websockets, но это не является обязательным требованием.

Как работает сигнализация WebRTC?

WebRTC использует существующий протокол, называемый Session Description Protocol. Через этот протокол два WebRTC-агента будут обмениваться всем состоянием, необходимым для установления соединения. Сам протокол прост для чтения и понимания. Сложность заключается в понимании всех значений, которые WebRTC заполняет.

Этот протокол не специфичен для WebRTC. Мы сначала изучим Session Description Protocol, даже не говоря о WebRTC. WebRTC использует только подмножество протокола, поэтому мы рассмотрим только то, что нам нужно. После того, как мы поймем протокол, мы перейдем к его практическому использованию в WebRTC.

Что такое *Session Description Protocol* (SDP)?

Session Description Protocol определен в RFC 8866. Это протокол “ключ/значение” с новой строкой после каждого значения. Он будет похож на INI-файл. Описание сессии содержит ноль или более описаний медиа. Мысленно вы можете представить его как описание сессии, содержащее массив описаний медиа.

Описание медиа обычно соответствует одному потоку медиа. Так что если бы вы хотели описать вызов с тремя видеопотоками и двумя аудиотреками, у вас было бы пять описаний медиа.

Как читать SDP

Каждая строка в описании сессии будет начинаться с одного символа - это ваш ключ. За ним следует знак равенства. Все после знака равенства - это значение. После завершения значения будет новая строка.

Session Description Protocol определяет все допустимые ключи. Вы можете использовать только буквы для ключей, как определено в протоколе. У этих ключей есть значимый смысл, который будет объяснен позже.

Возьмем этот фрагмент описания сессии:

```
a=my-sdp-value
a=second-value
```

У вас две строки. Каждая с ключом `a`. Первая строка имеет значение `my-sdp-value`, вторая строка имеет значение `second-value`.

WebRTC использует только некоторые ключи SDP

Не все значения ключей, определенные Session Description Protocol, используются WebRTC. Важны только ключи, используемые в JavaScript Session Establishment Protocol (JSEP), определенном в RFC 8829. Следующие семь ключей - единственные, которые вам нужно понять прямо сейчас:

- `v` - Версия, должна быть равна `0`.
- `o` - Происхождение, содержит уникальный ID, полезный для повторных переговоров.
- `s` - Имя сессии, должно быть равно `-`.
- `t` - Время, должно быть равно `0 0`.
- `m` - Описание медиа (`m=<media> <port> <proto> <fmt> ...`), описано подробнее ниже.
- `a` - Атрибут, текстовое поле. Это самая распространенная строка в WebRTC.
- `c` - Данные о подключении, должны быть равны `IN IP4 0.0.0.0`.

Описания медиа в описании сессии

Описание сессии может содержать неограниченное количество описаний медиа.

Определение описания медиа содержит список форматов. Эти форматы сопоставляются с RTP Payload Types. Сам кодек затем определяется атрибутом со значением `rtpmap` в описании медиа. Важность RTP и RTP Payload Types обсуждается позже в главе о медиа. Каждое описание медиа может содержать неограниченное количество атрибутов.

Возьмем этот фрагмент описания сессии в качестве примера:

```
v=0
m=audio 4000 RTP/AVP 111
a=rtpmap:111 OPUS/48000/2
m=video 4000 RTP/AVP 96
a=rtpmap:96 VP8/90000
a=my-sdp-value
```

У вас два описания медиа: одно типа audio с fmt `111` и одно типа video с форматом `96`. Первое описание медиа имеет только один атрибут. Этот атрибут сопоставляет Payload Type `111` с Opus. Второе описание медиа имеет два атрибута. Первый атрибут сопоставляет Payload Type `96` с VP8, а второй атрибут просто `my-sdp-value`.

Полный пример

Следующий пример объединяет все концепции, о которых мы говорили. Это все функции Session Description Protocol, которые использует WebRTC. Если вы можете прочитать это, вы можете прочитать любое WebRTC-описание сессии!

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=-
c=IN IP4 127.0.0.1
t=0 0
m=audio 4000 RTP/AVP 111
a=rtpmap:111 OPUS/48000/2
m=video 4002 RTP/AVP 96
a=rtpmap:96 VP8/90000
```

- `v`, `o`, `s`, `c`, `t` определены, но не влияют на сеанс WebRTC.
- У вас два описания медиа. Одно типа `audio` и одно типа `video`.
- У каждого есть один атрибут. Этот атрибут настраивает детали RTP-конвейера, который обсуждается в главе “Медиа-коммуникация”.

Как *Session Description Protocol* и WebRTC работают вместе

Следующий элемент головоломки - понимание *как* WebRTC использует Session Description Protocol.

Что такое Offers и Answers?

WebRTC использует модель предложения/ответа. Это означает, что один WebRTC-агент делает “Предложение” для начала вызова, и другие WebRTC-агенты “Отвечают”, если они готовы принять предложенное.

Это дает отвечающему возможность отклонить неподдерживаемые кодеки в описаниях медиа. Так два пира могут понять, какие форматы они готовы обмениваться.

Трансиверы для отправки и получения

Трансиверы - это концепция, специфичная для WebRTC, которую вы увидите в API. Она раскрывает “Описание медиа” в JavaScript API. Каждое описание медиа становится Трансивером. Каждый

раз, когда вы создаете Трансивер, новое описание медиа добавляется в локальное описание сессии.

Каждое описание медиа в WebRTC будет иметь атрибут направления. Это позволяет WebRTC-агенту заявить “Я собираюсь отправить вам этот кодек, но я не готов принимать ничего взамен”. Есть четыре допустимых значения:

- `send`
- `recv`
- `sendrecv`
- `inactive`

SDP-значения, используемые WebRTC

Это список некоторых распространенных атрибутов, которые вы увидите в описании сессии от WebRTC-агента. Многие из этих значений контролируют подсистемы, о которых мы еще не говорили.

`group:BUNDLE` Объединение - это акт передачи нескольких типов трафика по одному соединению. Некоторые реализации WebRTC используют выделенное соединение для каждого медиапотока. Объединение следует предпочитать.

`fingerprint:sha-256` Это хеш сертификата, который пир использует для DTLS. После завершения DTLS-рукопожатия вы сравниваете его с фактическим сертификатом, чтобы подтвердить, что общаетесь с кем ожидаете.

`setup:` Это контролирует поведение DTLS-агента. Определяет, будет ли он работать как клиент или сервер после подключения ICE. Возможные значения:

- `setup:active` - Работать как DTLS-клиент.
- `setup:passive` - Работать как DTLS-сервер.
- `setup:actpass` - Попросить другого WebRTC-агента выбрать.

`mid` Атрибут “mid” используется для идентификации медиапотоков в описании сессии.

`ice-ufrag` Это значение фрагмента пользователя для ICE-агента. Используется для аутентификации ICE-трафика.

ice-pwd Это пароль для ICE-агента. Используется для аутентификации ICE-трафика.

rtpmap Это значение используется для сопоставления определенного кодека с RTP Payload Type. Payload types не статичны, поэтому для каждого вызова отправляющий решает payload types для каждого кодека.

fmtp Определяет дополнительные значения для одного Payload Type. Это полезно для передачи определенного видеопрофиля или настроек кодировщика.

candidate Это ICE-кандидат, исходящий от ICE-агента. Это один из возможных адресов, на котором доступен WebRTC-агент. Они полностью объясняются в следующей главе.

ssrc Synchronization Source (SSRC) определяет один медиапоток.

`label` - это ID для этого отдельного потока. `mslabel` - это ID для контейнера, который может содержать несколько потоков внутри.

Пример описания сессии WebRTC

Следующее - полное описание сессии, сгенерированное WebRTC-клиентом:

```
v=0
o=- 3546004397921447048 1596742744 IN IP4 0.0.0.0
s=-
t=0 0
a=fingerprint:sha-256 0F:74:31:25:CB:A2:13:EC:28:6F:6D:2C:61:FF:5D:C2:BC:B9:DB:3D:98
a=group:BUNDLE 0 1
m=audio 9 UDP/TLS/RTP/SAVPF 111
c=IN IP4 0.0.0.0
a=setup:active
a=mid:0
a=ice-ufrag:CsxzEWmoKpJyscFj
a=ice-pwd:mktpbhgREmjEwUFSIJyPINPUhgDqJlSd
a=rtcp-mux
a=rtcp-rsize
a=rtpmap:111 opus/48000/2
a=fmtp:111 minptime=10;useinbandfec=1
a=ssrc:350842737 cname:yvKPspsHcYcwGFTw
a=ssrc:350842737 msid:yvKPspsHcYcwGFTw DfQnKjQQuwceLFdV
```

```
a=ssrc:350842737 mslabel:yvKPspsHcYcwGFTw
a=ssrc:350842737 label:DfQnKjQQuwceLFdV
a=msid:yvKPspsHcYcwGFTw DfQnKjQQuwceLFdV
a=sendrecv
a=candidate:foundation 1 udp 2130706431 192.168.1.1 53165 typ host generation 0
a=candidate:foundation 2 udp 2130706431 192.168.1.1 53165 typ host generation 0
a=candidate:foundation 1 udp 1694498815 1.2.3.4 57336 typ srflx raddr 0.0.0.0 rport 5733
a=candidate:foundation 2 udp 1694498815 1.2.3.4 57336 typ srflx raddr 0.0.0.0 rport 5733
a=end-of-candidates
m=video 9 UDP/TLS/RTP/SAVPF 96
c=IN IP4 0.0.0.0
a=setup:active
a=mid:1
a=ice-ufrag:CsxzEWmoKpJyscFj
a=ice-pwd:mktpbhgREmjEwUFSIJyPINPUhgDqJlSd
a=rtcp-mux
a=rtcp-rsize
a=rtpmap:96 VP8/90000
a=ssrc:2180035812 cname:XHbOTNRFnLtesHwJ
a=ssrc:2180035812 msid:XHbOTNRFnLtesHwJ JgtwEhBWNEiOnhuW
a=ssrc:2180035812 mslabel:XHbOTNRFnLtesHwJ
a=ssrc:2180035812 label:JgtwEhBWNEiOnhuW
a=msid:XHbOTNRFnLtesHwJ JgtwEhBWNEiOnhuW
a=sendrecv
```

Вот что мы знаем из этого сообщения:

- У нас два медиа-раздела: один аудио и один видео.
- Оба они `sendrecv` трансиверы. Мы получаем два потока и можем отправить два обратно.
- У нас есть ICE-кандидаты и детали аутентификации, поэтому мы можем попытаться подключиться.
- У нас есть отпечаток сертификата, поэтому мы можем провести безопасный вызов.

Дополнительные темы

В более поздних версиях этой книги также будут рассмотрены следующие темы:

- Повторные переговоры
- Симулкаст

Подключение

Почему WebRTC нужна выделенная подсистема для подключения?

Большинство приложений, развернутых сегодня, устанавливают клиент-серверные соединения. Клиент-серверное соединение требует, чтобы сервер имел стабильный, известный транспортный адрес. Клиент связывается с сервером, и сервер отвечает.

WebRTC не использует клиент-серверную модель, а устанавливает одноранговые (P2P) соединения. В P2P-соединении задача создания соединения равномерно распределена между обоими пирами. Это связано с тем, что транспортный адрес (IP и порт) в WebRTC не может быть предположен и может даже измениться во время сеанса. WebRTC соберет всю доступную информацию и приложит все усилия для достижения двунаправленной связи между двумя WebRTC-агентами.

Установление одноранговой связности может быть сложным. Эти агенты могут находиться в разных сетях без прямой связности. В ситуациях, где прямая связность существует, все еще могут возникать другие проблемы. В некоторых случаях ваши клиенты не говорят на одних сетевых протоколах (UDP <-> TCP) или используют разные версии IP (IPv4 <-> IPv6).

Несмотря на эти сложности при настройке P2P-соединения, вы получаете преимущества перед традиционной клиент-серверной технологией благодаря следующим атрибутам, которые предлагает WebRTC.

Уменьшенные затраты на пропускную способность

Поскольку медиакоммуникация происходит непосредственно между пирами, вам не нужно платить за, или размещать отдельный сервер для передачи медиа.

Нижняя задержка

Связь быстрее, когда она прямая! Когда пользователь должен запускать все через ваш сервер, это замедляет передачи.

Безопасная E2E-связь

Прямая связь безопаснее. Поскольку пользователи не маршрутизируют данные через ваш сервер, им даже не нужно доверять вам, что вы не расшифруете их.

Как это работает?

Процесс, описанный выше, называется Установление интерактивной связности (ICE). Другой протокол, который предшествует WebRTC.

ICE - это протокол, который пытается найти лучший способ связи между двумя ICE-агентами. Каждый ICE-агент публикует способы, которыми он достижим, это называется кандидатами. Кандидат - это по сути транспортный адрес агента, который он считает достижимым для другого пира. ICE затем определяет лучшую пару кандидатов.

Фактический процесс ICE описывается более подробно позже в этой главе. Для понимания того, почему существует ICE, полезно понять, какие сетевые поведения мы преодолеваем.

Сетевые реальные ограничения

ICE - это все о преодолении ограничений реальных сетей. Прежде чем мы исследуем решение, давайте поговорим о реальных проблемах.

Не в одной сети

Большую часть времени другой WebRTC-агент даже не будет в той же сети. Обычный звонок обычно происходит между двумя WebRTC-агентами в разных сетях с отсутствием прямой связности.

Ниже приведен график двух разных сетей, соединенных через публичный интернет. В каждой сети у вас два хоста.

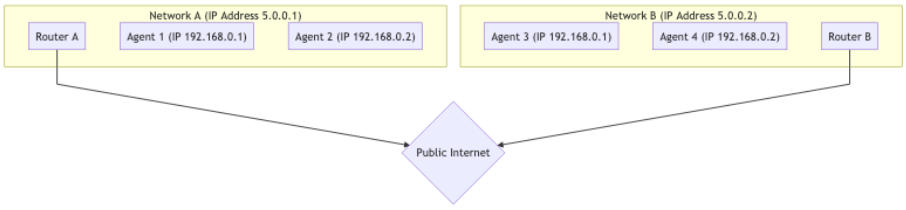


Figure 2: Две сети

Для хостов в одной сети соединение очень просто. Связь между `192.168.0.1 -> 192.168.0.2` легко сделать! Эти два хоста могут подключиться друг к другу без какой-либо внешней помощи.

Однако хост, использующий `Router B`, не имеет способа напрямую получить доступ к чему-либо за `Router A`. Как вы можете сказать разницу между `192.168.0.1` за `Router A` и тем же IP за `Router B`? Это частные IP! Хост, использующий `Router B`, мог бы отправить трафик непосредственно к `Router A`, но запрос завершится там. Как `Router A` знает, какой хост он должен передать сообщение?

Протокольные ограничения

Некоторые сети вообще не позволяют трафик UDP, или, возможно, они не позволяют TCP. Некоторые сети могут иметь очень низкий MTU (Maximum Transmission Unit). Существует множество переменных, которые сетевые администраторы могут изменять, что может сделать связь сложной.

Правила брандмауэра/IDS

Другой - “Глубокая проверка пакетов” и другие интеллектуальные фильтры. Некоторые сетевые администраторы будут запускать программное обеспечение, которое пытается обрабатывать каждый пакет. Часто это программное обеспечение не понимает WebRTC, поэтому оно блокирует его, потому что оно не знает, что делать, например, обрабатывая пакеты WebRTC как подозрительные UDP-пакеты на произвольном порту, который не белый.

NAT-маппинг

NAT (Network Address Translation) маппинг - это волшебство, которое делает возможным подключение WebRTC. Это как WebRTC позволяет двум пирам в совершенно разных подсетях общаться, решая проблему “не в одной сети” выше. Хотя это создает новые вызовы, давайте объясним, как работает NAT-маппинг сначала.

Он не использует ретрансляцию, прокси или сервер. Снова у нас есть `Agent 1` и `Agent 2`, и они находятся в разных сетях. Однако трафик течет полностью через. Визуализируется это так:

Чтобы сделать эту связь, вы устанавливаете NAT-маппинг. Agent 1 использует порт 7000 для установки WebRTC-соединения с Agent 2. Это создает привязку `192.168.0.1:7000` к `5.0.0.1:7000`.

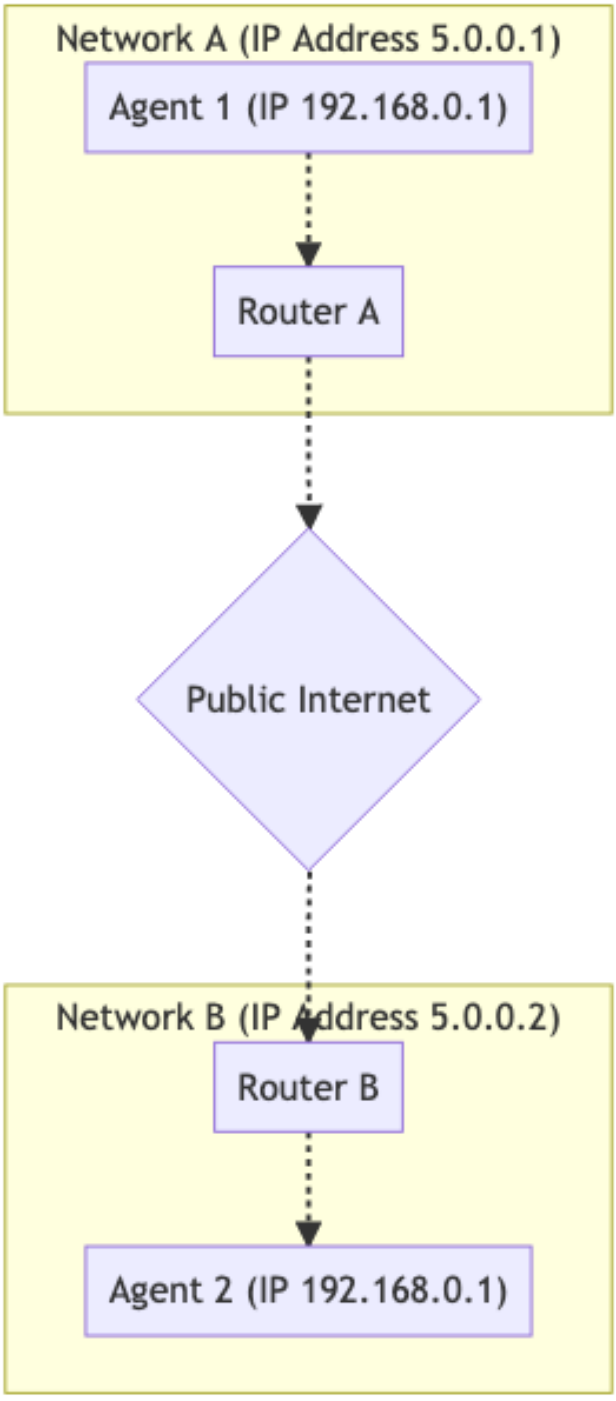


Figure 3: NAT-маппинг

Это затем позволяет Agent 2 достичь Agent 1, отправляя пакеты на `5.0.0.1:7000`. Создание NAT-маппинга, как в этом примере, похоже на автоматизированную версию настройки портового перенаправления в вашем роутере.

Недостатком NAT-маппинга является то, что нет единого вида маппинга (например, статическое портовое перенаправление), и поведение несогласовано между сетями. ISPs и производители оборудования могут делать это по-разному. В некоторых случаях сетевые администраторы могут даже отключить его.

Хорошая новость в том, что полный диапазон поведений понятен и наблюдаем, поэтому ICE-агент способен подтвердить, что он создал NAT-маппинг, и атрибуты маппинга.

Документ, описывающий эти поведения, - RFC 4787.

Создание маппинга

Создание маппинга - это самая простая часть. Когда вы отправляете пакет по адресу вне вашей сети, создается маппинг! NAT-маппинг - это временный публичный IP и порт, который выделяется вашим NAT. Исходящее сообщение будет переписано, чтобы иметь свой источник, заданный новым адресом маппинга. Если сообщение отправлено на маппинг, оно будет автоматически маршрутизироваться обратно к хосту внутри NAT, который его создал. Детали вокруг маппинга - это то, где все усложняется.

Поведение создания маппинга

Создание маппинга делится на три разных категории:

Конечная точка, независимая от маппинга Создается один маппинг для каждого отправителя внутри NAT. Если вы отправите два пакета на два разных удаленных адреса, маппинг будет переиспользован. Оба удаленных хоста увидят один и тот же IP-адрес и порт. Если удаленные хосты отвечают, они будут отправлены обратно к той же локальной слушательной точке.

Это лучший сценарий. Для работы звонка, по крайней мере, одна сторона МОЖЕТ быть этого типа.

Адрес, зависимый от маппинга Создается новый маппинг каждый раз, когда вы отправляете пакет по новому адресу. Если вы отправите два пакета на разные хосты, создадутся два маппинга. Если вы отправите два пакета на один удаленный

хост, но разные порты назначения, новый маппинг НЕ будет создан.

Адрес и порт, зависимый от маппинга Создается новый маппинг, если удаленный IP или порт отличается. Если вы отправите два пакета на один удаленный хост, но разные порты назначения, создается новый маппинг.

Поведение фильтрации маппинга

Поведение фильтрации - это правила, которые определяют, кто может использовать маппинг. Они делятся на три похожие категории:

Конечная точка, независимая от фильтрации Кто угодно может использовать маппинг. Вы можете поделиться маппингом с несколькими другими пирами, и они могут все отправить трафик к нему.

Адрес, зависимый от фильтрации Только хост, для которого создан маппинг, может использовать маппинг. Если вы отправите пакет на хост `A`, вы можете получить ответ только от этого же хоста. Если хост `B` попытается отправить пакет на этот маппинг, он будет проигнорирован.

Адрес и порт, зависимый от фильтрации Только хост и порт, для которых создан маппинг, могут использовать этот маппинг. Если вы отправите пакет на `A:5000`, вы можете получить ответ только от этого же хоста и порта. Если `A:5001` попытается отправить пакет на этот маппинг, он будет проигнорирован.

Обновление маппинга

Рекомендуется, чтобы если маппинг не использовался в течение 5 минут, он должен быть уничтожен. Это полностью зависит от ISP или производителя оборудования.

STUN

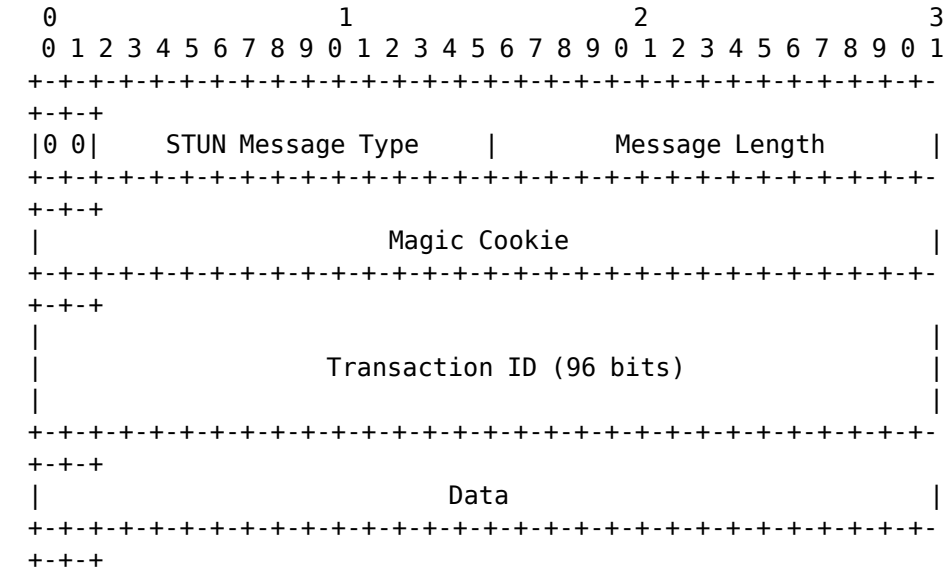
STUN (Session Traversal Utilities for NAT) - это протокол, созданный специально для работы с NAT. Это еще одна технология, которая предшествует WebRTC (и ICE!). Он определен в RFC 8489, который также определяет структуру STUN-пакета. Протокол STUN также используется ICE/TURN.

STUN полезен, потому что он позволяет программному созданию NAT-маппингов. До STUN мы могли создать NAT-маппинг, но у нас не было ни малейшего представления о том, какой у него IP-адрес и порт! STUN не только дает вам возможность создать маппинг, но и предоставляет детали, чтобы вы могли поделиться ими с другими, так что они могут отправить трафик обратно к вам через маппинг, который вы только что создали.

Давайте начнем с базового описания STUN. Позже мы расширим TURN и использование ICE. На данный момент мы просто опишем поток запрос/ответ для создания маппинга. Затем мы поговорим о том, как получить детали, чтобы поделиться ими. Это процесс, который происходит, когда у вас есть `stun:` сервер в ваших ICE-URL для WebRTC PeerConnection. В сущности, STUN помогает конечной точке за NAT определить, какой маппинг был создан, запрашивая STUN-сервер за пределами NAT, чтобы сообщить, что он наблюдает.

Структура пакета

Каждый STUN-пакет имеет следующую структуру:



Тип STUN-сообщения Каждый STUN-пакет имеет тип. На данный момент нам важно следующее:

- Запрос привязки - `0x0001`
- Ответ привязки - `0x0101`

Чтобы создать NAT-маппинг, мы делаем `Запрос привязки`. Затем сервер отвечает с `Ответ привязки`.

Длина сообщения Это длина секции `Данные`. Эта секция содержит произвольные данные, определяемые `Тип сообщения`.

Magic Cookie Фиксированное значение `0x2112A442` в сетевом порядке байтов, оно помогает отличить трафик STUN от других протоколов.

Transaction ID 96-битный идентификатор, который уникально идентифицирует запрос/ответ. Это помогает вам сопоставить ваши запросы и ответы.

Данные Данные будут содержать список STUN-атрибутов. STUN-атрибут имеет следующую структуру:

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+
|         Type                  |           Length              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+
|                         Value (variable)                ....
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+
```

`STUN Запрос привязки` не использует атрибуты. Это означает, что `STUN Запрос привязки` содержит только заголовок.

`STUN Ответ привязки` использует атрибут `XOR-MAPPED-ADDRESS (0x0020)`. Этот атрибут содержит IP и порт. Это IP и порт NAT-маппинга, который создается!

Создание NAT-маппинга

Создание NAT-маппинга с использованием STUN - это просто отправка одного запроса! Вы отправляете `STUN Запрос привязки` на STUN-сервер. STUN-сервер затем отвечает с `STUN Ответ привязки`. Этот `STUN Ответ привязки` будет содержать `Mapped Address`. `Mapped Address` - это как STUN-сервер видит вас и является вашим `NAT-маппингом`. `Mapped Address` - это то, что вы бы поделились, если хотели, чтобы кто-то отправил пакеты к вам.

Люди также называют Mapped Address вашим Общественным IP или Серверный рефлексивный кандидат.

Определение типа NAT

К сожалению, Mapped Address может быть бесполезен во всех случаях. Если это Адрес, зависимый, только STUN-сервер может отправить трафик обратно к вам. Если вы поделились им и другой пир попытался отправить сообщения, они будут отброшены. Это делает его бесполезным для общения с другими. Возможно, вы обнаружите, что случай Адрес, зависимый на самом деле решаем, если хост, который запускает STUN-сервер, также может передавать пакеты для вас к пиру! Это приводит нас к решению, использующему TURN ниже.

RFC 5780 определяет метод для запуска теста, чтобы определить ваш тип NAT. Это полезно, потому что вы бы знали заранее, возможно ли прямое подключение.

TURN

TURN (Traversal Using Relays around NAT) определен в RFC 8656 является решением, когда прямое подключение невозможно. Это может быть из-за того, что у вас два несовместимых типа NAT, или, возможно, они не могут говорить на одном и том же протоколе! TURN также может использоваться для целей конфиденциальности. Запуская все ваше общение через TURN, вы скрываете фактический адрес клиента.

TURN использует отдельный сервер. Этот сервер действует как прокси для клиента. Клиент подключается к TURN-серверу и создает Allocation. Создание Allocation позволяет клиенту получить временный IP/Port/Протокол, который может использоваться для отправки трафика обратно к клиенту. Этот новый слушатель известен как Relayed Transport Address. Думайте об этом как о пересылаемом адресе, вы его раздаете, чтобы другие могли отправить вам трафик через TURN! Для каждого пира вы даете Relay Transport Address, вы должны создать новую Permission, чтобы разрешить общение с вами.

Когда вы отправляете исходящий трафик через TURN, он отправляется через Relayed Transport Address. Когда удаленный пир получает трафик, он видит, что он приходит от TURN-сервера.

Жизненный цикл TURN

Следующее - это все, что должен сделать клиент, который хочет создать TURN-allocation. Общение с кем-то, кто использует TURN, не требует изменений. Другой пир получает IP и порт, и они общаются с ним, как с любым другим хостом.

Allocation Allocation - это ядро TURN. `Allocation` - это по сути “TURN-сессия”. Чтобы создать TURN-allocation, вы общаетесь с TURN `Server Transport Address` (обычно порт `3478`).

При создании `Allocation` вам нужно предоставить следующее: * Имя пользователя/Пароль - Создание TURN-allocation требует аутентификации. * Transport Allocation - Транспортный протокол между сервером (`Relayed Transport Address`) и пирами, может быть UDP или TCP. * Even-Port - Вы можете запросить последовательные порты для нескольких allocation, не относящихся к WebRTC.

Если запрос выполнен, вы получаете ответ от TURN-сервера со следующими STUN-атрибутами в секции `Данные`: * `XOR-MAPPED-ADDRESS - Mapped Address TURN Client`. Когда кто-то отправляет данные на `Relayed Transport Address`, это куда они перенаправляются. * `RELAYED-ADDRESS` - Это адрес, который вы выдаете другим клиентам. Если кто-то отправляет пакет на этот адрес, он передается TURN-клиенту. * `LIFETIME` - Через сколько времени эта TURN-allocation уничтожается. Вы можете продлить срок действия, отправив запрос `Refresh`.

Permissions Удаленный хост не может отправить в ваш `Relayed Transport Address`, пока вы не создадите для него разрешение. Когда вы создаете разрешение, вы говорите TURN-серверу, что этот IP и порт разрешен для отправки входящего трафика.

Удаленный хост должен отправить `STUN Binding Request` на TURN-сервер. Обычная ошибка - это то, что удаленный хост отправит `STUN Binding Request` на другой сервер. Они затем попросят вас создать разрешение для этого IP.

Предположим, вы хотите создать разрешение для хоста за `Адрес, зависимым маппингом`. Если вы сгенерируете `Mapped Address` от другого TURN-сервера, все входящие трафик будет отброшен. Каждый раз, когда они общаются с разным хостом, генерируется новый маппинг. Разрешения истекают через 5 минут, если они не обновляются.

SendIndication/ChannelData Эти два сообщения предназначены для TURN-клиента, чтобы отправить сообщения удаленному пиру.

SendIndication - это самодостаточное сообщение. В нем находится данные, которые вы хотите отправить, и кто вы хотите отправить их. Это бесполезно, если вы отправляете много сообщений удаленному пиру. Если вы отправите 1,000 сообщений, вы повторите их IP-адрес 1,000 раз!

ChannelData позволяет вам отправлять данные, но не повторять IP-адрес. Вы создаете Channel с IP и портом. Затем вы отправляете с ChannelId, и IP и порт будут заполнены с серверной стороны. Это лучший выбор, если вы отправляете много сообщений.

Обновление Allocation уничтожаются автоматически. TURN-клиент должен обновлять их раньше, чем `LIFETIME`, указанный при создании allocation.

Использование TURN

Использование TURN существует в двух формах. Обычно один пир действует как “TURN-клиент”, а другой - как прямое общение. В некоторых случаях у вас может быть использование TURN с обеих сторон, например, потому что оба клиента находятся в сетях, которые блокируют UDP, и, следовательно, подключение к соответствующим TURN-серверам происходит через TCP.

Эти диаграммы помогают проиллюстрировать, как это может выглядеть.

Одна TURN-allocation для общения

Две TURN-allocations для общения

ICE

ICE (Interactive Connectivity Establishment) - это как WebRTC подключает два агента. Определен в RFC 8445, это еще одна технология, которая предшествует WebRTC! ICE - это протокол для установления связи. Он определяет все возможные маршруты между двумя пирами и затем обеспечивает, что вы остаетесь подключенными.

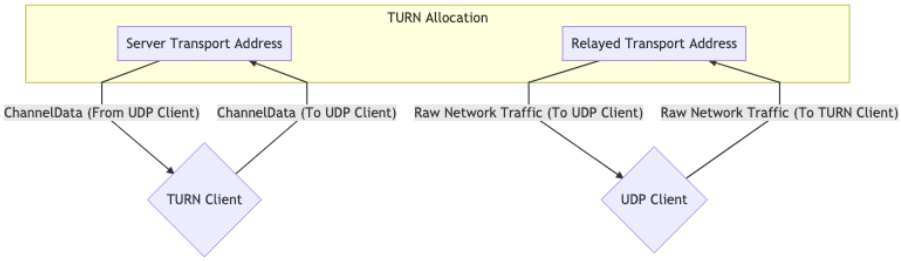


Figure 4: Одна TURN-allocation

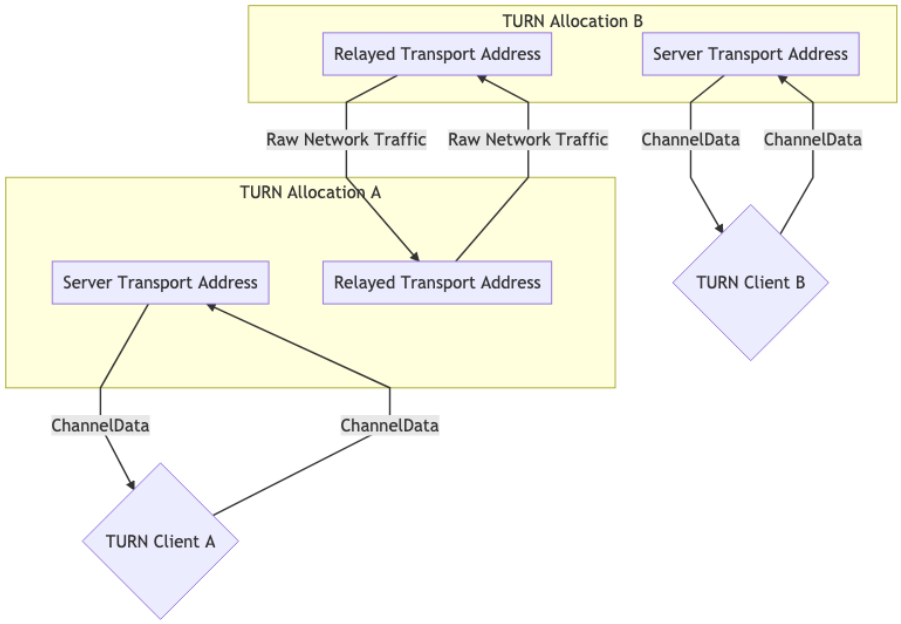


Figure 5: Две TURN-allocations

Эти маршруты известны как Candidate Pairs, которые являются парой локального и удаленного транспортного адреса. Это где STUN и TURN вступают в игру с ICE. Эти адреса могут быть ваш локальный IP-адрес плюс порт, NAT-маппинг, или Relayed Transport Address. Каждая сторона собирает все адреса, которые хотят использовать, обмениваются ими и затем пытаются подключиться!

Два ICE-агента общаются с помощью пакетов ICE ping (или формально называемых проверками связности) для установления связи. После установления связи они могут отправлять все, что хотят. Это будет как использование обычного сокета. Эти проверки используют протокол STUN.

Создание ICE-агента

ICE-агент либо Controlling или Controlled. Controlling Agent - это тот, кто решает выбранную Candidate Pair. Обычно пир, отправляющий офер, является контролирующей стороной.

Каждая сторона должна иметь user fragment и password. Эти два значения должны быть обменены перед началом проверки связности. User fragment отправляется в открытом виде и полезен для демультиплексирования нескольких ICE-сессий. Password используется для генерации атрибута MESSAGE-INTEGRITY. В конце каждого STUN-пакета есть атрибут, который является хэшем всего пакета с использованием Password в качестве ключа. Это используется для аутентификации пакета и обеспечения того, что он не был изменен.

Для WebRTC все эти значения распределяются через Session Description, как описано в предыдущей главе.

Сбор кандидатов

Теперь нам нужно собрать все возможные адреса, которые мы достижимы. Эти адреса известны как кандидаты.

Хост Хост-кандидат слушает непосредственно на локальном интерфейсе. Это может быть UDP или TCP.

mDNS mDNS-кандидат похож на хост-кандидат, но IP-адрес скрыт. Вместо того, чтобы сообщать другой стороне ваш IP-адрес, вы даете им UUID как имя хоста. Затем вы устанавливаете многоадресный слушатель и отвечаете, если кто-то запросит UUID, который вы опубликовали.

Если вы находитесь в той же сети, что и агент, вы можете найти друг друга через Multicast. Если вы не находитесь в той же сети, вы не сможете подключиться (если сетевой администратор явно настроил сеть, чтобы разрешить пакеты Multicast для прохождения).

Это полезно для целей конфиденциальности. Пользователь мог бы узнать ваш локальный IP-адрес через WebRTC с хост-кандидатом (без даже пытаясь подключиться к вам), но с mDNS-кандидатом, теперь они получают случайный UUID.

Серверный рефлексивный Серверный рефлексивный кандидат генерируется путем выполнения `STUN Binding Request` к STUN-серверу.

Когда вы получаете `STUN Binding Response`, `XOR-MAPPED-ADDRESS` является вашим серверным рефлексивным кандидатом.

Пир-рефлексивный Пир-рефлексивный кандидат создается, когда удаленный пир получает ваш запрос от адреса, который ранее был неизвестен для пира. При получении он сообщает (отражает) этот адрес обратно к вам. Пир знает, что запрос был отправлен вами, а не кем-то другим, потому что ICE - это аутентифицированный протокол.

Это обычно происходит, когда `Хост-кандидат` общается с `Серверный рефлексивный кандидат`, который находится в другой подсети, что приводит к созданию нового `NAT-маппинга`. Помните, мы сказали, что проверки связности на самом деле STUN-пакеты? Формат ответа STUN естественным образом позволяет пиру сообщить обратно пир-рефлексивный адрес.

Relay Relay-кандидат генерируется путем использования TURN-сервера.

После начального рукопожатия с TURN-сервером вам дается `RELAYED-ADDRESS`, это ваш Relay-кандидат.

Проверка связности

Теперь мы знаем `user fragment`, `password` удаленного агента и кандидаты. Мы можем теперь попытаться подключиться! Каждый кандидат связывается с каждым другим. Так что если у вас 3 кандидата с каждой стороны, теперь у вас 9 пар кандидатов.

Визуально это выглядит так:

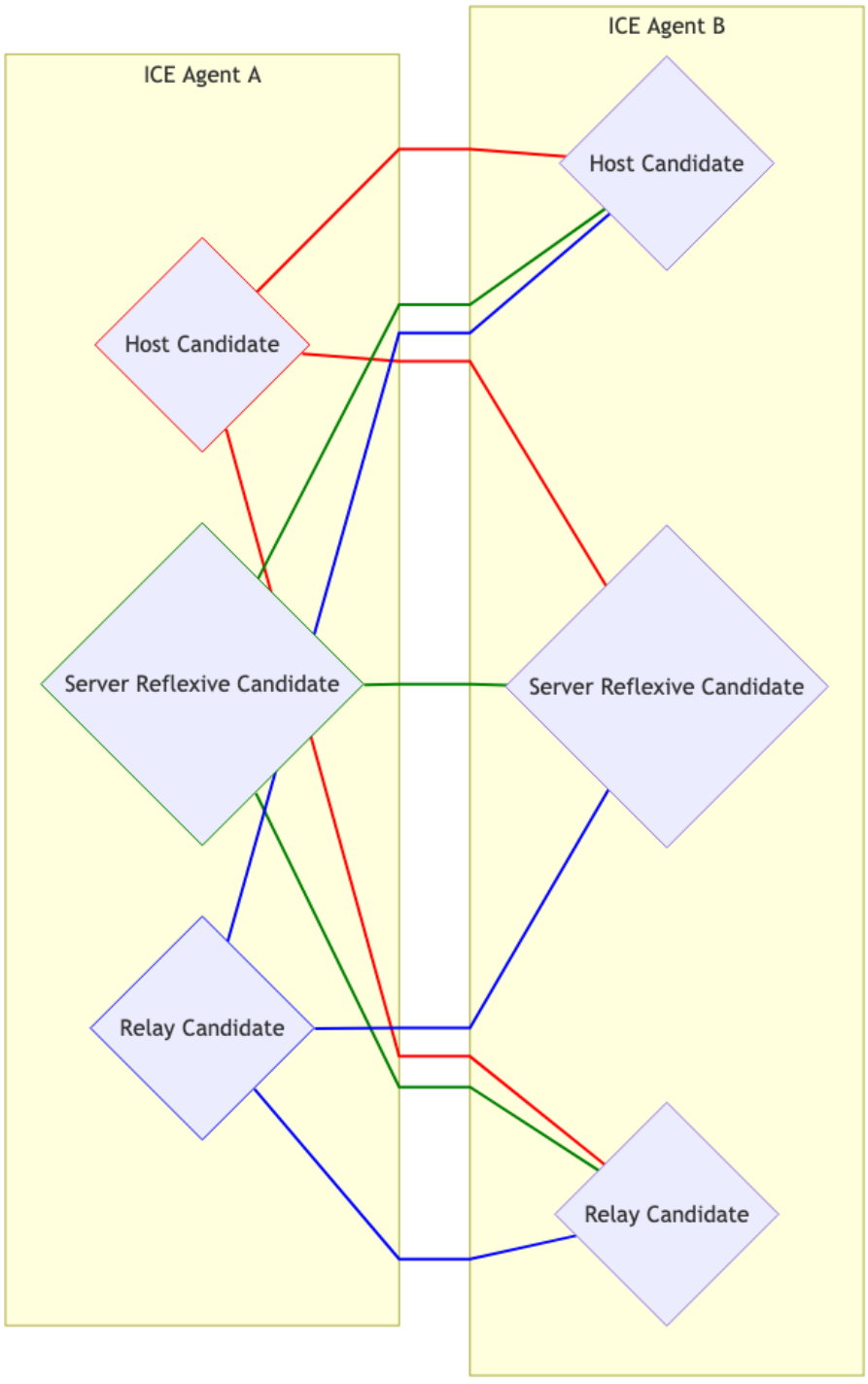


Figure 6: Проверка связности

Выбор кандидата

Контролирующий и контролируемый агенты начинают отправлять трафик на каждой паре. Это необходимо, если один агент находится за Адрес, зависимым маппингом, это приведет к созданию Пир-рефлексивного кандидата.

Каждая Candidate Pair, которая видела сетевой трафик, затем повышается до Valid Candidate пары. Контролирующий агент затем выбирает одну Valid Candidate пару и номинирует ее. Это становится Nominated Pair. Контролирующий и контролируемый агенты затем пытаются установить один последний раунд двунаправленной связи. Если это удается, Nominated Pair становится Selected Candidate Pair! Эта пара используется для остальной части сессии.

Перезапуски

Если Selected Candidate Pair перестает работать по какой-либо причине (NAT-маппинг истек, TURN-сервер упал) ICE-агент переходит в состояние Failed. Оба агента могут быть перезапущены, и процесс будет повторен полностью.

Защита

Какую безопасность имеет WebRTC?

Каждое WebRTC-соединение аутентифицировано и зашифровано. Вы можете быть уверены, что третья сторона не может видеть, что вы отправляете, или вставлять поддельные сообщения. Вы также можете быть уверены, что WebRTC-агент, который сгенерировал Session Description, - тот, с кем вы общаетесь.

Очень важно, чтобы никто не изменял эти сообщения. Нормально, если третья сторона читает Session Description в транзите. Однако WebRTC не защищает от его изменения. Злоумышленник может выполнить атаку “человек посередине”, изменив ICE-кандидаты и обновив отпечаток сертификата.

Как это работает?

WebRTC использует два существующих протокола: Datagram Transport Layer Security (DTLS) и Secure Real-time Transport Protocol (SRTP).

DTLS позволяет согласовать сеанс и затем безопасно обмениваться данными между двумя пирами. Это родственник TLS, той же

технологии, которая работает в HTTPS, но DTLS использует UDP вместо TCP в качестве транспортного уровня. Это означает, что протокол должен справляться с ненадежной доставкой. SRTP специально разработан для безопасного обмена медиа. Есть некоторые оптимизации, которые мы можем сделать, используя его вместо DTLS.

Сначала используется DTLS. Он выполняет рукопожатие поверх соединения, предоставленного ICE. DTLS - это клиент/серверный протокол, поэтому одна сторона должна начать рукопожатие. Роли клиента/сервера выбираются во время сигнализации. Во время DTLS-рукопожатия обе стороны предлагают сертификат. После завершения рукопожатия этот сертификат сравнивается с хешем сертификата в Session Description. Это делается для обеспечения того, что рукопожатие произошло с ожидаемым WebRTC-агентом. DTLS-соединение затем становится доступным для использования в коммуникации DataChannel.

Чтобы создать SRTP-сессию, мы инициализируем ее, используя ключи, сгенерированные DTLS. У SRTP нет механизма рукопожатия, поэтому он должен быть загружен с внешними ключами. После этого медиа могут обмениваться, будучи зашифрованными с помощью SRTP!

Безопасность 101

Чтобы понять технологию, представленную в этой главе, вам сначала нужно понять эти термины. Криптография - сложный предмет, поэтому было бы полезно проконсультироваться и с другими источниками!

Открытый текст и шифротекст

Открытый текст - это вход для шифра. Шифротекст - это выход шифра.

Шифр

Шифр - это серия шагов, которая преобразует открытый текст в шифротекст. Шифр затем может быть обращен, так что вы можете преобразовать свой шифротекст обратно в открытый текст. У шифра обычно есть ключ для изменения его поведения. Другой термин для этого - шифрование и дешифрование.

Простой шифр - ROT13. Каждая буква перемещается на 13 символов вперед. Чтобы отменить шифр, вы перемещаете 13

символов назад. Открытый текст `HELLO` станет шифротекстом `URYYB`. В этом случае шифр - ROT, а ключ - 13.

Хеш-функции

Криптографическая хеш-функция - это необратимый процесс, который генерирует дайджест. При заданном входе она генерирует один и тот же выход каждый раз. Важно, чтобы выход *не* был обратимым. Имея выход, вы не должны иметь возможность определить его вход. Хеширование полезно, когда вы хотите убедиться, что сообщение не было изменено.

Простая (хотя и явно не подходящая для реальной криптографии) хеш-функция будет заключаться в том, чтобы брать только каждую вторую букву. `HELLO` станет `HLO`. Вы не можете предположить, что `HELLO` был входом, но можете подтвердить, что `HELLO` будет соответствовать хеш-дайджесту.

Криптография с открытым/закрытым ключом

Криптография с открытым/закрытым ключом описывает тип шифров, которые использует DTLS и SRTP. В этой системе у вас есть два ключа: открытый и закрытый. Открытый ключ предназначен для шифрования сообщений и безопасен для обмена. Закрытый ключ предназначен для дешифрования и никогда не должен быть раскрыт. Это единственный ключ, который может дешифровать сообщения, зашифрованные открытым ключом.

Обмен Диффи-Хеллмана

Обмен Диффи-Хеллмана позволяет двум пользователям, которые никогда не встречались, безопасно создать общий секрет через интернет. Пользователь `A` может отправить секрет пользователю `B` без беспокойства о перехвате. Это зависит от сложности решения проблемы дискретного логарифма. Вам не нужно полностью понимать, как это работает, но полезно знать, что именно это делает DTLS-рукопожатие возможным.

Wikipedia имеет пример этого в действии здесь.

Псевдослучайная функция

Псевдослучайная функция (PRF) - это предопределенная функция для генерации значения, которое выглядит случайным. Она может принимать несколько входов и генерировать один выход.

Функция генерации ключа

Генерация ключа - это тип псевдослучайной функции. Генерация ключа - это функция, используемая для усиления ключа. Один распространенный шаблон - растяжение ключа.

Допустим, вам дан ключ размером 8 байт. Вы могли бы использовать KDF, чтобы сделать его сильнее.

Nonce

Nonce - это дополнительный вход в шифр. Это используется для того, чтобы получить разный выход от шифра, даже если вы шифруете одно и то же сообщение несколько раз.

Если вы зашифруете одно и то же сообщение 10 раз, шифр даст вам один и тот же шифротекст 10 раз. Используя nonce, вы можете получить разный выход, все еще используя тот же ключ. Важно использовать разный nonce для каждого сообщения! В противном случае это сводит на нет большую часть ценности.

Код аутентификации сообщения

Код аутентификации сообщения - это хеш, который помещается в конец сообщения. MAC доказывает, что сообщение исходит от ожидаемого пользователя.

Если вы не используете MAC, злоумышленник может вставлять недопустимые сообщения. После дешифрования у вас будет просто мусор, потому что они не знают ключ.

Ротация ключа

Ротация ключа - это практика смены ключа через интервал. Это делает украденный ключ менее значимым. Если ключ украден или утек, можно расшифровать меньше данных.

DTLS

DTLS (Datagram Transport Layer Security) позволяет двум пирам устанавливать безопасную связь без предварительной конфигурации. Даже если кто-то подслушивает разговор, он не сможет расшифровать сообщения.

Для связи DTLS-клиента и сервера им нужно согласовать шифр и ключ. Они определяют эти значения, выполняя DTLS-рукопожатие. Во время рукопожатия сообщения находятся в открытом тексте. Когда DTLS-клиент/сервер обменялся

достаточным количеством деталей для начала шифрования, он отправляет `Change Cipher Spec`. После этого сообщения каждое последующее сообщение будет зашифровано!

Формат пакета

Каждый DTLS-пакет начинается с заголовка.

Тип содержимого Вы можете ожидать следующие типы:

- `20` - Change Cipher Spec
- `22` - Рукопожатие
- `23` - Прикладные данные

`Рукопожатие` используется для обмена деталями для запуска сеанса. `Change Cipher Spec` используется для уведомления другой стороны, что все будет зашифровано. `Прикладные данные` - это зашифрованные сообщения.

Версия Версия может быть `0x0000feff` (DTLS v1.0) или `0x0000fefd` (DTLS v1.2), версии v1.1 не существует.

Эпоха Эпоха начинается с `0`, но становится `1` после `Change Cipher Spec`. Любое сообщение с ненулевой эпохой зашифровано.

Порядковый номер Порядковый номер используется для сохранения сообщений в порядке. Каждое сообщение увеличивает порядковый номер. Когда эпоха увеличивается, порядковый номер начинается заново.

Длина и полезная нагрузка Полезная нагрузка зависит от `Типа содержимого`. Для `Прикладных данных` `Полезная нагрузка` - это зашифрованные данные. Для `Рукопожатия` она будет разной в зависимости от сообщения. Длина указывает, насколько большой `Полезная нагрузка`.

Конечный автомат рукопожатия

Во время рукопожатия клиент/сервер обмениваются серией сообщений. Эти сообщения сгруппированы в полеты. Каждый полет может содержать несколько сообщений (или только одно). Полет не считается завершенным, пока не будут получены все сообщения в полете. Мы подробнее опишем цель каждого сообщения ниже.

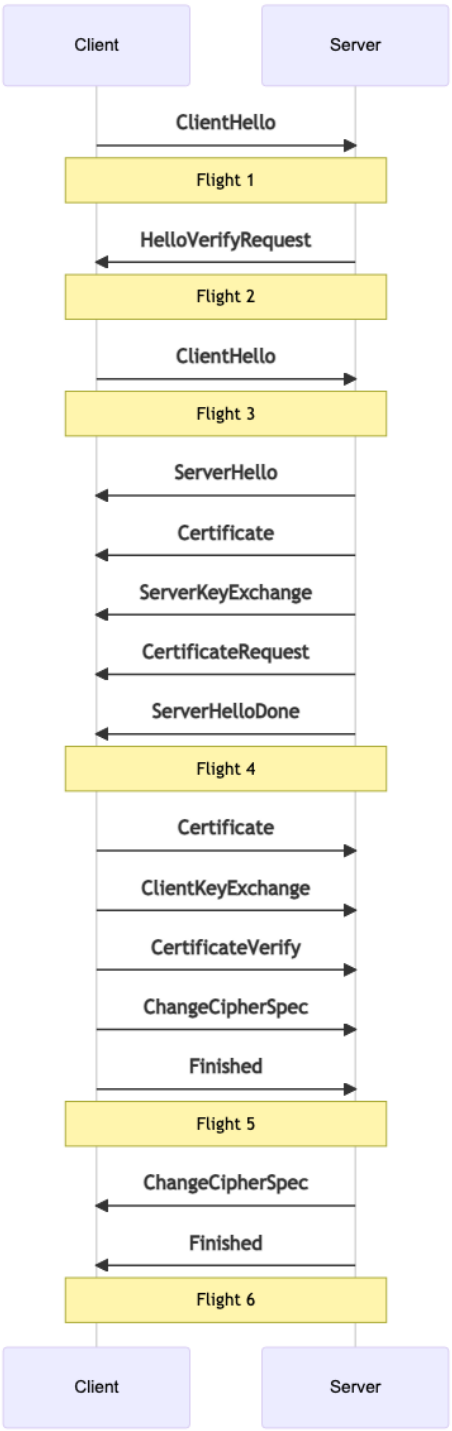


Figure 7: Рукопожатие

ClientHello ClientHello - это начальное сообщение, отправляемое клиентом. Оно содержит список атрибутов. Эти атрибуты сообщают серверу о шифрах и функциях, которые поддерживает клиент. Для WebRTC это также способ выбрать SRTP-шифр. Он также содержит случайные данные, которые будут использованы для генерации ключей для сеанса.

HelloVerifyRequest HelloVerifyRequest отправляется сервером клиенту. Это делается для того, чтобы убедиться, что клиент намеревался отправить запрос. Затем клиент повторно отправляет ClientHello, но с токеном, предоставленным в HelloVerifyRequest.

ServerHello ServerHello - это ответ сервера для конфигурации этого сеанса. Он содержит, какой шифр будет использоваться, когда этот сеанс закончится. Он также содержит случайные данные сервера.

Сертификат Сертификат содержит сертификат для клиента или сервера. Это используется для уникальной идентификации, с кем мы общались. После завершения рукопожатия мы убедимся, что этот сертификат при хешировании соответствует отпечатку в `SessionDescription`.

ServerKeyExchange/ClientKeyExchange Эти сообщения используются для передачи открытого ключа. При запуске клиент и сервер генерируют пару ключей. После рукопожатия эти значения будут использованы для генерации `Pre-Master Secret`.

CertificateRequest CertificateRequest отправляется сервером, уведомляя клиента, что он хочет сертификат. Сервер может либо запросить, либо потребовать сертификат.

ServerHelloDone ServerHelloDone уведомляет клиента, что сервер закончил рукопожатие.

CertificateVerify CertificateVerify - это способ, которым отправитель доказывает, что у него есть закрытый ключ, отправленный в сообщении Certificate.

ChangeCipherSpec ChangeCipherSpec информирует получателя, что все, отправленное после этого сообщения, будет зашифровано.

Finished Finished зашифрован и содержит хеш всех сообщений. Это утверждение, что рукопожатие не было изменено.

Генерация ключа

После завершения рукопожатия вы можете начать отправлять зашифрованные данные. Шифр был выбран сервером и находится в ServerHello. Как был выбран ключ?

Сначала мы генерируем `Pre-Master Secret`. Чтобы получить это значение, используется Диффи-Хеллман на ключах, обмененных `ServerKeyExchange` и `ClientKeyExchange`. Детали различаются в зависимости от выбранного шифра.

Затем генерируется `Master Secret`. Каждая версия DTLS имеет определенную `Псевдослучайную функцию`. Для DTLS 1.2 функция принимает `Pre-Master Secret` и случайные значения в `ClientHello` и `ServerHello`. Выход от запуска `Псевдослучайной функции` - это `Master Secret`. `Master Secret` - это значение, используемое для шифра.

Обмен ApplicationData

Основной движитель DTLS - `ApplicationData`. Теперь, когда у нас инициализирован шифр, мы можем начать шифровать и отправлять значения.

Сообщения `ApplicationData` используют заголовок DTLS, как описано ранее. `Полезная нагрузка` заполняется шифротекстом. Теперь у вас есть работающий DTLS-сеанс, и вы можете общаться безопасно.

DTLS имеет много других интересных функций, таких как повторные переговоры. Они не используются WebRTC, поэтому здесь не будут рассмотрены.

SRTP

SRTP - это протокол, специально разработанный для шифрования RTP-пакетов. Чтобы начать SRTP-сессию, вы указываете свои ключи и шифр. В отличие от DTLS, у него нет механизма рукопожатия. Вся конфигурация и ключи были сгенерированы во время DTLS-рукопожатия.

DTLS предоставляет выделенный API для экспорта ключей, которые будут использоваться другим процессом. Это определено в RFC 5705.

Создание сессии

SRTP определяет функцию генерации ключа, которая используется на входах. При создании SRTP-сессии входы проходят через нее для генерации ключей для нашего SRTP-шифра. После этого вы можете перейти к обработке медиа.

Обмен медиа

Каждый RTP-пакет имеет 16-битный порядковый номер. Эти порядковые номера используются для сохранения пакетов в порядке, как первичный ключ. Во время вызова они будут переполняться. SRTP отслеживает это и называет это счетчиком переполнения.

При шифровании пакета SRTP использует счетчик переполнения и порядковый номер в качестве nonce. Это для обеспечения того, что даже если вы отправите одни и те же данные дважды, шифротекст будет разным. Это важно для предотвращения возможности атакующего идентифицировать шаблоны или попытаться выполнить атаку повторного воспроизведения.

Сетевое взаимодействие в реальном времени

Почему сетевое взаимодействие так важно в коммуникации в реальном времени?

Сети являются ограничивающим фактором в коммуникации в реальном времени. В идеальном мире у нас была бы неограниченная пропускная способность, и пакеты прибывали бы мгновенно. Но это не так. Сети ограничены, и условия могут измениться в любой момент. Измерение и наблюдение сетевых условий также является сложной проблемой. Вы можете получить разное поведение в зависимости от оборудования, программного обеспечения и его конфигурации.

Коммуникация в реальном времени также создает проблему, которой не существует в большинстве других областей. Для веб-разработчика не фатально, если ваш сайт работает медленнее в некоторых сетях. Пока все данные прибывают, пользователи довольны. С WebRTC, если ваши данные опаздывают, они бесполезны. Никому не интересно, что было сказано в видеоконференции 5 секунд назад. Поэтому при разработке системы связи в реальном времени вам приходится идти на

компромисс. Каков ваш временной лимит и сколько данных вы можете отправить?

Эта глава охватывает концепции, применимые как к передаче данных, так и к медиа-коммуникации. В следующих главах мы выходим за рамки теории и обсуждаем, как подсистемы медиа и данных WebRTC решают эти проблемы.

Какие атрибуты сети делают ее сложной?

Код, эффективно работающий во всех сетях, сложен. Существует множество различных факторов, и они могут тонко влиять друг на друга. Вот наиболее распространенные проблемы, с которыми столкнутся разработчики.

Пропускная способность Пропускная способность - это максимальная скорость передачи данных по заданному маршруту. Важно помнить, что это не статичное число. Пропускная способность будет меняться вдоль маршрута по мере того, как больше (или меньше) людей его использует.

Время передачи и время круглого пути Время передачи - это время, за которое пакет достигает пункта назначения. Как и пропускная способность, это не константа. Время передачи может колебаться в любой момент.

```
transmission_time = receive_time - send_time
```

Чтобы вычислить время передачи, вам нужны синхронизированные с миллисекундной точностью часы на отправителе и получателе. Даже небольшое отклонение приведет к ненадежному измерению времени передачи. Поскольку WebRTC работает в крайне неоднородных средах, практически невозможно полагаться на идеальную синхронизацию времени между хостами.

Измерение времени круглого пути - это обходной путь для несовершенной синхронизации часов.

Вместо работы с распределенными часами WebRTC-пир отправляет специальный пакет со своей собственной меткой времени `sendertime1`. Сотрудничающий пир получает пакет и отражает метку времени обратно отправителю. Как только первоначальный отправитель получает отраженное время, он вычитает метку времени `sendertime1` из текущего времени `sendertime2`. Эта временная дельта называется “задержкой распространения круглого пути” или, чаще, временем круглого пути.

```
rtt = sendertime2 - sendertime1
```

Половина времени круглого пути считается достаточно хорошим приближением времени передачи. Этот обходной путь не лишен недостатков. Он предполагает, что на отправку и получение пакетов уходит одинаковое время. Однако в сотовых сетях операции отправки и получения могут быть не симметричны по времени. Вы могли заметить, что скорости загрузки на вашем телефоне почти всегда ниже скоростей скачивания.

```
transmission_time = rtt/2
```

Технические подробности измерения времени круглого пути описаны более подробно в главе о Sender и Receiver Reports RTCP.

Джиттер Джиттер - это факт, что `Время передачи` может варьироваться для каждого пакета. Ваши пакеты могут быть задержаны, но затем прибыть пачкой.

Потеря пакетов Потеря пакетов - это когда сообщения теряются при передаче. Потеря может быть стабильной или происходить скачкообразно. Это может быть связано с типом сети, например, спутниковой или Wi-Fi. Или может быть введена программным обеспечением по пути.

Максимальный блок передачи Максимальный блок передачи (Maximum Transmission Unit) - это ограничение на размер одного пакета. Сети не позволяют отправлять одно гигантское сообщение. На уровне протокола сообщения могут быть разделены на несколько более мелких пакетов.

MTU также будет различаться в зависимости от того, какой сетевой путь вы проходите. Вы можете использовать протокол, такой как Path MTU Discovery, чтобы определить максимальный размер пакета, который вы можете отправить.

Перегрузка

Перегрузка - это когда достигнуты пределы сети. Обычно это происходит, когда вы достигли пиковой пропускной способности, которую может обработать текущий маршрут. Или это может быть ограничение, наложенное оператором, например, почасовые лимиты, которые настраивает ваш интернет-провайдер.

Перегрузка проявляется по-разному. Нет стандартизированного поведения. В большинстве случаев при достижении перегрузки сеть будет сбрасывать избыточные пакеты. В других случаях сеть будет буферизовать. Это вызовет увеличение времени передачи для ваших пакетов. Вы также можете увидеть больше джиттера по мере перегрузки сети. Это быстро меняющаяся область, и новые алгоритмы обнаружения перегрузки все еще разрабатываются.

Динамичность

Сети крайне динамичны, и условия могут быстро меняться. Во время вызова вы можете отправить и получить сотни тысяч пакетов. Эти пакеты будут проходить через несколько транзитных узлов. Эти узлы будут совместно использоваться миллионами других пользователей. Даже в вашей локальной сети может загружаться HD-фильм или устройство может решить скачать обновление программного обеспечения.

Хороший вызов - это не просто измерение вашей сети при запуске. Вам нужно постоянно оценивать ситуацию. Вам также нужно справляться со всеми различными поведениями, возникающими из множества сетевого оборудования и программного обеспечения.

Решение проблемы потери пакетов

Обработка потери пакетов - первая проблема для решения. Существует несколько способов ее решения, каждый со своими преимуществами. Это зависит от того, что вы отправляете, и насколько вы терпимы к задержкам. Важно также отметить, что не все потери пакетов фатальны. Потеря некоторого видео может не быть проблемой, человеческий глаз может даже не заметить этого. Потеря текстовых сообщений пользователя фатальна.

Допустим, вы отправляете 10 пакетов, и пакеты 5 и 6 теряются. Вот способы решения этой проблемы.

Подтверждения

Подтверждения - это когда получатель уведомляет отправителя о каждом полученном пакете. Отправитель узнает о потере пакетов, когда получает подтверждение для пакета дважды, который не является окончательным. Когда отправитель

получает ACK для пакета 4 дважды, он знает, что пакет 5 еще не был виден.

Избирательные подтверждения

Избирательные подтверждения - это улучшение обычных подтверждений. Получатель может отправить SACK, который подтверждает несколько пакетов и уведомляет отправителя о пропусках. Теперь отправитель может получить SACK для пакетов 4 и 7. Он знает, что ему нужно повторно отправить пакеты 5 и 6.

Отрицательные подтверждения

Отрицательные подтверждения решают проблему противоположным образом. Вместо уведомления отправителя о том, что он получил, получатель уведомляет отправителя о том, что было потеряно. В нашем случае NACK будет отправлен для пакетов 5 и 6. Отправитель знает только пакеты, которые получатель хочет, чтобы были отправлены снова.

Упреждающая коррекция ошибок

Упреждающая коррекция ошибок исправляет потерю пакетов заблаговременно. Отправитель отправляет избыточные данные, что означает, что потерянный пакет не влияет на конечный поток. Один популярный алгоритм для этого - коррекция ошибок Рида-Соломона.

Это уменьшает задержку/сложность отправки и обработки подтверждений. Упреждающая коррекция ошибок - это напрасная трата пропускной способности, если в сети, в которой вы находитесь, нет потерь.

Решение джиттера

Джиттер присутствует в большинстве сетей. Даже внутри локальной сети есть много устройств, отправляющих данные с колеблющимися скоростями. Вы легко можете наблюдать джиттер, пингуя другое устройство командой `ping` и замечая колебания круглого пути задержки.

Чтобы решить джиттер, клиенты используют JitterBuffer. JitterBuffer обеспечивает стабильное время доставки пакетов. Недостаток заключается в том, что JitterBuffer добавляет некоторую задержку к пакетам, которые прибывают рано.

Преимущество в том, что поздние пакеты не вызывают джиттера. Представьте, что во время вызова вы видите следующие времена прибытия пакетов:

```
* time=1.46 ms
* time=1.93 ms
* time=1.57 ms
* time=1.55 ms
* time=1.54 ms
* time=1.72 ms
* time=1.45 ms
* time=1.73 ms
* time=1.80 ms
```

В этом случае около 1.8 мс было бы хорошим выбором. Пакеты, прибывающие поздно, будут использовать наш временной интервал задержки. Пакеты, прибывающие рано, будут немного задержаны и могут заполнить окно, истощенное поздними пакетами. Это означает, что мы больше не имеем заторов и обеспечиваем клиенту плавную скорость доставки.

Работа JitterBuffer

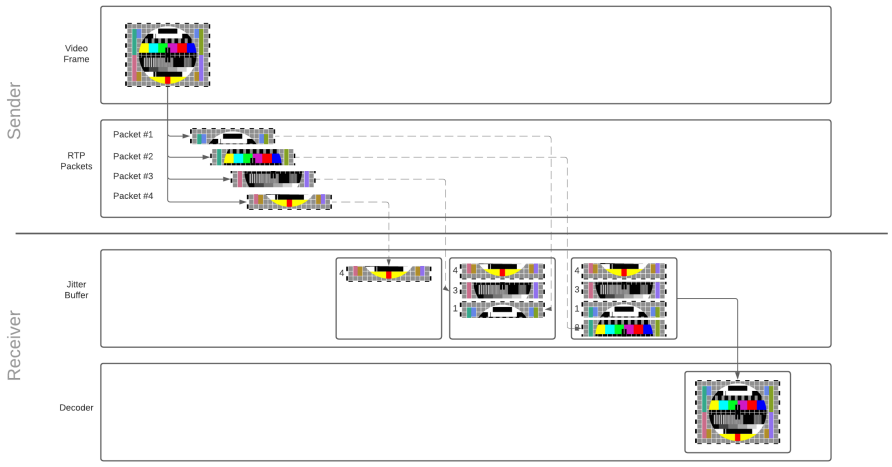


Figure 8: JitterBuffer

Каждый пакет добавляется в буфер джиттера сразу после получения. Как только накопится достаточно пакетов для реконструкции кадра, пакеты, составляющие кадр, освобождаются из буфера и выдаются для декодирования. Декодер, в свою очередь, декодирует и отрисовывает видеокадр на экране

пользователя. Поскольку буфер джиттера имеет ограниченную емкость, пакеты, которые слишком долго остаются в буфере, будут отброшены.

Подробнее о том, как видеокадры преобразуются в RTP-пакеты и почему необходима реконструкция, читайте в главе о медиа-коммуникации.

`jitterBufferDelay` предоставляет отличное представление о производительности вашей сети и ее влиянии на плавность воспроизведения. Это часть API статистики WebRTC, относящаяся к входящему потоку получателя. Задержка определяет время, которое видеокадры проводят в буфере джиттера перед выдачей для декодирования. Длинная задержка буфера джиттера означает, что ваша сеть сильно перегружена.

Обнаружение перегрузки

Прежде чем мы сможем разрешить перегрузку, нам нужно ее обнаружить. Для этого мы используем контроллер перегрузки. Это сложный предмет, который все еще быстро меняется. Новые алгоритмы все еще публикуются и тестируются. На высоком уровне они все работают одинаково. Контроллер перегрузки предоставляет оценки пропускной способности по некоторым входным данным. Вот некоторые возможные входные данные:

- **Потеря пакетов** - Пакеты сбрасываются по мере перегрузки сети.
- **Джиттер** - По мере перегрузки сетевого оборудования, постановка пакетов в очередь вызывает их нерегулярность.
- **Время круглого пути** - При перегрузке пакеты дольше прибывают. В отличие от джиттера, время круглого пути просто продолжает увеличиваться.
- **Явное уведомление о перегрузке** - Новые сети могут помечать пакеты как подверженные риску сброса для облегчения перегрузки.

Эти значения необходимо измерять непрерывно во время вызова. Использование сети может увеличиваться или уменьшаться, поэтому доступная пропускная способность может постоянно меняться.

Разрешение перегрузки

Теперь, когда у нас есть оценка пропускной способности, нам нужно скорректировать то, что мы отправляем. Как

мы корректируем, зависит от того, какие данные мы хотим отправить.

Отправка медленнее

Ограничение скорости, с которой вы отправляете данные, - первое решение для предотвращения перегрузки. Контроллер перегрузки дает вам оценку, и ответственность за ограничение скорости лежит на отправителе.

Этот метод используется для большинства коммуникаций с данными. С такими протоколами, как TCP, это все делается операционной системой и полностью прозрачно для пользователей и разработчиков.

Отправка меньшего количества

В некоторых случаях мы можем отправлять меньше информации, чтобы удовлетворить наши ограничения. У нас также есть жесткие сроки доставки наших данных, поэтому мы не можем отправлять медленнее. Это ограничения, которые относятся к медиа в реальном времени.

Если у нас недостаточно доступной пропускной способности, мы можем снизить качество отправляемого видео. Это требует тесной обратной связи между вашим видеокодировщиком и контроллером перегрузки.

Медиа-коммуникация

Что я получаю от медиа-коммуникации WebRTC?

WebRTC позволяет отправлять и получать неограниченное количество аудио- и видеопотоков. Вы можете добавлять и удалять эти потоки в любой момент во время вызова. Эти потоки могут быть полностью независимыми или объединенными! Вы можете отправлять видеопоток рабочего стола и одновременно включать аудио и видео с веб-камеры.

Протокол WebRTC не зависит от кодека. Базовый транспорт поддерживает все, даже то, что еще не существует! Однако WebRTC-агент, с которым вы общаетесь, может не иметь необходимых инструментов для его принятия.

WebRTC также разработан для работы с динамическими сетевыми условиями. Во время вызова ваша пропускная способность может увеличиваться или уменьшаться. Возможно,

вы внезапно испытаете большую потерю пакетов. Протокол разработан для работы со всем этим. WebRTC реагирует на сетевые условия и пытается обеспечить вам наилучший возможный опыт с доступными ресурсами.

Для WebRTC `Payload Type` является динамическим. VP8 в одном вызове может отличаться от другого. Инициатор вызова определяет сопоставление `Payload Types` с кодеками в `Session Description`.

Номер последовательности `Номер последовательности` используется для упорядочивания пакетов в потоке. Каждый раз при отправке пакета `Номер последовательности` увеличивается на единицу.

RTP разработан для работы в сетях с потерями. Это дает получателю способ обнаруживать потерянные пакеты.

Временная метка Момент выборки для этого пакета. Это не глобальные часы, а то, сколько времени прошло в медиапотоке. Несколько RTP-пакетов могут иметь одну и ту же временную метку, например, если они все являются частью одного видеокадра.

Источник синхронизации (SSRC) `SSRC` - это уникальный идентификатор для этого потока. Это позволяет запускать несколько потоков медиа через один RTP-поток.

Источник вклада (CSRC) Список, который сообщает, какие `SSRC` внесли вклад в этот пакет.

Обычно это используется для индикаторов разговора. Допустим, на стороне сервера вы объединили несколько аудиопотоков в один RTP-поток. Затем вы можете использовать это поле, чтобы сказать “Входные потоки A и C говорили в этот момент”.

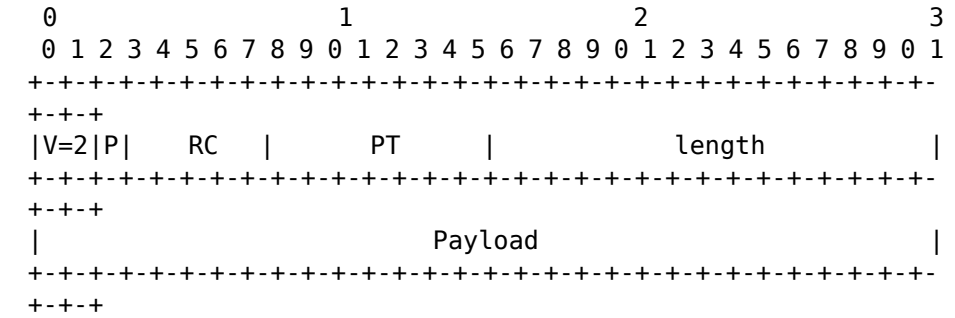
Полезная нагрузка Сами данные полезной нагрузки. Может заканчиваться счетчиком добавленных байтов заполнения, если установлен флаг заполнения.

Расширения

RTCP

Формат пакета

Каждый RTCP-пакет имеет следующую структуру:



Версия (V) Версия всегда равна 2.

Заполнение (P) Заполнение - это булево значение, которое контролирует наличие заполнения в полезной нагрузке.

Последний байт полезной нагрузки содержит счетчик добавленных байтов заполнения.

Счетчик отчетов о приеме (RC) Количество отчетов в этом пакете. Один RTCP-пакет может содержать несколько событий.

Тип пакета (PT) Уникальный идентификатор типа RTCP-пакета. WebRTC-агент не обязан поддерживать все эти типы, и поддержка между агентами может различаться. Однако вот наиболее часто встречающиеся типы:

- 192 - Запрос полного I-кадра (FIR)
- 193 - Отрицательные подтверждения (NACK)
- 200 - Отчет отправителя
- 201 - Отчет получателя
- 205 - Общий RTP-отзыв
- 206 - Отзыв, специфичный для полезной нагрузки

Значимость этих типов пакетов будет описана более подробно ниже.

Запрос полного I-кадра (FIR) и Индикация потери изображения (PLI)

Сообщения FIR и PLI служат похожей цели. Эти сообщения запрашивают полный ключевой кадр от отправителя. PLI используется, когда частичные кадры были переданы декодеру, но он не смог их декодировать.

Это может произойти из-за большой потери пакетов или, возможно, сбоя декодера.

Согласно RFC 5104, FIR не должен использоваться при потере пакетов или кадров. Это работа PLI. FIR запрашивает ключевой кадр по причинам, отличным от потери пакетов - например, когда новый участник входит в видеоконференцию. Им нужен полный ключевой кадр, чтобы начать декодирование видеопотока, декодер будет отбрасывать кадры до прихода ключевого кадра.

Хорошей идеей является запрос полного ключевого кадра сразу после подключения, это минимизирует задержку между подключением и появлением изображения на экране пользователя.

PLI-пакеты являются частью сообщений с отзывом, специфичным для полезной нагрузки.

На практике программное обеспечение, способное обрабатывать как PLI, так и FIR-пакеты, будет действовать одинаково в обоих случаях. Оно отправит сигнал кодировщику создать новый полный ключевой кадр.

Отрицательное подтверждение

NACK запрашивает у отправителя повторную передачу одного RTP-пакета. Обычно это вызвано потерей RTP-пакета, но может произойти и из-за его опоздания.

NACK гораздо эффективнее с точки зрения пропускной способности, чем запрос повторной отправки всего кадра. Поскольку RTP разбивает пакеты на очень маленькие куски, вы фактически запрашиваете только один небольшой отсутствующий фрагмент. Получатель создает RTCP-сообщение с SSRC и номером последовательности. Если отправитель не имеет этого RTP-пакета для повторной отправки, он просто игнорирует сообщение.

Отчеты получателя / Отчеты отправителя

Первая реализация - это пара Отчетов получателя и его дополнение, Отчеты отправителя. Эти RTCP-сообщения определены в RFC 3550 и отвечают за обмен информацией о сетевом статусе между конечными точками. Отчеты получателя фокусируются на сообщении качеств сети (включая потерю пакетов, время кругового обхода и джиттер), и они сочетаются с другими алгоритмами, которые затем отвечают за оценку доступной пропускной способности на основе этих отчетов.

Отчеты отправителя и получателя (SR и RR) вместе рисуют картину качества сети. Они отправляются по расписанию для каждого SSRC и являются входными данными, используемыми при оценке доступной пропускной способности. Эти оценки делаются отправителем после получения данных RR, содержащих следующие поля:

- **Доля потерь** - Какой процент пакетов был потерян с момента последнего Отчета получателя.
- **Совокупное количество потерянных пакетов** - Сколько пакетов было потеряно за весь звонок.
- **Расширенный наивысший номер последовательности, полученный** - Какой был последний полученный номер последовательности и сколько раз он переполнялся.
- **Межприбытийный джиттер** - Текущий джиттер за весь звонок.
- **Временная метка последнего отчета отправителя** - Последнее известное время на отправителе, используется для расчета времени кругового обхода.

SR и RR работают вместе для вычисления времени кругового обхода.

Отправитель включает свое локальное время, `sendertime1` в SR. Когда получатель получает пакет SR, он отправляет обратно RR. В числе прочего, RR включает `sendertime1`, только что полученный от отправителя. Между получением SR и отправкой RR будет задержка. Из-за этого RR также включает время “задержки с момента последнего отчета отправителя” - `DLSR`. `DLSR` используется для корректировки оценки времени кругового обхода позже в процессе. Как только отправитель получает RR, он вычитает `sendertime1` и `DLSR` из текущего времени `sendertime2`. Эта разница времени называется задержкой распространения кругового обхода или временем кругового обхода.

```
rtt = sendertime2 - sendertime1 - DLSR
```

Время кругового обхода простым языком: - Я отправляю вам

сообщение с текущим показанием моих часов, скажем, 4:20 вечера, 42 секунды и 420 миллисекунд. - Вы отправляете мне обратно эту же временную метку. - Вы также включаете время, прошедшее с момента чтения моего сообщения до отправки ответа, скажем, 5 миллисекунд. - Когда я получаю время обратно, я смотрю на часы снова. - Теперь мои часы показывают 4:20 вечера, 42 секунды 690 миллисекунд. - Это означает, что на то, чтобы дойти до вас и вернуться обратно ко мне, ушло 265 миллисекунд (690 - 420 - 5). - Следовательно, время кругового обхода составляет 265 миллисекунд.

TMMBR, TMMBN, REMB и TWCC, сопряженные с GCC

Google Congestion Control (GCC) Алгоритм Google Congestion Control (GCC) (описанный в draft-ietf-rmcat-gcc-02) решает задачу оценки пропускной способности. Он сочетается с различными другими протоколами для облегчения связанных коммуникационных требований. Следовательно, он хорошо подходит для работы как на стороне получателя (при работе с TMMBR/TMMBN или REMB), так и на стороне отправителя (при работе с TWCC).

Чтобы прийти к оценкам доступной пропускной способности, GCC фокусируется на потере пакетов и флуктуациях времени прибытия кадров как на двух основных метриках. Он пропускает эти метрики через два связанных контроллера: контроллер, основанный на потерях, и контроллер, основанный на задержках.

Первый компонент GCC, контроллер, основанный на потерях, прост:

- Если потеря пакетов выше 10%, оценка пропускной способности уменьшается.
- Если потеря пакетов между 2-10%, оценка пропускной способности остается той же.
- Если потеря пакетов ниже 2%, оценка пропускной способности увеличивается.

Измерения потери пакетов производятся часто. В зависимости от сопряженного коммуникационного протокола, потеря пакетов может быть либо явно сообщена (как с TWCC), либо выведена (как с TMMBR/TMMBN и REMB). Эти проценты оцениваются в окнах времени около одной секунды.

Контроллер, основанный на задержках, сотрудничает с контроллером, основанным на потерях, и смотрит на вариации времени прибытия пакетов. Этот контроллер, основанный на задержках, стремится определить, когда сетевые каналы

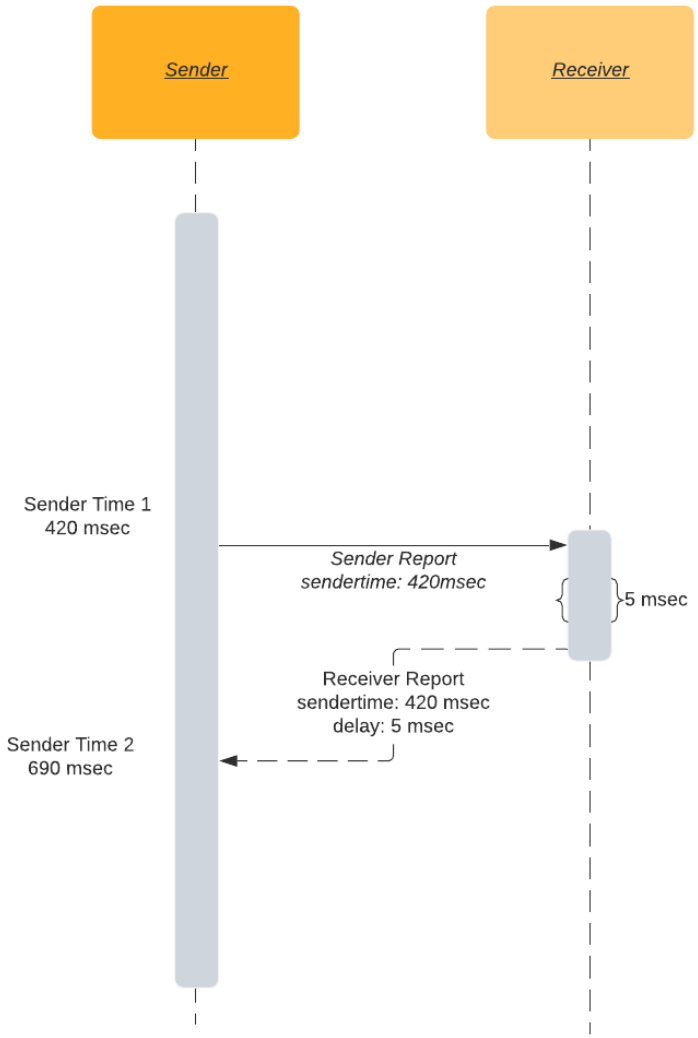


Figure 9: Время кругового обхода

становятся все более перегруженными, и может уменьшать оценки пропускной способности даже до возникновения потери пакетов. Теория состоит в том, что самый загруженный сетевой интерфейс вдоль пути будет продолжать накапливать пакеты, пока интерфейс не исчерпает емкость своих буферов. Если этот интерфейс продолжает получать больше трафика, чем он способен отправить, он будет вынужден отбрасывать все пакеты, которые не могут поместиться в его буферное пространство. Этот тип потери пакетов особенно разрушителен для коммуникации с низкой задержкой/в реальном времени, но он также может снизить пропускную способность для всей коммуникации через этот канал и должен, по возможности, избегаться. Таким образом, GCC пытается определить, растут ли сетевые каналы по глубине очередей *перед* фактической потерей пакетов. Он уменьшит использование пропускной способности, если наблюдает увеличение задержек очередей с течением времени.

Чтобы достичь этого, GCC пытается вывести увеличение глубины очереди, измеряя тонкие увеличения времени кругового обхода. Он записывает “межприбытийное время” кадров, `t(i) - t(i-1)`: разницу во времени прибытия двух групп пакетов (обычно последовательных видеокадров). Эти группы пакетов часто отправляются с регулярными интервалами времени (например, каждые 1/24 секунды для видео с 24 кадрами в секунду). В результате измерение межприбытийного времени сводится к простой записи разницы времени между началом первой группы пакетов (т.е. кадра) и первым кадром следующей.

На диаграмме ниже медианное увеличение задержки между пакетами составляет +20 мс, что является четким индикатором сетевой перегрузки.

Если межприбытийное время увеличивается со временем, это считается предполагаемым доказательством увеличения глубины очереди на соединительных сетевых интерфейсах и рассматривается как сетевая перегрузка. (Примечание: GCC достаточно умен, чтобы контролировать эти измерения с учетом флуктуаций размеров кадров.) GCC уточняет свои измерения задержки, используя фильтр Калмана, и делает много измерений времени кругового обхода сети (и его вариаций) перед обозначением перегрузки. Можно рассматривать фильтр Калмана GCC как замену линейной регрессии: помогающую делать точные прогнозы даже когда джиттер добавляет шум в измерения времени. При обнаружении перегрузки GCC уменьшит доступный битрейт. В качестве альтернативы, при стабильных сетевых условиях, он может медленно увеличивать

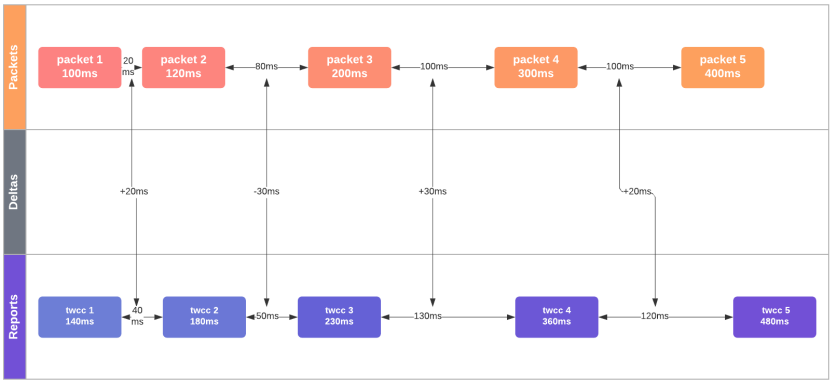


Figure 10: TWCC с задержкой

оценки пропускной способности, чтобы проверить более высокие значения нагрузки.

TMMBR, TMMBN и REMB Для TMMBR/TMMBN и REMB сторона получателя сначала оценивает доступную входящую пропускную способность (используя протокол, такой как GCC), а затем сообщает эти оценки пропускной способности удаленным отправителям. Им не нужно обмениваться подробностями о потере пакетов или других качествах сетевой перегрузки, потому что работа на стороне получателя позволяет им напрямую измерять межприбытийное время и потерю пакетов. Вместо этого TMMBR, TMMBN и REMB обмениваются только самими оценками пропускной способности:

- **Временный максимальный запрос скорости потока медиа** - Мантисса/экспонента запрошенной скорости передачи для одного SSRC.
- **Временное максимальное уведомление о скорости потока медиа** - Сообщение для уведомления о получении TMMBR.
- **Максимальная скорость, оцененная получателем** - Мантисса/экспонента запрошенной скорости передачи для всей сессии.

TMMBR и TMMBN появились первыми и определены в RFC 5104. REMB появился позже, был представлен черновик в draft-alvestrand-rmcat-remb, но он никогда не был стандартизирован.

Пример сеанса, использующего REMB, может выглядеть следующим образом:

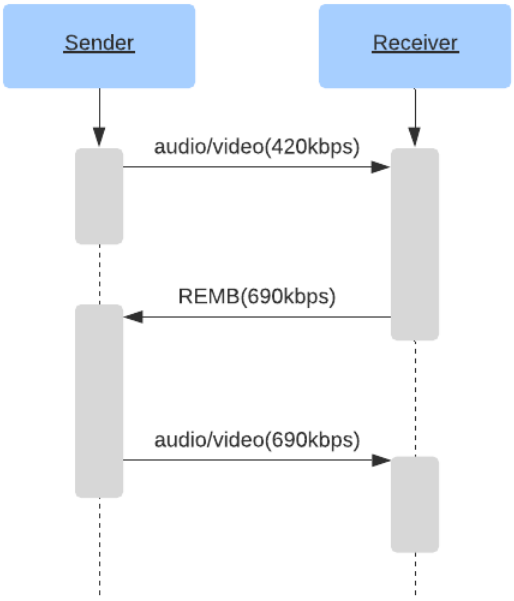


Figure 11: REMB

Этот метод отлично работает на бумаге. Отправитель получает оценку от получателя, устанавливает битрейт кодировщика на полученное значение. Вуаля! Мы адаптировались к сетевым условиям.

Однако на практике подход REMB имеет несколько недостатков.

Первый - неэффективность кодировщика. Когда вы устанавливаете битрейт для кодировщика, он не обязательно выведет точно запрошенный битрейт. Кодирование может выводить больше или меньше битов в зависимости от настроек кодировщика и кодируемого кадра.

Например, использование кодировщика x264 с `tune=zerolatency` может значительно отклоняться от указанного целевого битрейта. Вот возможный сценарий:

- Допустим, мы начинаем с установки битрейта на 1000 кбит/с.
- Кодировщик выводит только 700 кбит/с, потому что недостаточно высокочастотных признаков для кодирования. (Или - “смотрит на стену”.)
- Допустим также, что получатель получает видео 700 кбит/с с нулевой потерей пакетов. Затем он применяет правило REMB 1 для увеличения входящего битрейта на 8%.
- Получатель отправляет REMB-пакет с предложением 756 кбит/с (700 кбит/с * 1,08) отправителю.
- Отправитель устанавливает битрейт кодировщика на 756 кбит/с.
- Кодировщик выводит еще меньший битрейт.
- Этот процесс продолжает повторяться, снижая битрейт до абсолютного минимума.

Вы можете видеть, как это вызовет тяжелую настройку параметров кодировщика и удивит пользователей неприемлемым видео даже при отличном подключении.

Контроль перегрузки по всей транспортной сети Контроль перегрузки по всей транспортной сети (Transport Wide Congestion Control) является последним достижением в коммуникации сетевого статуса RTCP. Он определен в draft-holmer-rmcat-transport-wide-cc-extensions-01, но также никогда не был стандартизирован.

TWCC использует довольно простой принцип:

С REMB получатель инструктирует сторону отправки о доступной скорости загрузки. Он использует точные измерения

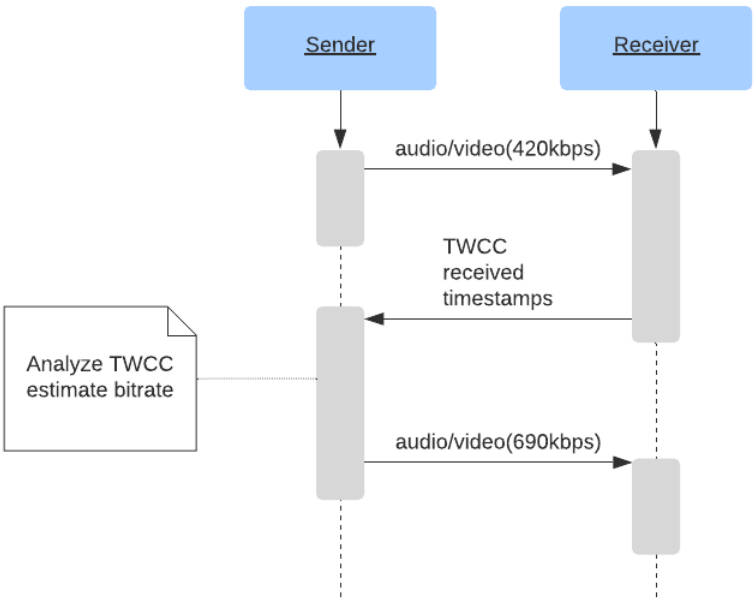


Figure 12: TWCC

о выведенной потере пакетов и данные, которые есть только у него, о межпакетном времени прибытия.

TWCC - это почти гибридный подход между поколениями протоколов SR/RR и REMB. Он возвращает оценки пропускной способности на сторону отправки (похоже на SR/RR), но техника оценки пропускной способности более близка к поколению REMB.

С TWCC получатель сообщает отправителю время прибытия каждого пакета. Этой информации достаточно для того, чтобы отправитель измерил вариацию задержки межпакетного прибытия, а также определил, какие пакеты были потеряны или пришли слишком поздно, чтобы внести вклад в аудио- или видеопоток. При частом обмене этими данными отправитель может быстро адаптироваться к изменяющимся сетевым условиям и варьировать свою выходную пропускную способность, используя такой алгоритм, как GCC.

Отправитель отслеживает отправленные пакеты, их последовательные номера, размеры и временные метки. Когда отправитель получает RTCP-сообщения от получателя, он сравнивает межпакетные задержки отправки с задержками получения. Если задержки получения увеличиваются, это сигнализирует о сетевой перегрузке, и отправитель должен принять корректирующие меры.

Предоставляя отправителю необработанные данные, TWCC обеспечивает превосходный обзор сетевых условий в реальном времени: - Почти мгновенное поведение потери пакетов вплоть до отдельных потерянных пакетов - Точная скорость отправки - Точная скорость получения - Измерение джиттера - Различия между задержками отправки и получения пакетов - Описание того, как сеть справлялась с импульсной или стабильной доставкой пропускной способности

Одним из самых значительных вкладов TWCC является гибкость, которую он предоставляет разработчикам WebRTC. Консолидируя алгоритм контроля перегрузки на стороне отправки, он позволяет использовать простой клиентский код, который может быть широко использован и требует минимальных улучшений со временем. Сложные алгоритмы контроля перегрузки затем могут быстрее итерироваться на оборудовании, которое они напрямую контролируют (например, на Selective Forwarding Unit, обсуждаемом в разделе 8). В случае браузеров и мобильных устройств это означает, что эти клиенты могут извлечь выгоду из улучшений алгоритма без необходимости ожидания стандартизации или обновлений

браузера (которые могут занять очень много времени, чтобы стать широко доступными).

Альтернативы оценки пропускной способности

Наиболее распространенной реализацией является “Алгоритм Google Congestion Control для коммуникации в реальном времени”, определенный в draft-alvestrand-rmcat-congestion.

Существует несколько альтернатив GCC, например, NADA: Унифицированная схема контроля перегрузки для медиа в реальном времени и SCReAM - Самосинхронизирующаяся адаптация скорости для мультимедиа.

Задержка против Качества

Медиа в реальном времени - это компромисс между задержкой и качеством. Чем больше задержки вы готовы терпеть, тем более высокого качества видео вы можете ожидать.

Реальные ограничения

Все эти ограничения вызваны ограничениями реального мира. Это все характеристики вашей сети, которые вам нужно преодолеть.

Видео сложно

Транспортировка видео - это не просто. Чтобы сохранить 30 минут несжатого 720p 8-битного видео, вам нужно около 110 ГБ. С такими цифрами видеоконференция из 4 человек не состоится. Нам нужен способ сделать это меньше, и ответ - сжатие видео. Но это не обходится без недостатков.

Видео 101

Мы не будем подробно рассматривать сжатие видео, а только настолько, чтобы понять, почему RTP устроен именно так. Сжатие видео кодирует видео в новый формат, который требует меньше битов для представления того же видео.

Сжатие с потерями и без потерь

Вы можете закодировать видео без потерь (никакая информация не теряется) или с потерями (информация может быть потеряна).

Поскольку кодирование без потерь требует отправки большего количества данных пиру, что приводит к потоку с большей задержкой и большим количеством потерянных пакетов, RTP обычно использует сжатие с потерями, даже если качество видео не будет таким хорошим.

Внутрикадровое и межкадровое сжатие

Сжатие видео бывает двух типов. Первый - внутрикадровое. Внутрикадровое сжатие уменьшает количество битов, используемых для описания одного видеокадра. Те же методы используются для сжатия неподвижных изображений, например, метод сжатия JPEG.

Второй тип - межкадровое сжатие. Поскольку видео состоит из множества картинок, мы ищем способы не отправлять одну и ту же информацию дважды.

Типы межкадрового сжатия

Затем у вас есть три типа кадров:

- **I-кадр** - Полная картинка, может быть декодирована без чего-либо еще.
- **P-кадр** - Частичная картинка, содержащая только изменения от предыдущей картинки.
- **B-кадр** - Частичная картинка, является модификацией предыдущих и будущих картинок.

Ниже представлена визуализация трех типов кадров.

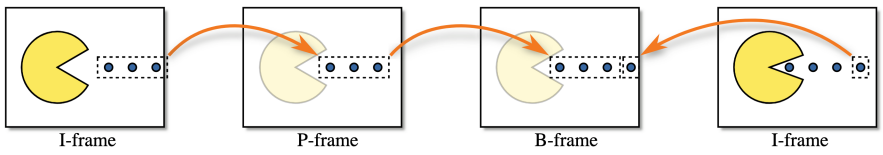


Figure 13: Типы кадров

Видео хрупко

Сжатие видео крайне зависимо от состояния, что делает его сложным для передачи через интернет. Что произойдет, если вы потеряете часть I-кадра? Как P-кадр узнает, что модифицировать? По мере усложнения видеосжатия эта

проблема становится еще более актуальной. К счастью, RTP и RTCP имеют решение.

Коммуникация данных

Что я получаю от коммуникации данных WebRTC?

WebRTC предоставляет каналы данных для обмена данными. Между двумя пирами вы можете открыть 65 534 канала данных. Канал данных основан на датаграммах, и у каждого есть свои настройки надежности. По умолчанию каждый канал данных имеет гарантированную упорядоченную доставку.

Если вы подходите к WebRTC с точки зрения медиа, каналы данных могут показаться бесполезными. Зачем вам вся эта подсистема, когда вы могли бы просто использовать HTTP или WebSocket?

Реальная сила каналов данных заключается в том, что вы можете настроить их так, чтобы они работали как UDP с неупорядоченной/потерянной доставкой. Это необходимо для ситуаций с низкой задержкой и высокой производительностью. Вы можете измерять обратное давление и убеждаться, что отправляете только то, что поддерживает ваша сеть.

Как это работает?

WebRTC использует протокол управления потоком передачи (Stream Control Transmission Protocol, SCTP), определенный в RFC 4960. SCTP - это транспортный протокол, который был задуман как альтернатива TCP или UDP. Для WebRTC мы используем его как протокол прикладного уровня, работающий поверх нашего DTLS-соединения.

SCTP дает вам потоки, и каждый поток может быть настроен независимо. Каналы данных WebRTC - это просто тонкие абстракции поверх них. Настройки надежности и упорядочивания просто передаются прямо в SCTP-агент.

Каналы данных имеют некоторые функции, которые SCTP не может выразить, например, метки каналов. Для решения этого WebRTC использует протокол установления канала данных (Data Channel Establishment Protocol, DCEP), который определен в RFC 8832. DCEP определяет сообщение для обмена меткой канала и протоколом.

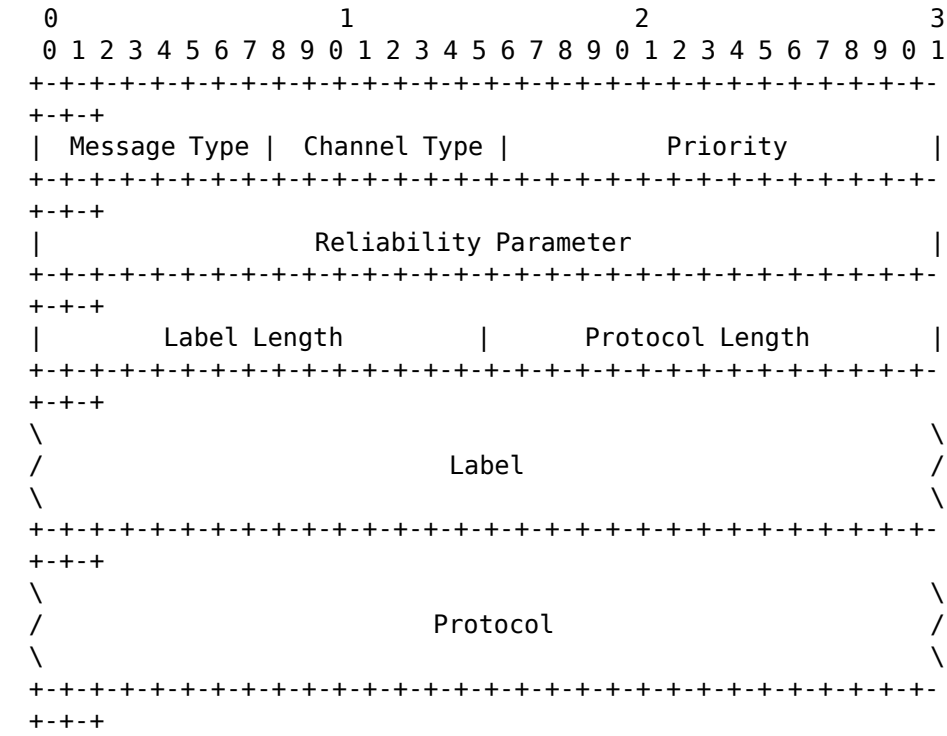
DCEP

DCEP только имеет два сообщения `DATA_CHANNEL_OPEN` и `DATA_CHANNEL_ACK`. Для каждого открытого канала данных удаленный должен ответить с подтверждением.

DATA_CHANNEL_OPEN

Это сообщение отправляется WebRTC-агентом, который хочет открыть канал.

Формат пакета



Тип сообщения Тип сообщения является статическим значением `0x03`.

Тип канала Тип канала контролирует атрибуты надежности/упорядочивания канала. Он может иметь следующие значения:

- `DATA_CHANNEL_RELIABLE (0x00)` - Сообщения не теряются и приходят в порядке

- DATA_CHANNEL_RELIABLE_UNORDERED (0x80) - Сообщения не теряются, но они могут приходить в неправильном порядке.
- DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT (0x01) - Сообщения могут теряться после попытки запрошенного количества раз, но они приходят в порядке.
- DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT_UNORDERED (0x81) - Сообщения могут теряться после попытки запрошенного количества раз и могут приходить в неправильном порядке.
- DATA_CHANNEL_PARTIAL_RELIABLE_TIMED (0x02) - Сообщения могут теряться, если они не приходят в запрошенное количество времени, но они приходят в порядке.
- DATA_CHANNEL_PARTIAL_RELIABLE_TIMED_UNORDERED (0x82) - Сообщения могут теряться, если они не приходят в запрошенное количество времени и могут приходить в неправильном порядке.

Приоритет Приоритет канала данных. Каналы данных с более высоким приоритетом будут запускаться первыми. Большие сообщения пользователя с низким приоритетом не будут задерживать отправку сообщений с более высоким приоритетом.

Параметр надежности Если тип канала данных DATA_CHANNEL_PARTIAL_RELIABLE, суффиксы конфигурируют поведение:

- REXMIT - Определяет, сколько раз отправитель будет передавать сообщение перед отказом.
- TIMED - Определяет, сколько времени (в мс) отправитель будет передавать сообщение перед отказом.

Метка UTF-8-кодированная строка, содержащая имя канала данных. Эта строка может быть пустой.

Протокол Если это пустая строка, протокол не указан. Если это непустая строка, она должна указывать на зарегистрированный протокол в “WebSocket Subprotocol Name Registry”, определенном в RFC 6455.

DATA_CHANNEL_ACK

Это сообщение отправляется WebRTC-агентом для подтверждения того, что этот канал данных открыт.

Формат пакета

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+
|  Message Type |
+-+-+-+-+-+-+-+-+
```

Stream Control Transmission Protocol

SCTP является реальной силой каналов данных WebRTC. Он предоставляет все эти функции канала данных:

- Мультиплексирование
- Надежная доставка с использованием механизма передачи TCP-подобной передачи
- Опции частичной надежности
- Предотвращение перегрузки
- Управление потоком

Чтобы понять SCTP, мы рассмотрим его в трех частях. Цель состоит в том, чтобы вы узнали достаточно, чтобы отладить и изучить глубокие детали SCTP самостоятельно после этой главы.

Концепции

SCTP является протоколом с богатыми функциями. Этот раздел будет охватывать только части SCTP, используемые WebRTC. Функции в SCTP, которые не используются WebRTC, включают мультихостинг и выбор пути.

С более чем двадцатилетним развитием SCTP может быть сложно полностью понять.

Ассоциация

Ассоциация - это термин для SCTP-сессии. Это состояние, которое разделяется между двумя SCTP-агентами, когда они общаются.

Потоки

Поток - это однонаправленная последовательность данных пользователя. Когда вы создаете канал данных, вы фактически создаете SCTP-поток. Каждая SCTP-ассоциация содержит список потоков. Каждый поток может быть настроен с разными типами надежности.

WebRTC позволяет вам настраивать только один поток при создании, но SCTP фактически позволяет изменять конфигурацию в любое время.

Датаграмма

SCTP передает данные как датаграммы, а не как байтовый поток. Отправка и получение данных напоминает использование UDP вместо TCP. Вам не нужно добавлять дополнительный код для передачи нескольких файлов по одному потоку.

SCTP-сообщения не имеют ограничений по размеру, как UDP. Одно SCTP-сообщение может быть размером в несколько гигабайт.

Части

Протокол SCTP состоит из частей. Существует множество различных типов частей. Эти части используются для всех коммуникаций. Данные пользователя, инициализация соединения, управление перегрузкой, и многое другое делается через части.

Каждый SCTP-пакет содержит список частей. Таким образом, в одном UDP-пакете можно иметь несколько частей, несущих сообщения из разных потоков.

Номер последовательности передачи

Номер последовательности передачи (TSN) является глобальным уникальным идентификатором для частей DATA. Части DATA несут все сообщения, которые пользователь хочет отправить. TSN важен, потому что он помогает получателю определить, являются ли пакеты потерянными или неправильными.

Если получатель замечает отсутствующий TSN, он не передает данные пользователю до тех пор, пока он не будет выполнен.

Идентификатор потока

Каждый поток имеет уникальный идентификатор. Когда вы создаете канал данных с явным ID, он фактически передается прямо в SCTP как идентификатор потока. Если вы не передаете ID, идентификатор потока выбирается для вас.

Идентификатор протокола полезной нагрузки

Каждая часть DATA также имеет Идентификатор протокола полезной нагрузки (PPID). Это используется для уникальной идентификации типа данных, который обменивается. SCTP имеет много PPIDs, но WebRTC использует только следующие пять:

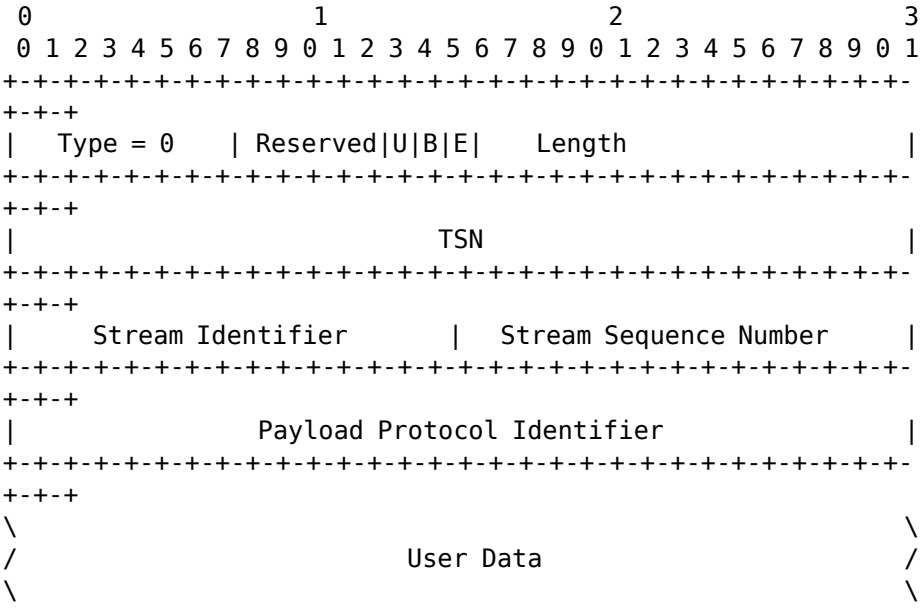
- `WebRTC DCEP` (`50`) - Сообщения DCEP.
- `WebRTC String` (`51`) - Сообщения канала данных String.
- `WebRTC Binary` (`53`) - Сообщения канала данных Binary.
- `WebRTC String Empty` (`56`) - Сообщения канала данных String с длиной 0.
- `WebRTC Binary Empty` (`57`) - Сообщения канала данных Binary с длиной 0.

Протокол

Следующие некоторые из частей, используемых протоколом SCTP. Это не является исчерпывающим представлением. Это обеспечивает достаточные структуры для состояние машины для понимания.

Каждая часть начинается с поля `type`. Перед списком частей также имеется заголовок.

Части DATA



```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Части DATA являются способом обмена всеми данными пользователя. Когда вы отправляете что-либо по каналу данных, это как это происходит.

`U` бит установлен, если это неупорядоченный пакет. Мы можем игнорировать Номер последовательности потока.

`B` и `E` - это биты начала и конца. Если вы хотите отправить сообщение, которое слишком велико для одной части DATA, оно должно быть фрагментировано на несколько частей DATA, отправленных в отдельных пакетах. С битами `B` и `E` и номерами последовательностей SCTP может выразить это.

- `B=1`, `E=0` - Первая часть фрагментированного сообщения пользователя.
- `B=0`, `E=0` - Средняя часть фрагментированного сообщения пользователя.
- `B=0`, `E=1` - Последняя часть фрагментированного сообщения пользователя.
- `B=1`, `E=1` - Нефрагментированное сообщение.

`TSN` - это номер последовательности передачи. Это глобальный уникальный идентификатор для этой части DATA. После 4,294,967,295 частей это обернется к 0. TSN увеличивается для каждой части фрагментированного сообщения пользователя, чтобы получатель знал, как упорядочить полученные части для восстановления исходного сообщения.

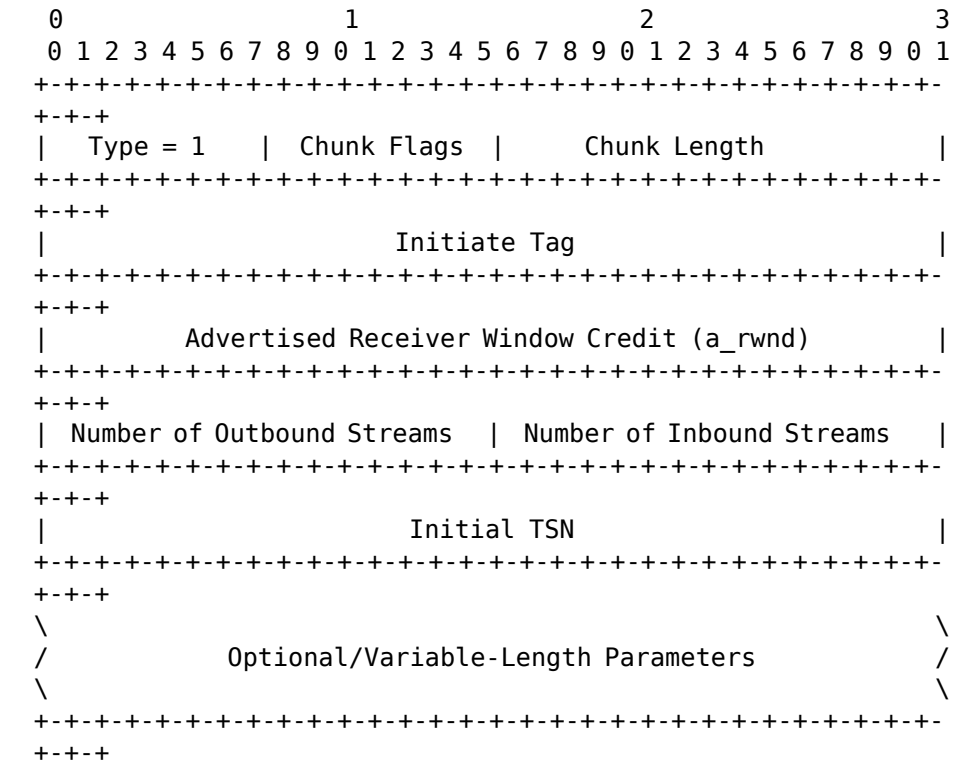
`Идентификатор потока` - это уникальный идентификатор потока, к которому принадлежит эта данные.

`Номер последовательности потока` - это 16-битное число, увеличивающееся с каждым сообщением пользователя и включаемое в заголовок части DATA-сообщения. После 65535 сообщений это обернется к 0. Это число используется для определения порядка доставки сообщений получателю, если `U` установлен в 0. Подобно TSN, за исключением того, что Номер последовательности потока увеличивается только для целого сообщения, а не для каждой отдельной части DATA.

`Идентификатор протокола полезной нагрузки` - это тип данных, который течет через этот поток. Для WebRTC это будет DCEP, String или Binary.

`Данные пользователя` - это то, что вы отправляете. Все данные, которые вы отправляете по каналу данных WebRTC передаются через часть DATA.

INIT Chunk



Часть INIT начинает процесс создания ассоциации.

`Initiate Tag` используется для генерации файла cookie. Файлы cookie используются для Man-In-The-Middle и защиты от DoS-атак. Они описываются более подробно в разделе состояние машины.

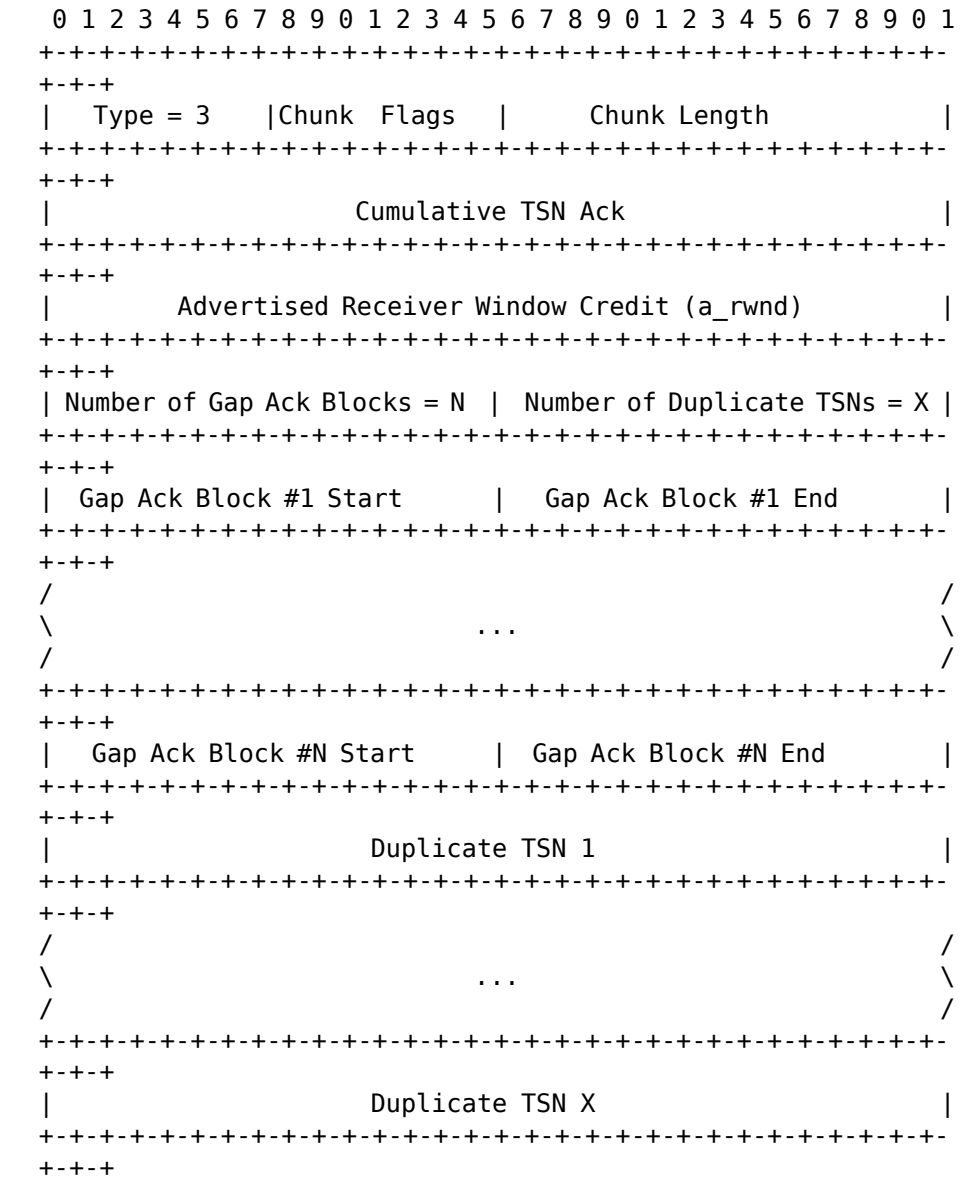
`Advertised Receiver Window Credit` используется для управления перегрузкой SCTP. Это сообщает, насколько большой буфер имеется у получателя для этой ассоциации.

`Число исходящих/входящих потоков` уведомляет удаленный о том, сколько потоков поддерживает этот агент.

`Initial TSN` - это случайный `uint32` для запуска локального TSN.

`Опциональные параметры` позволяют SCTP вводить новые функции в протокол.

SACK Chunk



Часть SACK (Selective Acknowledgment) уведомляет отправитель о том, что он получил пакет. Пока отправитель не получит SACK для TSN он будет передавать часть DATA в вопрос. SACK делает больше, чем просто обновление TSN.

`Cumulative TSN ACK` - это самый высокий TSN, который был получен.

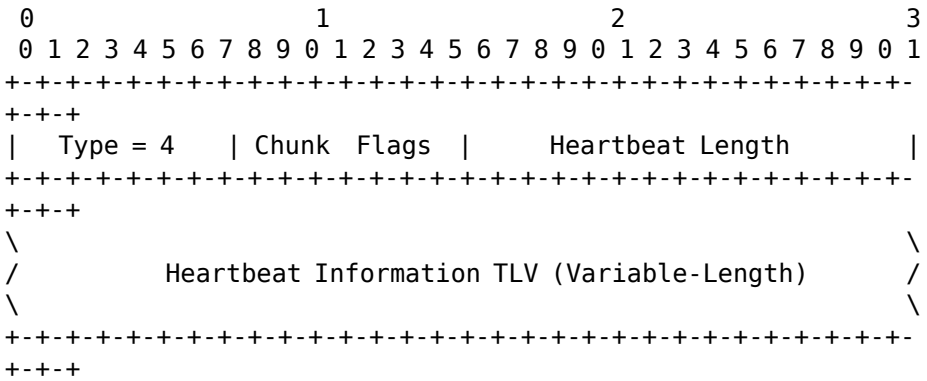
`Advertised Receiver Window Credit` - размер буфера получателя.

Получатель может изменить это во время сессии, если становится доступно больше памяти.

`Ack Blocks` TSNs, которые были получены после `Cumulative TSN ACK`. Это используется, если есть разрыв в доставленных пакетах. Скажем, части DATA с TSNs `100`, `102`, `103` и `104` доставлены. `Cumulative TSN ACK` был бы `100`, но `Ack Blocks` мог бы сказать отправителю, что ему не нужно передавать `102`, `103` или `104`.

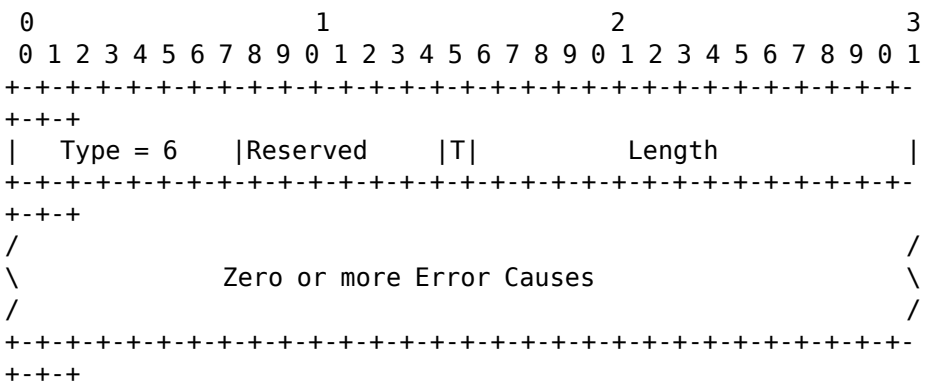
`Duplicate TSN` информирует отправителя о том, что он получил следующие части DATA более одного раза.

HEARTBEAT Chunk



Часть HEARTBEAT используется для утверждения, что удаленный все еще отвечает. Полезно, если вы не отправляете частей DATA и вам нужно поддерживать NAT карту открытой.

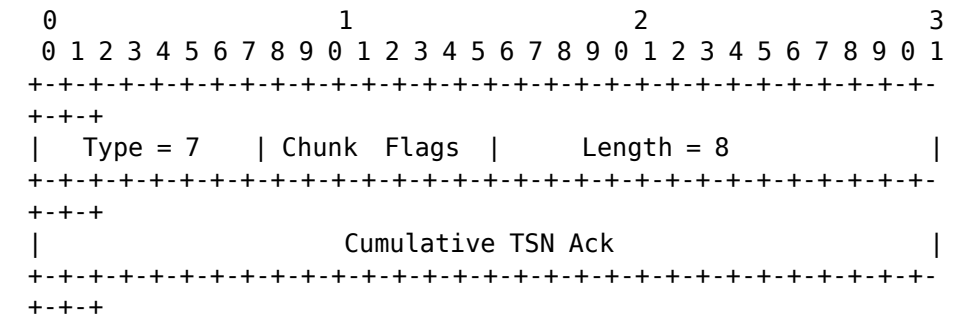
ABORT Chunk



Часть ABORT резко завершает ассоциацию. Используется,

когда одна сторона входит в состояние ошибки. Завершение соединения использует часть SHUTDOWN.

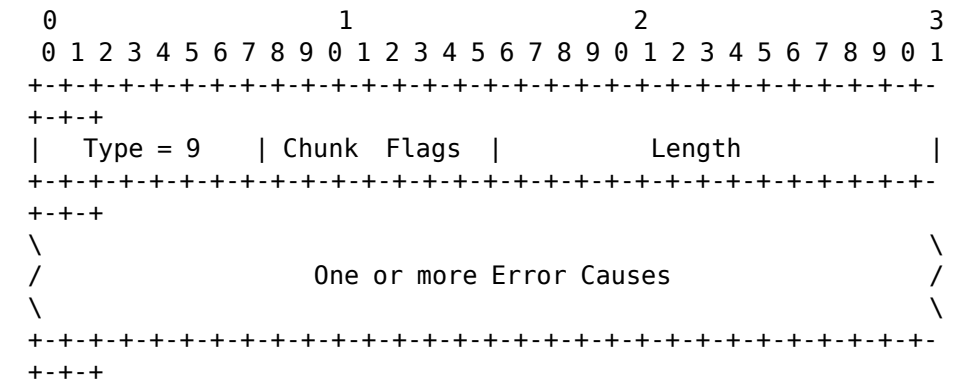
SHUTDOWN Chunk



Часть SHUTDOWN начинает грациозное завершение SCTP-ассоциации. Каждый агент информирует удаленный о последнем TSN, который он отправил. Это гарантирует, что никакие пакеты не будут потеряны. WebRTC не выполняет грациозное завершение SCTP-ассоциации. Вам нужно самостоятельно разорвать каждый канал данных, чтобы обработать его грациозно.

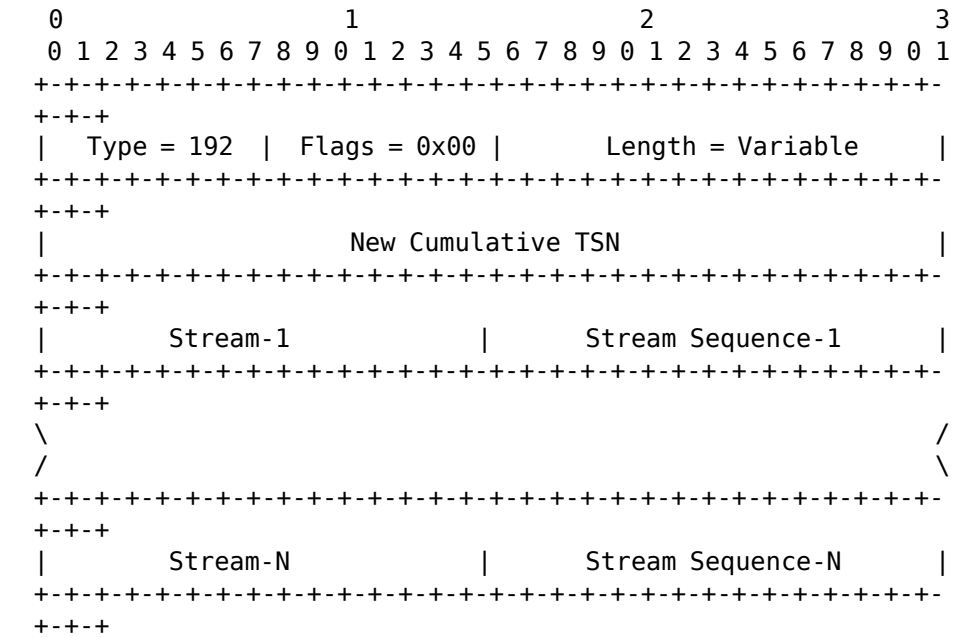
`Cumulative TSN ACK` - это последний TSN, который был отправлен. Каждая сторона знает не отключаться, пока они не получат часть DATA с этим TSN.

ERROR Chunk



Часть ERROR используется для уведомления удаленного SCTP-агента о нефатальной ошибке.

FORWARD TSN Chunk



Часть `FORWARD TSN` перемещает глобальный TSN вперед. SCTP делает это, так что вы можете пропустить некоторые пакеты, которые вам больше не нужны. Скажем, вы отправляете 10 11 12 13 14 15 и эти пакеты действительны только если все они приходят. Эти данные также чувствительны к реальному времени, поэтому, если они приходят поздно, они не полезны.

Если вы теряете 12 и 13, нет причин отправлять 14 и 15! SCTP использует часть `FORWARD TSN` для достижения этого. Он говорит получателю, что 14 и 15 не будут доставлены.

`New Cumulative TSN` - это новый TSN соединения. Все пакеты до этого TSN не будут сохранены.

`Stream` и `Stream Sequence` используются для перехода `Номер последовательности потока` вперед. См. часть DATA для значения этого поля.

State Machine

Это некоторые интересные части состояния машины SCTP. WebRTC не использует все функции состояния машины SCTP, поэтому мы исключили эти части. Мы также упростили некоторые компоненты, чтобы сделать их понятными самостоятельно.

Flow установления соединения

Части `INIT` и `INIT ACK` используются для обмена возможностями и конфигурациями каждого пира. SCTP использует файл cookie во время рукопожатия для проверки пира, с которым он общается. Это гарантирует, что рукопожатие не перехватывается и предотвращает DoS-атаки.

Часть `INIT ACK` содержит файл cookie. Файл cookie затем возвращается его создателю с помощью `COOKIE ECHO`. Если проверка файла cookie успешна, `COOKIE ACK` отправляется, и части DATA готовы к обмену.

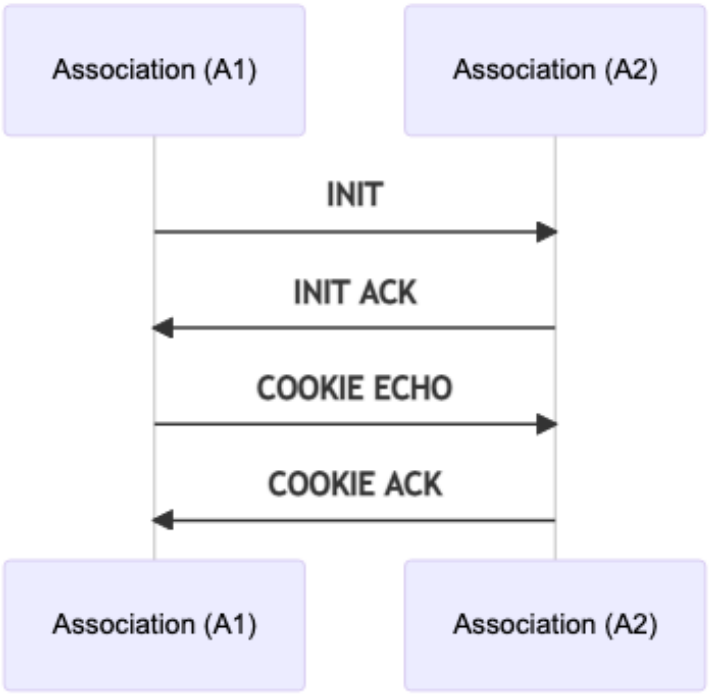


Figure 14: Connection establishment

Flow разрыва соединения

SCTP использует часть `SHUTDOWN`. Когда агент получает часть `SHUTDOWN`, он будет ждать, пока он не получит запрошенный `Cumulative TSN ACK`. Это позволяет пользователю убедиться, что все данные доставлены, даже если соединение потеряно.

Keep-Alive Mechanism

SCTP использует части `HEARTBEAT REQUEST` и `HEARTBEAT ACK` для поддержания соединения. Эти отправляются на настраиваемом интервале. SCTP также выполняет экспоненциальную задержку, если пакет не пришел.

Часть `HEARTBEAT` также содержит временное значение. Это позволяет двум ассоциациям вычислить время поездки между двумя агентами.

Прикладной WebRTC

Теперь, когда вы знаете, как работает WebRTC, пришло время строить с его помощью! Эта глава исследует, что люди создают с помощью WebRTC, и как они это делают. Вы узнаете все интересные вещи, происходящие с WebRTC. Мощь WebRTC дается ценой. Создание производственных сервисов WebRTC - это сложная задача. Эта глава попытается объяснить эти сложности, прежде чем вы с ними столкнетесь.

По случаям использования

Многие думают, что WebRTC - это просто технология для видеоконференций в веб-браузере. Но это гораздо больше! WebRTC используется в широком спектре приложений. Новые случаи использования появляются постоянно. В этой главе мы перечислим некоторые распространенные и то, как WebRTC революционизирует их.

Видеоконференции

Видеоконференции - это изначальный случай использования WebRTC. Протокол содержит несколько необходимых функций, которые не предлагает ни один другой протокол в браузере. Вы могли бы построить систему видеоконференций с WebSocket, и она может работать в оптимальных условиях. Если вы хотите что-то, что можно развернуть в реальных сетевых условиях, WebRTC - лучший выбор.

WebRTC обеспечивает контроль перегрузки и адаптивную скорость передачи для медиа. По мере изменения условий сети пользователи все равно получат наилучший возможный опыт. Разработчикам даже не нужно писать дополнительный код для измерения этих условий.

Участники могут отправлять и получать несколько потоков. Они также могут добавлять и удалять эти потоки в любой момент во время вызова. Кодеки также согласовываются. Вся эта функциональность предоставляется браузером, от разработчика не требуется писать никакого пользовательского кода.

Видеоконференции также выигрывают от каналов данных. Пользователи могут отправлять метаданные или делиться документами. Вы можете создавать несколько потоков и настраивать их, если вам нужна производительность больше, чем надежность.

Broadcasting

В пространстве трансляций появляется все больше новых проектов, использующих WebRTC. Протокол имеет много преимуществ как для издателя, так и для потребителя медиа.

WebRTC в браузере облегчает пользователям публикацию видео. Он устраняет необходимость загрузки нового клиента. Любая платформа, имеющая веб-браузер, может публиковать видео. Издатели могут отправлять несколько треков и изменять или удалять их в любой момент. Это огромное улучшение по сравнению с устаревшими протоколами, которые позволяли иметь только один аудио- или видеотрек на соединение.

WebRTC дает разработчикам больший контроль над компромиссом между задержкой и качеством. Если важнее, чтобы задержка никогда не превышала определенного порога, и вы готовы терпеть некоторые артефакты декодирования, вы можете настроить зрителя воспроизводить медиа сразу после его поступления. С другими протоколами, работающими поверх TCP, это не так просто. В браузере вы можете запросить данные, и все.

Удаленный доступ

Удаленный доступ - это когда вы удаленно получаете доступ к другому компьютеру через WebRTC. Вы можете иметь полный контроль над удаленным хостом или, возможно, только над одним приложением. Это отлично подходит для выполнения вычислительно затратных задач, когда локальное оборудование не справляется. Например, для запуска новой видеоигры или программы САПР. WebRTC смог революционизировать это пространство тремя способами.

WebRTC можно использовать для удаленного доступа к хосту, который не имеет глобальной маршрутизации. Благодаря обходу

NAT вы можете получить доступ к компьютеру, доступному только через STUN. Это отлично для безопасности и конфиденциальности. Вашим пользователям не нужно направлять видео через входной поток или “jump box”. Обход NAT также облегчает развертывание. Вам не нужно беспокоиться о переадресации портов или настройке статического IP заранее.

Каналы данных также очень мощны в этом сценарии. Они могут быть настроены так, чтобы принимать только самые последние данные. При использовании TCP вы рискуете столкнуться с блокировкой головы очереди. Старый щелчок мыши или нажатие клавиши может прийти поздно и заблокировать последующие. Каналы данных WebRTC разработаны для работы с этим и могут быть настроены так, чтобы не пересылать потерянные пакеты. Вы также можете измерять обратное давление и убедиться, что не отправляете больше данных, чем поддерживает ваша сеть.

Доступность WebRTC в браузере стала огромным улучшением качества жизни. Вам не нужно загружать проприетарный клиент для начала сеанса. Все больше клиентов поставляются с встроенным WebRTC, smart TV теперь получают полноценные веб-браузеры.

Обмен файлами и обход цензуры

Обмен файлами и обход цензуры - это принципиально разные проблемы. Однако WebRTC решает для них одни и те же задачи, делая их легкодоступными и сложными для блокировки.

Первая проблема, которую решает WebRTC - получение клиента. Чтобы присоединиться к сети обмена файлами, вам нужно загрузить клиент. Даже если сеть распределенная, вам все равно сначала нужно получить клиент. В ограниченной сети загрузка часто блокируется. Даже если вы сможете загрузить его, пользователь может не суметь установить и запустить клиент. WebRTC уже доступен в каждом веб-браузере, что делает его легкодоступным.

Вторая проблема, которую решает WebRTC - блокировка трафика. Если вы используете протокол, предназначенный только для обмена файлами или обхода цензуры, его гораздо легче заблокировать. Поскольку WebRTC - это универсальный протокол, его блокировка затронет всех. Блокировка WebRTC может помешать другим пользователям сети присоединяться к видеоконференциям.

Интернет вещей

Интернет вещей (IoT) охватывает несколько различных вариантов использования. Для многих это означает сетевые подключенные камеры безопасности. Используя WebRTC, вы можете транслировать видео другому WebRTC-пиру, например, на телефон или в браузер. Другой вариант использования - подключение устройств и обмен данными с датчиков. Вы можете иметь два устройства в вашей локальной сети, обменивающиеся данными о климате, шуме или освещении.

У WebRTC есть огромное преимущество с точки зрения конфиденциальности по сравнению с устаревшими протоколами потоковой передачи видео. Поскольку WebRTC поддерживает одноранговое подключение, камера может отправлять видео напрямую в ваш браузер. Нет причин отправлять ваше видео на сторонний сервер. Даже когда видео зашифровано, злоумышленник может делать предположения по метаданным вызова.

Совместимость - еще одно преимущество для пространства IoT. WebRTC доступен на многих языках: C#, C++, C, Go, Java, Python, Rust и TypeScript. Это означает, что вы можете использовать язык, который лучше всего подходит вам. Вам также не нужно обращаться к проприетарным протоколам или форматам, чтобы подключить своих клиентов.

Мостовая передача медиа-протоколов

У вас есть существующее оборудование и программное обеспечение, производящее видео, но вы пока не можете его обновить. Ожидание от пользователей загрузки проприетарного клиента для просмотра видео раздражает. Ответ - запустить WebRTC-мост. Мост переводит между двумя протоколами, чтобы пользователи могли использовать браузер с вашей устаревшей настройкой.

Многие форматы, с которыми разработчики создают мосты, используют те же протоколы, что и WebRTC. SIP часто предоставляется через WebRTC и позволяет пользователям совершать телефонные звонки из браузера. RTSP используется во многих устаревших камерах безопасности. Они оба используют одни и те же базовые протоколы (RTP и SDP), поэтому создание моста вычислительно недорого. Мост просто должен добавлять или удалять специфические для WebRTC вещи.

Мостовая передача протоколов данных

Веб-браузер может работать только с ограниченным набором протоколов. Вы можете использовать HTTP, WebSockets, WebRTC и QUIC. Если вы хотите подключиться к чему-либо еще, вам нужен протокольный мост. Протокольный мост - это сервер, который преобразует внешний трафик в то, что может получить браузер. Популярный пример - использование SSH из браузера для доступа к серверу. Каналы данных WebRTC имеют два преимущества перед конкурентами.

Каналы данных WebRTC позволяют ненадежную и неупорядоченную доставку. В случаях, когда критична низкая задержка, это необходимо. Вы не хотите, чтобы новые данные блокировались старыми - это известно как блокировка головы очереди. Представьте, что вы играете в многопользовательский шутер от первого лица. Действительно ли вам важно, где был игрок две секунды назад? Если эти данные не пришли вовремя, нет смысла продолжать пытаться их отправлять. Ненадежная и неупорядоченная доставка позволяет использовать данные сразу после их поступления.

Каналы данных также обеспечивают обратное давление. Это показывает, отправляете ли вы данные быстрее, чем может поддерживать ваше соединение. Затем у вас есть два варианта действий, когда это происходит. Канал данных может быть настроен либо на буферизацию и доставку данных с задержкой, либо на сброс данных, не поступивших в реальном времени.

Телеоперация

Телеоперация - это акт удаленного управления устройством через каналы данных WebRTC и отправки видео через RTP. Сегодня разработчики удаленно управляют автомобилями через WebRTC! Это используется для управления роботами на строительных площадках и доставки посылок. Использование WebRTC для этих задач имеет смысл по двум причинам.

Повсеместность WebRTC облегчает пользователям управление. Пользователю нужен только веб-браузер и устройство ввода. Браузеры даже поддерживают ввод с джойстиков и геймпадов. WebRTC полностью устраняет необходимость установки дополнительного клиента на устройство пользователя.

Распределенная CDN

Распределенные CDN - это подмножество обмена файлами. Распространяемые файлы настраиваются оператором CDN. Когда пользователи присоединяются к сети CDN, они могут загружать и делиться разрешенными файлами. Пользователи получают все те же преимущества, что и при обмене файлами.

Такие CDN отлично работают, когда вы находитесь в офисе с плохим внешним подключением, но отличным подключением локальной сети. Один пользователь может загрузить видео, а затем поделиться им со всеми остальными. Поскольку все не пытаются получить один и тот же файл через внешнюю сеть, передача завершится быстрее.

Топологии WebRTC

WebRTC - это протокол для подключения двух агентов, так как же разработчики подключают сотни людей одновременно? Существует несколько различных способов сделать это, и у каждого есть свои плюсы и минусы. Эти решения в основном делятся на две категории: одноранговые (Peer-to-Peer) или клиент-серверные. Гибкость WebRTC позволяет нам создавать и те, и другие.

Один-к-Одному

Один-к-Одному - это первый тип подключения, который вы будете использовать с WebRTC. Вы напрямую подключаете два WebRTC-агента, и они могут отправлять двунаправленные медиа и данные. Подключение выглядит так.

Figure 15: Один-к-Одному

Полная сетка

Полная сетка - это решение, если вы хотите создать конференц-связь или многопользовательскую игру. В этой топологии каждый пользователь устанавливает прямое подключение с каждым другим пользователем. Это позволяет вам создать приложение, но имеет некоторые недостатки.

В топологии полной сетки каждый пользователь подключен напрямую. Это означает, что вам нужно кодировать и загружать видео независимо для каждого участника вызова. Сетевые условия между каждым подключением будут разными, поэтому вы не можете использовать одно и то же видео повторно. Обработка ошибок также сложна в таких развертываниях. Вам нужно тщательно оценивать, потеряли ли вы полное подключение или только подключение с одним удаленным пиром.

Из-за этих соображений полная сетка лучше всего подходит для небольших групп. Для чего-либо большего лучше использовать клиент-серверную топологию.

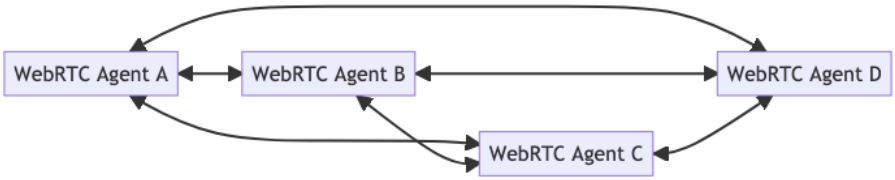


Figure 16: Полная сетка

Гибридная сетка

Гибридная сетка - это альтернатива полной сетке, которая может смягчить некоторые ее проблемы. В гибридной сетке подключения устанавливаются не между всеми пользователями. Вместо этого медиа ретранслируется через пиров в сети. Это означает, что создателю медиа не нужно использовать столько пропускной способности для распространения медиа.

Это имеет некоторые недостатки. В этой настройке оригинальный создатель медиа не знает, кому отправляется его видео, и не знает, было ли оно успешно доставлено. Кроме того, с каждым переходом в вашей гибридной сетке будет увеличиваться задержка.

Блок селективной пересылки

Блок селективной пересылки (SFU - Selective Forwarding Unit) также решает проблемы полной сетки, но совершенно по-другому. SFU реализует клиент-серверную топологию вместо одноранговой. Каждый WebRTC-пир подключается к SFU и загружает свои медиа. SFU затем пересылает эти медиа каждому подключенному клиенту.

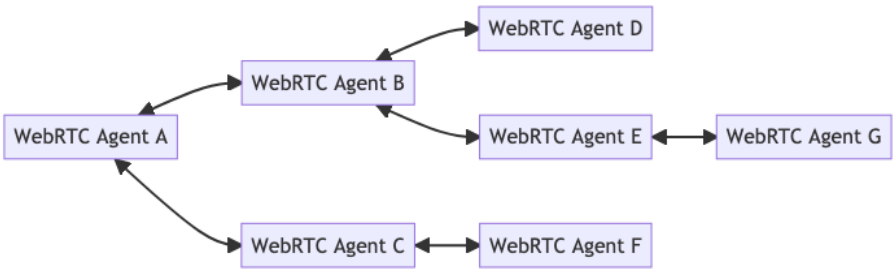


Figure 17: Гибридная сетка

С SFU каждому WebRTC-агенту нужно закодировать и загрузить свое видео только один раз. Бремя распространения его всем зрителям ложится на SFU. Подключение с SFU также гораздо проще, чем одноранговое. Вы можете запустить SFU по глобально маршрутизируемому адресу, что значительно упрощает подключение клиентов. Вам не нужно беспокоиться о сопоставлениях NAT. Вам все еще нужно убедиться, что ваш SFU доступен через TCP (либо через ICE-TCP, либо через TURN).

Создание простого SFU можно выполнить за выходные. Создание хорошего SFU, который может обрабатывать все типы клиентов, - это бесконечная задача. Настройка контроля перегрузки, исправление ошибок и оптимизация производительности - это непрерывный процесс.

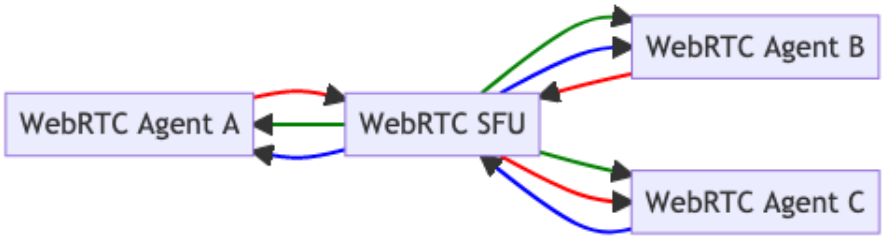


Figure 18: Блок селективной пересылки

MCU

MCU (Multi-point Conferencing Unit) - это клиент-серверная топология, похожая на SFU, но с композицией выходных потоков. Вместо распространения исходящих медиа без изменений она перекодирует их в один поток.

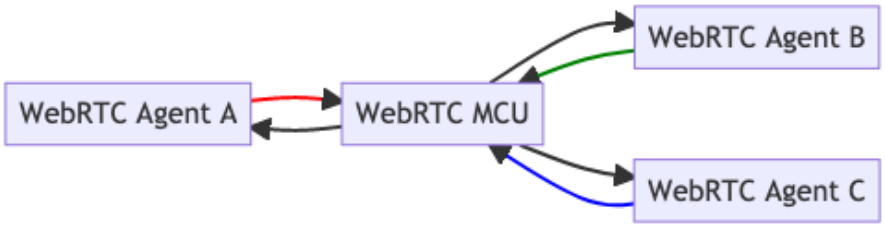


Figure 19: Многоточечный конференц-блок

Отладка

Отладка WebRTC может быть сложной задачей. Существует много подвижных частей, и они могут ломаться независимо. Если вы не будете осторожны, вы можете потерять недели, глядя не на те вещи. Когда вы наконец найдете сломанную часть, вам придется кое-что изучить, чтобы понять почему.

Эта глава поможет вам настроиться на отладку WebRTC. Она покажет, как разбить проблему на составляющие. После того, как мы поймем проблему, мы быстро обзорно рассмотрим популярные инструменты отладки.

Изолируйте проблему

При отладке вам нужно изолировать источник проблемы. Начните с самого начала...

Проверка STUN-сервера с помощью netcat:

1. Подготовьте **20-байтный** пакет binding-запроса:

   `00 01 00 00 21 12 a4 42 54 45 53 54 54 45 53 54 54 45 53 54`

   Интерпретация:

   - `00 01` - тип сообщения.
   - `00 00` - длина раздела данных.
   - `21 12 a4 42` - магический cookie.
   - `54 45 53 54 54 45 53 54 54 45 53 54` (декодируется в ASCII как TESTTESTTEST) - 12-байтный идентификатор транзакции.

2. Отправьте запрос и ждите **32-байтный** ответ:

   ```
   echo -ne '\x00\x01\x00\x00\x21\x12\xa4\x42\x54\x45\x53\x54\x54\x45\x53\x54\x54\x4
   u stun.l.google.com 19302 | xxd
   ```

Интерпретация:

- `01 01` - тип сообщения
- `00 0c` - длина раздела данных, декодируется в 12 в десятичной системе
- `21 12 a4 42` - магический cookie
- `54 45 53 54 54 45 53 54 54 45 53 54` (декодируется в ASCII как `TESTTESTTEST`) - 12-байтный идентификатор транзакции.
- `00 20 00 08 00 01 6f 32 7f 36 de 89` - 12-байтные данные, интерпретация:
  - `00 20` - тип: `XOR-MAPPED-ADDRESS`
  - `00 08` - длина раздела значений, декодируется в 8 в десятичной системе
  - `00 01 6f 32 7f 36 de 89` - значение данных, интерпретация:
    - `00 01` - тип адреса (IPv4)
    - `6f 32` - XOR-mapped порт
    - `7f 36 de 89` - XOR-mapped IP-адрес

Декодирование XOR-mapped раздела громоздко, но мы можем заставить STUN-сервер выполнить фиктивное XOR-маскирование, предоставив (недопустимый) фиктивный магический cookie, установленный в `00 00 00 00`:

```
echo -ne '\x00\x01\x00\x00\x00\x00\x00\x00\x54\x45\x53\x54\x54\x45\x53\x54\x54\x45\x5
u stun.l.google.com 19302 | xxd
```

XOR с фиктивным магическим cookie идемпотентен, поэтому порт и адрес будут в открытом виде в ответе. Это не сработает во всех ситуациях, потому что некоторые маршрутизаторы манипулируют проходящими пакетами, жульничая с IP-адресом. Если мы посмотрим на возвращаемое значение данных (последние восемь байт):

- `00 01 4e 20 5e 24 7a cb` - значение данных, интерпретация:
  - `00 01` - тип адреса (IPv4)
  - `4e 20` - mapped порт, который декодируется в 20000 в десятичной системе
  - `5e 24 7a cb` - IP-адрес, который декодируется в `94.36.122.203` в точечно-десятичной нотации.

Сбой безопасности

Сбой медиа

Сбой данных

Инструменты профессионала

netcat (nc)

netcat - это утилита командной строки для чтения и записи сетевых подключений с использованием TCP или UDP. Обычно доступна как команда `nc`.

tcpdump

tcpdump - анализатор сетевых пакетов из командной строки.

Распространенные команды: - Захватить UDP-пакеты к и от порта 19302, вывести шестнадцатеричный дамп содержимого пакета:

`` `sudo tcpdump 'udp port 19302' -xx` ``

- То же самое, но сохранить пакеты в файл PCAP (packet capture) для последующего осмотра:

  ```
  sudo tcpdump 'udp port 19302' -w stun.pcap
  ```

  Файл PCAP можно открыть в приложении Wireshark: `wireshark stun.pcap`

Wireshark

Wireshark - широко используемый анализатор сетевых протоколов.

Инструменты WebRTC в браузерах

Браузеры имеют встроенные инструменты, которые можно использовать для проверки устанавливаемых подключений. Chrome имеет `chrome://webrtc-internals` и `chrome://webrtc-logs`. Firefox имеет `about:webrtc`.

Задержка

Задержка от конца до конца не является простой суммой задержек каждого компонента.

Хотя теоретически можно измерить задержку компонентов конвейера передачи живого видео по отдельности, а затем сложить их, на практике по крайней мере некоторые компоненты

будут либо недоступны для инструментирования, либо будут давать значительно отличающиеся результаты при измерении вне конвейера. Переменные глубины очередей между этапами конвейера, топология сети и изменения экспозиции камеры - лишь несколько примеров компонентов, влияющих на задержку от конца до конца.

Внутренняя задержка каждого компонента в системе потоковой передачи может меняться и влиять на последующие компоненты. Даже содержимое захваченного видео влияет на задержку. Например, для высокочастотных элементов, таких как ветви деревьев, требуется гораздо больше битов по сравнению с низкочастотным чистым голубым небом. Камера с включенной автоэкспозицией может захватывать кадр гораздо дольше ожидаемых 33 миллисекунд, даже если частота съемки установлена на 30 кадров в секунду. Передача по сети, особенно сотовой, также очень динамична из-за меняющегося спроса. Больше пользователей означает больше трафика в эфире. Ваше физическое местоположение (известные зоны слабого сигнала) и множество других факторов увеличивают потери пакетов и задержку.

Ручное измерение задержки от конца до конца

Когда мы говорим о задержке от конца до конца, мы подразумеваем время между происходящим событием и его наблюдением, то есть появлением видеокадров на экране.

```
EndToEndLatency = T(observe) - T(happen)
```

Наивный подход - зафиксировать время происходящего события и вычесть из времени наблюдения. Однако при уточнении до миллисекунд синхронизация времени становится проблемой. Попытки синхронизировать часы в распределенных системах в основном бесполезны, даже небольшая ошибка синхронизации времени приводит к недостоверному измерению задержки.

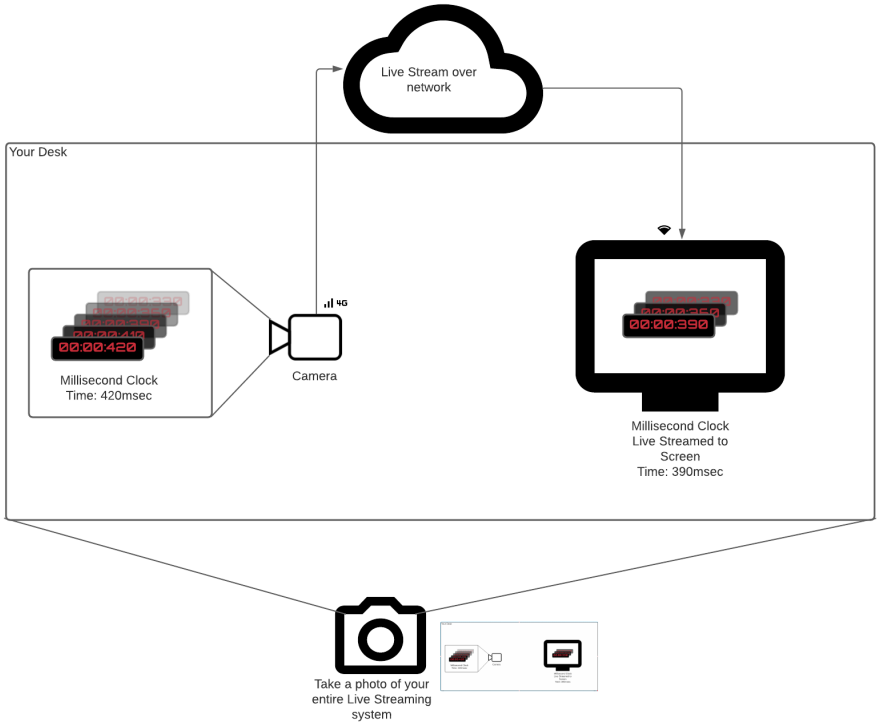
Простой обходной путь для проблем синхронизации часов - использовать один и тот же clock. Поместите отправителя и получателя в одну систему отсчета.

Представьте, что у вас есть тикающие миллисекундные часы или любой другой источник событий. Вы хотите измерить задержку в системе, которая транслирует часы на удаленный экран, наведя на них камеру. Очевидный способ измерить время между тиканием миллисекундного таймера (`Thappen`) и появлением видеокадров часов на экране (`Tobserve`) следующий:

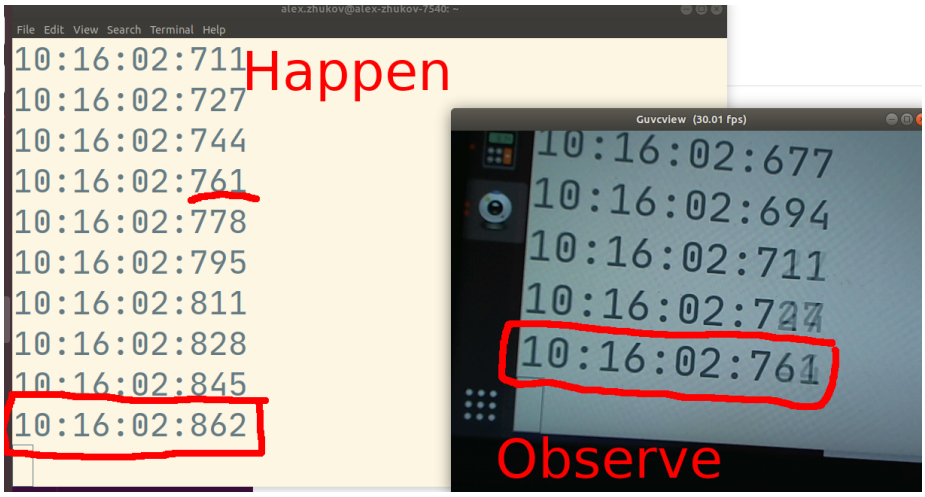
- Наведите камеру на миллисекундные часы.

- Отправьте видеокадры получателю, находящемуся в том же физическом месте.
- Сфотографируйте (используйте телефон) миллисекундный таймер и полученное видео на экране.
- Вычтите два времени.

Это самое правдивое измерение задержки от конца до конца. Оно учитывает задержки всех компонентов (камера, кодировщик, сеть, декодер) и не полагается на синхронизацию часов.



.



На фото выше измеренная задержка от конца до конца составляет 101 мс. Событие происходит прямо сейчас в 10:16:02.862, но наблюдатель системы потоковой передачи видит 10:16:02.761.

Автоматическое измерение задержки от конца до конца

На момент написания (май 2021 года) стандарт WebRTC для задержки от конца до конца активно обсуждается. Firefox реализовал набор API для создания автоматических измерений задержки поверх стандартных WebRTC API. Однако в этом параграфе мы обсудим наиболее совместимый способ автоматического измерения задержки.

Время кругового обхода вкратце: я отправляю вам свое время `tR1`, когда я получаю обратно мой `tR1` во время `tR2`, я знаю, что время кругового обхода равно `tR2 - tR1`.

При наличии канала связи между отправителем и получателем (например, DataChannel) получатель может смоделировать монотонные часы отправителя, выполнив следующие шаги:

1. Во время `tR1` получатель отправляет сообщение со своей локальной меткой времени монотонных часов.
2. Когда оно принимается отправителем с локальным временем `tS1`, отправитель отвечает копией `tR1`, а также своим `tS1` и временем видеодорожки отправителя `tSV1`.
3. Во время `tR2` на стороне получателя время кругового обхода вычисляется путем вычитания времени отправки и получения сообщения: `RTT = tR2 - tR1`.
4. Время кругового обхода `RTT` вместе с локальной меткой

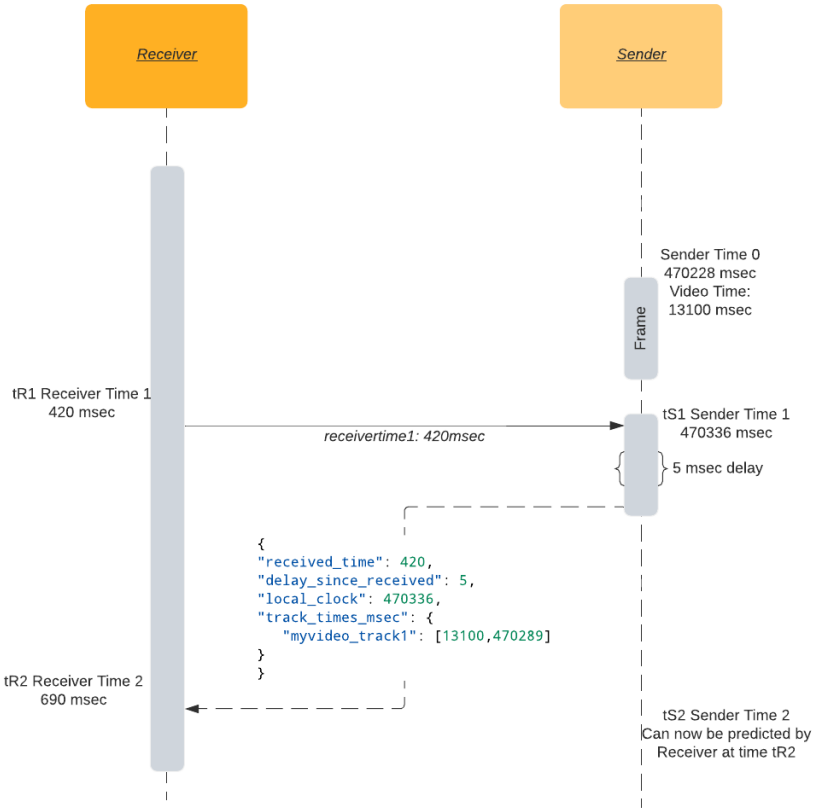


Figure 20: Измерение задержки в стиле NTP

времени отправителя tS1 достаточно для создания оценки монотонных часов отправителя. Текущее время на отправителе во время tR2 будет равно tS1 плюс половина времени кругового обхода.

5. Локальная метка времени отправителя tS1, сопоставленная с меткой времени видеодорожки tSV1 вместе со временем кругового обхода RTT, достаточна для синхронизации времени видеодорожки получателя со временем видеодорожки отправителя.

Теперь, когда мы знаем, сколько времени прошло с момента последнего известного времени видеокадра отправителя tSV1, мы можем приблизительно оценить задержку, вычтя время текущего отображаемого видеокадра (actual_video_time) из ожидаемого времени:

```
expected_video_time = tSV1 + time_since(tSV1)
latency = expected_video_time - actual_video_time
```

Недостаток этого метода заключается в том, что он не включает внутреннюю задержку камеры. Большинство видеосистем считают метку времени захвата кадра временем доставки кадра с камеры в основную память, что происходит через несколько мгновений после фактического происходящего события.

Пример оценки задержки Образец реализации открывает канал данных latency на получателе и периодически отправляет метки времени монотонного таймера получателя отправителю. Отправитель отвечает JSON-сообщением, и получатель вычисляет задержку на основе сообщения.

```
{
    "received_time": 64714,       // Метка времени, отправленная получателем, отража
    "delay_since_received": 46,   // Время, прошедшее с момента последнего полученно
    "local_clock": 1597366470336, // Текущее время монотонных часов отправителя.
    "track_times_msec": {
        "myvideo_track1": [
            13100,        // Метка времени RTP видеокадра (в миллисекундах).
            1597366470289 // Метка времени монотонных часов видеокадра.
        ]
    }
}
```

Откройте канал данных на получателе:

```
dataChannel = peerConnection.createDataChannel('latency');
```

Отправляйте время получателя tR1 периодически. В этом примере используется 2 секунды без особой причины:

```
setInterval(() => {
    let tR1 = Math.trunc(performance.now());
    dataChannel.send("" + tR1);
}, 2000);
```

Обработайте входящее сообщение от получателя на отправителе:

```
// Предполагаем, что event.data - строка вида "1234567".
tR1 = event.data
now = Math.trunc(performance.now());
tSV1 = 42000; // Текущая метка времени RTP кадра, преобразованная в миллисекундный м
tS1 = 1597366470289; // Текущая метка времени монотонных часов кадра.
msg = {
  "received_time": tR1,
  "delay_since_received": 0,
  "local_clock": now,
  "track_times_msec": {
    "myvideo_track1": [tSV1, tS1]
  }
}
dataChannel.send(JSON.stringify(msg));
```

Обработайте входящее сообщение от отправителя и выведите оценку задержки в `console`:

```
let tR2 = performance.now();
let fromSender = JSON.parse(event.data);
let tR1 = fromSender['received_time'];
let delay = fromSender['delay_since_received']; // Сколько времени прошло между полу
let senderTimeFromResponse = fromSender['local_clock'];
let rtt = tR2 - delay - tR1;
let networkLatency = rtt / 2;
let senderTime = (senderTimeFromResponse + delay + networkLatency);
VIDEO.requestVideoFrameCallback((now, framemeta) => {
    // Оценить текущее время отправителя.
    let delaySinceVideoCallbackRequested = now - tR2;
    senderTime += delaySinceVideoCallbackRequested;
    let [tSV1, tS1] = Object.entries(fromSender['track_times_msec'])[0][1]
    let timeSinceLastKnownFrame = senderTime - tS1;
    let expectedVideoTimeMsec = tSV1 + timeSinceLastKnownFrame;
    let actualVideoTimeMsec = Math.trunc(framemeta.rtpTimestamp / 90); // Преобразов
    let latency = expectedVideoTimeMsec - actualVideoTimeMsec;
    console.log('latency', latency, 'msec');
});
```

Фактическое время видео в браузере

`<video>.requestVideoFrameCallback()` позволяет веб-

авторам получать уведомления о представлении кадра для компоновки.

До недавнего времени (до мая 2020 года) было практически невозможно надежно получить метку времени текущего отображаемого видеокадра в браузерах. Существовали обходные методы на основе `video.currentTime`, но они были не особенно точными.

Разработчики браузеров Chrome и Mozilla поддержали введение нового стандарта W3C, `HTMLVideoElement.requestVideoFrameCallback()`, который добавляет API-обратный вызов для доступа к текущему времени видеокадра.

Хотя дополнение кажется тривиальным, оно позволило создать множество сложных медиаприложений в Интернете, требующих синхронизации аудио и видео.

Специально для WebRTC обратный вызов будет включать поле `rtpTimestamp`, метку времени RTP, связанную с текущим видеокадром. Это должно быть присутствующим для приложений WebRTC, но отсутствовать в других случаях.

Советы по отладке задержки

Поскольку отладка, вероятно, повлияет на измеряемую задержку, общее правило - упростить вашу настройку до наименьшего возможного размера, который все еще может воспроизвести проблему. Чем больше компонентов вы сможете удалить, тем легче будет определить, какой компонент вызывает проблему с задержкой.

Задержка камеры В зависимости от настроек камеры ее задержка может варьироваться. Проверьте настройки автоэкспозиции, автофокуса и автоматического баланса белого. Все “автоматические” функции веб-камер занимают дополнительное время для анализа захваченного изображения перед его передачей в стек WebRTC.

Если вы используете Linux, вы можете использовать инструмент командной строки `v4l2-ctl` для управления настройками камеры:

```
# Отключить автофокус:
v4l2-ctl -d /dev/video0 -c focus_auto=0
# Установить фокус на бесконечность:
v4l2-ctl -d /dev/video0 -c focus_absolute=0
```

Вы также можете использовать графический инструмент `guvcview` для быстрой проверки и настройки параметров камеры.

Задержка кодировщика Большинство современных кодировщиков будут буферизовать некоторые кадры перед выводом закодированного. Их первый приоритет - баланс между качеством создаваемой картинки и битрейтом. Многопроходное кодирование - крайний пример пренебрежения кодировщика к выходной задержке. Во время первого прохода кодировщик полностью поглощает все видео и только после этого начинает выводить кадры.

Однако с правильной настройкой люди достигали субкадровых задержек. Убедитесь, что ваш кодировщик не использует чрезмерное количество эталонных кадров и не полагается на B-кадры. Настройки задержки каждого кодека различаются, но для x264 мы рекомендуем использовать `tune=zerolatency` и `profile=baseline` для минимальной задержки вывода кадров.

Сетевая задержка Сетевой задержкой вы можете arguably сделать меньше всего, кроме как обновить сетевое подключение. Сетевая задержка очень похожа на погоду - вы не можете остановить дождь, но можете посмотреть прогноз и взять зонт. WebRTC измеряет сетевые условия с точностью до миллисекунд.

Важные метрики: - Время кругового обхода. - Потеря пакетов и повторные передачи пакетов.

Время кругового обхода

Стек WebRTC имеет встроенный механизм измерения времени кругового обхода сети (RTT) механизм. Достаточно хорошее приближение задержки - половина RTT. Предполагается, что отправка и получение пакета занимают одинаковое время, что не всегда так. RTT устанавливает нижнюю границу задержки от конца до конца. Ваши видеокадры не могут достичь получателя быстрее, чем `RTT/2`, независимо от того, насколько оптимизирован ваш конвейер от камеры до кодировщика.

Встроенный механизм RTT основан на специальных RTCP-пакетах, называемых отчетами отправителя/получателя. Отправитель отправляет свое показание времени получателю, получатель в свою очередь отражает тот же временной штамп обратно отправителю. Таким образом, отправитель знает, сколько времени заняла передача пакета получателю и обратно. Обратитесь к главе Отчеты отправителя/получателя для более подробной информации об измерении RTT.

Потеря пакетов и повторные передачи

Как RTP, так и RTCP - это протоколы на основе UDP, которые не гарантируют упорядочивания, успешной доставки или отсутствия дублирования. Все вышеперечисленное может и происходит в реальных приложениях WebRTC.

Несложная реализация декодера ожидает, что все пакеты кадра будут доставлены, чтобы декодер мог успешно восстановить изображение. При наличии потери пакетов могут появиться артефакты декодирования, если теряются пакеты P-кадра. Если теряются пакеты I-кадра, то все зависимые кадры либо получат серьезные артефакты, либо вообще не будут декодированы. Скорее всего, это приведет к “замораживанию” видео на мгновение.

Чтобы избежать (точнее, попытаться избежать) замораживания видео или артефактов декодирования, WebRTC использует сообщения отрицательного подтверждения (NACK). Когда получатель не получает ожидаемый RTP-пакет, он возвращает сообщение NACK, чтобы сообщить отправителю отправить отсутствующий пакет снова. Получатель *ждет* повторной передачи пакета. Такие повторные передачи вызывают увеличение задержки. Количество отправленных и полученных пакетов NACK записывается во встроенных статистических полях WebRTC outbound stream nackCount и inbound stream nackCount.

Вы можете увидеть красивые графики входящего и исходящего `nackCount` на странице внутренних данных WebRTC. Если вы видите, что `nackCount` увеличивается, это означает, что сеть испытывает высокую потерю пакетов, и стек WebRTC делает все возможное, чтобы создать гладкое видео/аудио-взаимодействие, несмотря на это.

Когда потеря пакетов настолько высока, что декодер не может создать изображение, или последующие зависимые изображения, как в случае полностью потерянного I-кадра, все будущие P-кадры не будут декодированы. Получатель попытается смягчить это, отправив специальное сообщение Picture Loss Indication (PLI). Как только отправитель получает `PLI`, он создаст новый I-кадр, чтобы помочь декодеру получателя. I-кадры обычно больше по размеру, чем P-кадры. Это увеличивает количество пакетов, которые необходимо передать. Как и с сообщениями NACK, получателю придется ждать нового I-кадра, что введет дополнительную задержку.

Следите за `pliCount` на странице внутренних данных WebRTC. Если он увеличивается, настройте кодировщик для создания меньшего количества пакетов или включите более устойчивый

к ошибкам режим.

Задержка на стороне получателя Задержка будет зависеть от пакетов, приходящих не по порядку. Если нижняя часть пакета изображения придет раньше верхней, вам придется ждать верхнюю часть перед декодированием. Это подробно объясняется в главе Решение джиттера.

Вы также можете обратиться к встроенной метрике jitterBuffer-Delay, чтобы увидеть, как долго кадр удерживался в приемном буфере в ожидании всех его пакетов, прежде чем был передан декодеру.

История

При изучении WebRTC разработчики часто чувствуют разочарование из-за сложности. Они видят функции WebRTC, не относящиеся к их текущему проекту, и хотят, чтобы WebRTC был проще. Проблема заключается в том, что у каждого свой набор вариантов использования. Коммуникации в реальном времени имеют богатую историю с множеством людей, создающих различные вещи.

Эта глава содержит интервью с авторами протоколов, составляющих WebRTC. Она дает представление о проектных решениях при создании каждого протокола и заканчивается интервью о самом WebRTC. Если вы понимаете намерения и проектные решения программного обеспечения, вы можете создавать более эффективные системы.

RTP

RTP и RTCP - это протокол, который обрабатывает весь медиа-транспорт для WebRTC. Он был определен в RFC 1889 в январе 1996 года. Нам очень повезло, что один из авторов Рон Фредерик рассказывает о нем сам. Рон недавно загрузил Network Video tool на GitHub, проект, который повлиял на RTP.

Своими словами

В октябре 1992 года я начал экспериментировать с платой захвата кадров Sun VideoPix, с идеей написать сетевой инструмент видеоконференцсвязи на основе IP-многоадресной рассылки. Он был смоделирован по образцу “vat” - инструмента аудиоконференцсвязи, разработанного в LBL, в котором

использовался аналогичный легковесный протокол сеанса для пользователей, присоединяющихся к конференциям, где вы просто отправляете данные в определенную многоадресную группу и следите за трафиком от других членов группы.

В программе для того, чтобы она была действительно успешной, ей нужно было сжимать видеоданные перед отправкой в сеть. Моя цель состояла в том, чтобы создать приемлемый по виду поток данных, который мог бы поместиться в 128 кбит/с, или доступную полосу пропускания стандартной домашней линии ISDN. Я также надеялся создать что-то, что все еще можно было бы смотреть, которое помещалось бы в половину этой полосы пропускания. Это означало, что мне нужно было приблизительно в 20 раз сжимать видео для конкретного размера изображения и частоты кадров, с которыми я работал. Я смог достичь этого сжатия и подал заявку на патент на методы, которые я использовал, позже выданный патент US5485212A: Software video compression for teleconferencing.

В начале ноября 1992 года я выпустил инструмент видеоконференцсвязи “nv” (в бинарном формате) в интернет-сообщество. После некоторых начальных тестов он использовался для видеоконференцсвязи частей ноябрьского Интернет-Инженерного Задания по задачам (IETF) по всему миру. Примерно 200 подсетей в 15 странах были способны принимать эту трансляцию, и примерно 50-100 человек получали видео, используя “nv” в течение недели.

В течение следующих нескольких месяцев три других встречи и некоторые меньшие встречи использовали “nv” для трансляции в интернет на большой шкале, включая Австралийскую Сеть-Встречу, Рабочую встречу по аудио- и видеоконференцсвязи CU-SeeMe и встречу MultiG о распределенных виртуальных реальностях в Швеции.

Выпуск исходного кода “nv” последовал в феврале 1993 года, и в марте я выпустил версию инструмента, в которой я ввел новую схему сжатия на основе вейвлетов. В мае 1993 года я добавил поддержку цветного видео.

Сетевой протокол, используемый для “nv” и других интернет-инструментов конференцсвязи, стал основой Реалтайм-Транспортного Протокола (RTP), стандартизированного через Интернет-Инженерный Задание по задачам (IETF), первоначально опубликованного в RFCs 1889-1890 и затем пересмотренного в RFCs 3550-3551 вместе с различными другими RFCs, которые охватывали профили для переноса конкретных форматов аудио и видео.

В течение следующих нескольких лет работа продолжалась

над “nv”, портируя инструмент на множество дополнительных аппаратных платформ и устройства захвата видео. Он продолжал использоваться как один из основных инструментов для трансляции конференций в интернет в то время, включая был выбран NASA для трансляции онлайн-покрытия миссий шаттла.

В 1994 году я добавил поддержку в “nv” для поддержки алгоритмов сжатия видео, разработанных другими, включая некоторые схемы аппаратного сжатия, такие как формат CellB, поддерживаемый картой захвата видео SunVideo. Это также позволило “nv” отправлять видео в формате CUSeeMe, отправлять видео пользователям, запустившим CUSeeMe на Mac и PC.

Последняя публично выпущенная версия “nv” была версией 3.3beta, выпущенной в июле 1994 года. Я работал над выпуском “4.0alpha”, который был предназначен для миграции “nv” на версию 2 RTP-протокола, но эта работа никогда не была завершена из-за моего перехода к другим проектам. Копия 4.0 альфа-кода включена в архив Network Video tool для полноты, но он не завершен и имеет известные проблемы, особенно в неполной поддержке RTPv2.

Фреймворк, предоставленный в “nv”, позже стал основой для видеоконференцсвязи в проекте “Jupiter multi-media MOO” в Xerox PARC, который, в конечном итоге, стал основой для компании-разработчика “PlaceWare”, позже приобретенной Microsoft. Он также использовался как основа для нескольких проектов аппаратного видеоконференцсвязи, которые позволяли отправлять полное качество NTSC видео по высокоскоростным Ethernet и сетям ATM. Я также позже использовал некоторый из этого кода как основу для “Mediastore”, который был сетевым сервисом записи и воспроизведения видео.

Помните ли вы мотивации/идеи других участников черновика?

Мы все были исследователями, работающими над IP-многоадресной рассылкой и помогающими создавать магистраль интернет-многоадресной рассылки (aka MBONE). MBONE был создан Стивом Дирингом (который первым разработал IP-многоадресную рассылку), Ваном Джекобсоном и Стивом Касснером. У Стива Дирринга и меня был один научный руководитель в Стэнфорде, и Стив закончил работать в Xerox PARC, когда покинул Стэнфорд, я провел лето в Xerox PARC стажером, работая над проектами, связанными с IP-многоадресной рассылкой, и продолжал

работать на них неполный день, будучи в Стэнфорде, а позже - полный день. Ван Джекобсон и Стив Касснер были двумя из четырех авторов первоначальных RTP RFCs, вместе с Хеннингом Шульцринном и мной. У нас всех были инструменты MBONE, над которыми мы работали и которые позволяли различные формы онлайн-сотрудничества, и попытка создать общий базовый протокол, который могли бы использовать все эти инструменты, и привела к RTP.

Многоадресная рассылка очень интересна. WebRTC полностью использует одноадресную рассылку, не могли бы вы рассказать об этом подробнее?

Перед тем, как попасть в Стэнфорд и узнать о IP-многоадресной рассылке, у меня была длинная история работы над способами использования компьютеров для общения людей друг с другом. Для меня это началось в начале 80-х, когда я запустил систему доски объявлений с набором номеров, где люди могли войти и оставлять друг другу сообщения, как частные (что-то вроде электронной почты), так и публичные (группы обсуждений). Примерно в то же время я узнал об онлайн-провайдере CompuServe. Одной из крутых функций на CompuServe был так называемый “CB Simulator”, где люди могли общаться друг с другом в реальном времени. Это было полностью текстовое, но с понятием “каналов”, как в настоящей радиосвязи CB, и несколько человек могли видеть, что печатают другие, если они находились в одном канале. Я построил свою собственную версию CB, которая работала на системе разделения времени, к которой у меня был доступ и которая позволяла пользователям этой системы отправлять друг другу сообщения в реальном времени, и в течение следующих нескольких лет я работал с друзьями над разработкой более сложных инструментов реального времени на нескольких различных компьютерных системах и сетях. На самом деле, одна из этих систем до сих пор работает, и я использую ее, чтобы каждый день общаться с людьми, с которыми учился в колледже 30+ лет назад!

Все эти инструменты были текстовыми, так как компьютеры того времени в основном не имели никаких аудио/видео возможностей, но когда я попал в Стэнфорд и узнал о IP-многоадресной рассылке, я был заинтригован идеей использования многоадресной рассылки, чтобы получить что-то вроде настоящего “радио”, где вы могли бы отправить сигнал в сеть, не направляя его никому конкретно, но все, кто настроился на этот “канал”, могли бы его получить. Как оказалось, компьютер, на который я портировал код IP-многоадресной рассылки, был первым

поколением SPARC-станции от Sun, и у него действительно было встроенное аппаратное обеспечение телефонного качества! Вы могли оцифровать звук с микрофона и воспроизводить его через встроенные динамики (или через выход для наушников). Поэтому моей первой мыслью было понять, как отправить этот аудио в сеть в реальном времени, используя IP-многоадресную рассылку, и посмотреть, смогу ли я построить эквивалент “CB-радио” с реальным аудио вместо текста.

Были некоторые сложные вещи, которые нужно было решить, например, тот факт, что компьютер мог воспроизводить только один аудиопоток за раз, поэтому, если несколько человек говорили, вам нужно было математически “смешивать” несколько аудиопотоков в один перед тем, как его можно было воспроизвести, но это все можно было сделать в программном обеспечении, как только вы поняли, как работает аудиосемплирование. Этот аудиоприложение привел меня к работе над MBONE и, в конечном итоге, к переходу от аудио к видео с “nv”.

Что было упущено в протоколе, что вы хотели бы добавить? Есть ли что-то в протоколе, о чем вы жалеете?

Я не сказал бы, что жалею, но одной из больших жалоб, которые у людей в конце концов появились на RTP, была сложность реализации RTCP, контрольного протокола, который работал параллельно с основным трафиком данных RTP. Я думаю, что эта сложность была большой частью того, почему RTP не был более широко принят, особенно в случае одноадресной рассылки, где не было такой большой необходимости в некоторых функциях RTCP. По мере того, как пропускная способность сети становилась менее дефицитной, а перегрузка не была такой большой проблемой, многие люди просто стали передавать аудио и видео по обычному TCP (и позже HTTP), и в целом это работало “достаточно хорошо”, чтобы не заниматься RTP.

Что вы себе представляли, создавая RTP? Есть ли какие-то интересные проекты/идеи с RTP, которые были утеряны во времени?

Одной из забавных вещей, которые я создал, была версия классической игры “Spacewar”, которая использовала IP-многоадресную рассылку. Без какого-либо центрального сервера несколько клиентов могли запустить двоичный файл spacewar и начать трансляцию расположения своего корабля, его скорости, направления, в котором он был обращен, и подобной информации о любых “пулях”, которые он выпустил, и

все другие экземпляры могли бы перехватить эту информацию и отобразить ее локально, позволяя пользователям видеть корабли и пули друг друга, с “взрывающимися” кораблями при столкновении или попадании в них пуль. Я даже сделал “обломки” от взрыва живым объектом, который мог уничтожать другие корабли, иногда приводя к забавным цепным реакциям!

В духе оригинальной игры я визуализировал ее с использованием симулированной векторной графики, так что можно было делать такие вещи, как масштабирование вида, и все будет масштабироваться вверх/вниз. Сами корабли представляли собой набор линейных сегментов в векторной форме, которые помогли мне разработать некоторые из моих коллег в PARC, так что у каждого корабля был свой уникальный вид.

В основном, любая технология, которая могла бы извлечь пользу из потока данных в реальном времени, не требующего идеальной последовательной доставки, могла бы извлечь пользу из RTP. Так что, помимо аудио и видео, мы могли бы создавать такие вещи, как общая электронная доска. Даже передача файлов могла бы извлечь пользу из RTP, особенно в сочетании с IP-многоадресной рассылкой.

Представьте что-то вроде BitTorrent, но где вам не нужны все данные, идущие точка-точка между узлами. Изначальный сидер мог бы отправить многоадресный поток всем личинкам одновременно, и любые потери пакетов по пути могли бы быстро очищаться повторной передачей от любого узла, который успешно получил данные. Вы даже могли бы ограничить запросы повторной передачи так, чтобы какой-нибудь близлежащий узел доставил копию данных, и это также могло бы быть передано многоадресной рассылкой другим в этом регионе, так как потеря пакета в середине сети, как правило, означала бы, что все клиенты ниже этой точки пропустили одни и те же данные.

Почему вам пришлось самостоятельно заниматься сжатием видео? Не было ли чего-либо доступного в то время?

К тому времени существовали основные концепции сжатия видео, с появлением стандарта MPEG-1 примерно в то же время, когда появился “nv”, но реального времени кодирования с MPEG-1 определенно не было. Изменения, которые я внес, были связаны с тем, чтобы взять эти базовые концепции и аппроксимировать их более дешевыми алгоритмами, где я избегал таких вещей, как косинусные преобразования и

операции с плавающей точкой, и даже избегал целочисленных умножений, так как они были очень медленными на SPARC-станциях. Я старался делать все, что мог, только с помощью сложения/вычитания и побитового маскирования и сдвига, и это давало достаточно скорости, чтобы все еще ощущать что-то, похожее на видео.

В течение года или двух после выпуска “nv” появилось много различных аудио- и видеоинструментов, не только на MBONE, но и в других местах, таких как инструмент CU-SeeMe, созданный на Mac. Было очевидно, что это идея, время которой пришло. Я действительно закончил тем, что сделал “nv” совместимым с многими из этих инструментов, и в нескольких случаях другие инструменты перенимали мои кодеки “nv”, чтобы они могли взаимодействовать при использовании моей схемы сжатия.

WebRTC

WebRTC потребовал усилий по стандартизации, которые превосходят все остальные усилия, описанные в этой главе. Он потребовал сотрудничества между двумя различными организациями по стандартизации (IETF и W3C) и сотен отдельных лиц из многих компаний и стран. Чтобы дать нам взглянуть на мотивации и колоссальные усилия, потребовавшиеся для создания WebRTC, у нас есть Серж Лашапель.

Серж - продукт-менеджер в Google, в настоящее время работающий продукт-менеджером для Google Workspace. Это мое резюме интервью.

Что привело вас к работе над WebRTC?

Я был увлечен созданием коммуникационного программного обеспечения с тех пор, как учился в колледже. В 90-х годах начала появляться технология, подобная nv, но ее было сложно использовать. Я создал проект, который позволял присоединиться к видеозвонку прямо из браузера. Я также портировал его на Windows.

Я взял этот опыт в Marratech, компанию, которую я сооснователь. Мы создали программное обеспечение для групповых видеоконференций. Технологический ландшафт был совершенно иным. Передовым в видео был многоадресный режим сети. Пользователь мог полагаться на сеть для доставки видеопакета всем участникам звонка. Это означало, что у нас были очень простые серверы. Однако это имело большой недостаток - сети должны были быть разработаны для его поддержки. Индустрия отошла от

многоадресной рассылки к пакетным коммутаторам, более известным как SFU.

Первый проект Google

Первый проект, над которым работала будущая команда WebRTC, был голосовой и видеочат Gmail. Внедрение аудио и видео в браузер было непростой задачей. Это требовало специальных компонентов, которые нужно было лицензировать у разных компаний. Аудио был лицензирован у GIPs, видео - у Vidyo, а сетевая часть - у libjingle. Магия заключалась в том, чтобы заставить все это работать вместе.

Каждая подсистема имеет совершенно разные API и предполагает решение разных проблем. Чтобы заставить все это работать вместе, нужны глубокие знания сетей, криптографии, медиа и многого другого. Джастин Убертти был тем, кто взял на себя эту работу. Он объединил эти компоненты, чтобы создать пригодный к использованию продукт.

Отрисовка в реальном времени в браузере также была очень сложной. Нам пришлось использовать NPAPI (Netscape Plugin API) и делать много хитрых вещей, чтобы это работало. Уроки, которые мы извлекли из этого проекта, очень повлияли на WebRTC.

Chrome

В то же время внутри Google начался проект Chrome. Было много волнения, и у этого проекта были огромные цели. Велись разговоры о WebGL, автономной работе, возможностях базы данных, низкой задержке ввода для игр - и это только некоторые из них.

Отказ от NPAPI стал большим приоритетом. Это мощный API, но он несет большие последствия для безопасности. Chrome использует дизайн песочницы, чтобы защитить пользователей. Операции, которые могут быть потенциально небезопасными, выполняются в разных процессах. Даже если что-то пойдет не так, злоумышленник все еще не получит доступ к пользовательским данным.

WebRTC рождается

Для меня WebRTC родился с несколькими мотивациями. В совокупности они дали жизнь этому усилию.

Не должно быть так сложно создавать RTC-опыт. Слишком много усилий тратится впустую на повторную реализацию одного и того же разными разработчиками. Мы должны решить эти раздражающие проблемы интеграции один раз и сосредоточиться на других вещах.

Человеческое общение не должно быть ограничено и должно быть открытым. Как может быть нормально, что текст и HTML открыты, но мой голос и мое изображение в реальном времени - нет?

Безопасность - приоритет. Использование NPAPI не было лучшим для пользователей. Это также был шанс создать протокол, безопасный по умолчанию.

Чтобы сделать WebRTC реальностью, Google приобрел и открыл исходный код компонентов, которые использовались ранее. On2 был приобретен за его видеотехнологию, а Global IP Solutions за его RTC-технологию. Я отвечал за усилия по приобретению GIPS. Мы приступили к работе по объединению этих технологий и их упрощению для использования внутри и вне браузера.

Стандартизация

Стандартизация WebRTC была чем-то, что мы действительно хотели сделать, но чего я раньше не делал, как и кто-либо из нашей непосредственной команды. Для этого нам очень повезло иметь Харальда Альвестранда в Google. Он уже проделал обширную работу в IETF и начал процесс стандартизации WebRTC.

Летом 2010 года в Маастрихте был запланирован неформальный обед. Разработчики из многих компаний собрались вместе, чтобы обсудить, каким должен быть WebRTC. На обеде присутствовали инженеры из Google, Cisco, Ericsson, Skype, Mozilla, Linden Labs и других. Полный список участников и презентационные слайды можно найти на rtc-web.alvestrand.com.

Skype также предоставил отличное руководство благодаря работе, которую они проделали с Opus в IETF.

Опираясь на плечи гигантов

При работе в IETF вы развиваете работу, которая была до вас. С WebRTC нам повезло, что существовало так много вещей. Нам не пришлось решать каждую проблему, потому что они уже были решены. Если вам не нравится предсуществующая технология, это может быть frustrating. Должна быть очень веская причина,

чтобы пренебречь существующей работой, поэтому создание собственной версии не является вариантом.

Мы также сознательно не пытались повторно стандартизировать такие вещи, как сигнализация. Это уже было решено с помощью SIP и других усилий, не связанных с IETF, и казалось, что это может стать очень политизированным. В конце концов, просто не казалось, что здесь есть много ценного, что можно добавить.

Я не остался таким же вовлеченным в стандартизацию, как Джастин и Харальд, но мне понравилось время, проведенное там. Я был более взволнован возвращением к созданию вещей для пользователей.

Будущее

WebRTC сегодня находится в отличном состоянии. Происходят различные итеративные изменения, но ничего конкретного, над чем бы я работал.

Я больше всего взволнован тем, что облачные вычисления могут сделать для коммуникаций. Используя передовые алгоритмы, мы можем удалять фоновый шум во время звонка и делать коммуникацию возможной там, где ее раньше не было. Мы также видим, что WebRTC распространяется далеко за пределы коммуникаций… Кто бы мог подумать, что через 9 лет он будет обеспечивать облачные игры? Все это было бы невозможно без фундамента WebRTC.

Часто задаваемые вопросы

{{<details “Почему WebRTC использует UDP?”>}} Обход NAT требует использования UDP. Без обхода NAT установление однорангового P2P-соединения было бы невозможно. UDP не обеспечивает “гарантированную доставку”, как TCP, поэтому WebRTC реализует её на уровне пользователя.

Подробнее смотрите в разделе Подключение({{< ref “03-connecting” >}}). {{

}}

{{<details “Сколько каналов данных я могу иметь?”>}} 65534 канала, так как идентификатор потока имеет 16 бит. Вы можете закрывать и открывать новые каналы в любое время. {{

}}

{{<details “Налагает ли WebRTC ограничения пропускной способности?”>}} Как каналы данных, так и RTP используют контроль перегрузки. Это означает, что WebRTC активно измеряет вашу пропускную способность и пытается использовать оптимальное количество. Это баланс между отправкой максимально возможного объема данных без перегрузки соединения. {{

}}

{{<details “Могу ли я отправлять бинарные данные?”>}} Да, вы можете отправлять как текстовые, так и бинарные данные через каналы данных. {{

}}

{{<details “Какую задержку можно ожидать с WebRTC?”>}} Для немного настроенных медиа можно ожидать задержку менее 500 миллисекунд. Если вы готовы настраивать или жертвовать качеством ради задержки, разработчики добивались задержки менее 100 миллисекунд.

Каналы данных поддерживают опцию “частичной надежности”, которая может уменьшить задержку, вызванную повторной передачей данных по ненадежному соединению. При правильной настройке было показано, что это превосходит TCP TLS-соединения. {{

}}

{{<details “Зачем нужна внеочередная доставка для каналов данных?”>}} Когда более новая информация делает старую устаревшей, например, информация о положении объекта, или когда каждое сообщение независимо от других и нужно избежать задержки блокировки первого в очереди. {{

}}

{{<details “Могу ли я отправлять аудио или видео через канал данных?”>}} Да, вы можете отправлять любые данные через канал данных. В случае браузера вы будете сами отвечать за декодирование данных и передачу их медиаплееру для воспроизведения, в то время как при использовании медиаканалов все это происходит автоматически. {{

}}

Глоссарий

- ACK: Подтверждение

- AVP: Профиль аудио и видео
- B-кадр: Двунаправленный предсказанный кадр. Частичное изображение, являющееся модификацией предыдущих и будущих изображений.
- DCEP: Протокол установления канала данных, определен в RFC 8832
- DeMux: Демультиплексор
- DLSR: Задержка с момента последнего отчета отправителя
- DTLS: Безопасность транспортного уровня для датаграмм, определен в RFC 6347
- E2E: От конца до конца
- FEC: Упреждающая коррекция ошибок
- FIR: Запрос полного внутрикадрового обновления
- G.711: Узкополосный аудиокодек
- GCC: Контроль перегрузки Google, определен в draft-ietf-rmcat-gcc-02
- H.264: Расширенное видеокодирование для универсальных аудиовизуальных служб
- H.265: Спецификация соответствия для высокоэффективного видеокодирования ITU-T H.265
- HEVC: Высокоэффективное видеокодирование
- HTTP: Протокол передачи гипертекста
- HTTPS: HTTP поверх TLS, определен в RFC 2818
- I-кадр: Внутрикадрово закодированный кадр. Полное изображение, которое может быть декодировано без чего-либо другого.
- ICE: Установление связности, определен в RFC 8445
- INIT: Инициализация
- IoT: Интернет вещей
- IPv4: Интернет-протокол, версия 4
- IPv6: Интернет-протокол, версия 6
- ITU-T: Сектор стандартизации телекоммуникаций Международного союза электросвязи
- JSEP: Протокол установления сеанса JavaScript, определен в RFC 8829
- MCU: Многоточечный конференц-блок
- mDNS: Многоадресный DNS, определен в RFC 6762
- MITM: Атака “человек посередине”
- MTU: Максимальный размер передаваемого блока данных
- MUX: Мультиплексирование
- NACK: Отрицательное подтверждение
- NADA: Динамическая адаптация, поддерживаемая сетью, определен в draft-zhu-rmcat-nada-04
- NAT: Преобразование сетевых адресов, определен в RFC 4787
- Opus: Полностью открытый, свободный от роялти, высокоуниверсальный

аудиокодек
- P-кадр: Предсказанный кадр. Частичное изображение, содержащее только изменения по сравнению с предыдущим изображением.
- P2P: Одноранговый
- PLI: Индикация потери изображения
- PPID: Идентификатор протокола полезной нагрузки
- REMB: Оценка максимальной скорости передачи получателем
- RFC: Запрос комментариев
- RMCAT: Методы избегания перегрузки медиа RTP
- RR: Отчет получателя
- RTCP: Протокол управления RTP, определен в RFC 3550
- RTP: Протокол передачи в реальном времени, определен в RFC 3550
- RTT: Время кругового обхода
- SACK: Избирательное подтверждение
- SCReAM: Самосинхронизирующаяся адаптация скорости для мультимедиа, определен в draft-johansson-rmcat-scream-cc-05
- SCTP: Протокол управления потоком передачи, определен в RFC 4960
- SDP: Протокол описания сеанса, определен в RFC 8866
- SFU: Блок селективной пересылки
- SR: Отчет отправителя
- SRTP: Защищенный протокол передачи в реальном времени, определен в RFC 3711
- SSRC: Источник синхронизации
- STUN: Утилиты обхода сеансов для NAT, определен в RFC 8489
- TCP: Протокол управления передачей
- TLS: Безопасность транспортного уровня, определена в RFC 8446
- TMMBN: Уведомление о временном максимальном битрейте медиапотока
- TMMBR: Запрос временного максимального битрейта медиапотока
- TSN: Номер последовательности передачи
- TURN: Обход с использованием ретрансляторов вокруг NAT, определен в RFC 8656
- TWCC: Контроль перегрузки по всей транспортной сети
- UDP: Протокол пользовательских датаграмм
- VP8, VP9: Высокоэффективные технологии сжатия видео (видеокодеки), разработанные проектом WebM. Любой может использовать эти кодеки без роялти.
- WebM: Открытый формат медиафайлов, разработанный для веба.

- WebRTC: Веб-коммуникации в реальном времени. Стандарт W3C WebRTC 1.0: Коммуникации в реальном времени между браузерами

Справочник

WebRTC (W3C)

- WebRTC 1.0: Коммуникации в реальном времени между браузерами [26 января 2021] (Статус: Рекомендация)
- Рабочая группа по веб-коммуникациям в реальном времени - Публикации

WebRTC (RFC)

- RFC8825: Обзор: Протоколы реального времени для браузерных приложений Х. Альвестранд [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8826: Соображения безопасности для WebRTC Э. Решкорла [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8836: Требования к контролю перегрузки для интерактивных мультимедиа в реальном времени Р. Джесуп, З. Саркер [Январь 2021] (Статус: ИНФОРМАЦИОННЫЙ)
- RFC8854: Требования к упреждающей коррекции ошибок в WebRTC Дж. Убертти [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

DTLS

- RFC6347: Безопасность транспортного уровня для датаграмм, версия 1.2 Э. Решкорла, Н. Модадугу [Январь 2012] (Устарел RFC4347) (Устарел RFC9147) (Обновлен RFC7507, RFC7905, RFC8996, RFC9146) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC9147: Протокол безопасности транспортного уровня для датаграмм (DTLS), версия 1.3 Э. Решкорла, Х. Тшофениг, Н. Модадугу [Апрель 2022] (Устарел RFC6347) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ) (Смотрите также: Статус DTLS 1.3 в OpenSSL)

Канал данных

- RFC8831: Каналы данных WebRTC Р. Джесуп, С. Лорето, М. Тюксен [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ

СТАНДАРТ)
- RFC8832: Протокол установления канала данных WebRTC Р. Джесуп, С. Лорето, М. Тюксен [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8864: Согласование каналов данных с использованием протокола описания сеанса (SDP) К. Драге, М. Макараджу, Р. Эйзак, Дж. Маркон, Р. Эвен [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

Медиатранспорт

- RFC8834: Медиатранспорт и использование RTP в WebRTC К. Перкинс, М. Вестерлунд, Дж. Отт [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8837: Маркировка пакетов кодом дифференцированного обслуживания (DSCP) для качества обслуживания WebRTC П. Джонс, С. Дхесикан, К. Дженнингс, Д. Друта [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

SCTP

- RFC3758: Протокол управления потоком передачи (SCTP) с расширением частичной надежности Р. Стюарт, М. Рамалхо, К. Сие, М. Тюксен, П. Конрад [Май 2004] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC5061: Протокол управления потоком передачи (SCTP) с динамической реконфигурацией адресов Р. Стюарт, К. Сие, М. Тюксен, С. Марушама, М. Кодзука [Сентябрь 2007] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC5827: Ранняя повторная передача для TCP и протокола управления потоком передачи (SCTP) М. Оллман, К. Аврашенков, У. Аиеста, Дж. Блэнтон, П. Хуртиг [Май 2010] (Статус: ЭКСПЕРИМЕНТАЛЬНЫЙ)
- RFC6083: Безопасность транспортного уровня для датаграмм (DTLS) для протокола управления потоком передачи (SCTP) М. Тюксен, Р. Сеггельман, Э. Решкорла [Январь 2011] (Обновлен RFC8996) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC6525: Протокол управления потоком передачи (SCTP) с реконфигурацией потока Р. Стюарт, М. Тюксен, П. Лей [Февраль 2012] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC6951: Инкапсуляция пакетов протокола управления потоком передачи (SCTP) в UDP для связи между конечными хостами М. Тюксен, Р. Стюарт [Май 2013] (Обновлен RFC8899) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

- RFC7765: Перезапуск RTO для TCP и протокола управления потоком передачи (SCTP) П. Хуртиг, А. Брунстром, А. Петлунд, М. Вельзл [Февраль 2016] (Статус: ЭКСПЕРИМЕНТАЛЬНЫЙ)
- RFC8260: Планировщики потоков и чередование пользовательских сообщений для протокола управления потоком передачи Р. Стюарт, М. Тюксен, С. Лорето, Р. Сеггельман [Ноябрь 2017] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8261: Инкапсуляция пакетов SCTP в безопасность транспортного уровня для датаграмм (DTLS) М. Тюксен, Р. Стюарт, Р. Джесуп, С. Лорето [Ноябрь 2017] (Обновлен RFC8899, RFC8996) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8841: Процедуры предложения/ответа протокола описания сеанса (SDP) для протокола управления потоком передачи (SCTP) поверх безопасности транспортного уровня для датаграмм (DTLS) К. Холмберг, Р. Шпоунт, С. Лорето, Г. Камарилло [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8899: Обнаружение максимального размера блока передачи на уровне пакетизации для транспортных датаграмм Г. Фэрхерст, Т. Джонс, М. Тюксен, И. Рюнгелер, Т. Фолькер [Сентябрь 2020] (Обновляет RFC4821, RFC4960, RFC6951, RFC8085, RFC8261) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC9260: Протокол управления потоком передачи Р. Стюарт, М. Тюксен, К. Нильсен [Июнь 2022] (Устарел RFC4460, RFC4960, RFC6096, RFC7053, RFC8540) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

SDP

- RFC8829: Протокол установления сеанса JavaScript (JSEP) Дж. Убертти, К. Дженнингс, Э. Решкорла [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8830: Идентификация медиапотока WebRTC в протоколе описания сеанса Х. Альвестранд [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8839: Процедуры предложения/ответа протокола описания сеанса (SDP) для установления связности (ICE) М. Пти-Хюгенен, С. Нандакумар, К. Холмберг, А. Кераннен, Р. Шпоунт [Январь 2021] (Устарел RFC5245, RFC6336) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8841: Процедуры предложения/ответа протокола описания сеанса (SDP) для протокола управления потоком передачи (SCTP) поверх безопасности транспортного уровня для датаграмм (DTLS) К. Холмберг, Р. Шпоунт, С.

Лорето, Г. Камарилло [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

- RFC8843: Согласование мультиплексирования медиа с использованием протокола описания сеанса (SDP) К. Холмберг, Х. Альвестранд, К. Дженнингс [Январь 2021] (Устарел RFC9143) (Обновляет RFC3264, RFC5888, RFC7941) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8844: Атаки с неизвестным обменом ключами при использовании TLS с протоколом описания сеанса (SDP) М. Томсон, Э. Решкорла [Январь 2021] (Обновляет RFC8122) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8851: Ограничения формата полезной нагрузки RTP А.Б. Роуч [Январь 2021] (Обновляет RFC4855) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8852: Описание источника идентификатора потока RTP А.Б. Роуч, С. Нандакумар, П. Тэтчер [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8853: Использование симультивещания в протоколе описания сеанса (SDP) и сеансах RTP Б. Бурман, М. Вестерлунд, С. Нандакумар, М. Занати [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8866: SDP: Протокол описания сеанса А. Беген, П. Кызиват, К. Перкинс, М. Хэндли [Январь 2021] (Устарел RFC4566) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

RTP

- RFC3550: RTP: Транспортный протокол для приложений реального времени Х. Шульцринне, С. Каснер, Р. Фредерик, В. Джейкобсон [Июль 2003] (Устарел RFC1889) (Обновлен RFC5506, RFC5761, RFC6051, RFC6222, RFC7022, RFC7160, RFC7164, RFC8083, RFC8108, RFC8860) (Также STD0064) (Статус: ИНТЕРНЕТ-СТАНДАРТ)
- RFC3611: Расширенные отчеты протокола управления RTP (RTCP XR) Т. Фридман, Р. Касерес, А. Кларк [Ноябрь 2003] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC3711: Защищенный протокол передачи в реальном времени (SRTP) М. Баугер, Д. Макгрю, М. Наслунд, Э. Каррара, К. Норрман [Март 2004] (Обновлен RFC5506, RFC6904) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC4585: Расширенный профиль RTP для обратной связи на основе протокола управления RTP (RTP/AVPF) Дж. Отт, С. Вингер, Н. Сато, К. Бурмейстер, Дж. Рей [Июль 2006] (Обновлен RFC5506, RFC8108) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

- RFC5104: Сообщения управления кодеком в аудиовизуальном профиле RTP с обратной связью (AVPF) С. Вингер, У. Чандра, М. Вестерлунд, Б. Бурман [Февраль 2008] (Обновлен RFC7728, RFC8082) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC5764: Расширение безопасности транспортного уровня для датаграмм (DTLS) для установления ключей для защищенного протокола передачи в реальном времени (SRTP) Д. Макгрю, Э. Решкорла [Май 2010] (Обновлен RFC7983) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC6904: Шифрование расширений заголовков в защищенном протоколе передачи в реальном времени (SRTP) Дж. Леннокс [Апрель 2013] (Обновляет RFC3711) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC7741: Формат полезной нагрузки RTP для видео VP8 П. Вестин, Х. Лундин, М. Гловер, Дж. Убертти, Ф. Галлиган [Март 2016] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8285: Общий механизм расширений заголовков RTP Д. Сингер, Х. Десинени, Р. Эвен [Октябрь 2017] (Устарел RFC5285) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8852: Описание источника идентификатора потока RTP А.Б. Роуч, С. Нандакумар, П. Тэтчер [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8858: Указание эксклюзивной поддержки мультиплексирования RTP и протокола управления RTP (RTCP) с использованием протокола описания сеанса (SDP) К. Холмберг [Январь 2021] (Обновляет RFC5761) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8860: Отправка нескольких типов медиа в одной сессии RTP М. Вестерлунд, К. Перкинс, Дж. Леннокс [Январь 2021] (Обновляет RFC3550, RFC3551) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8867: Тестовые случаи для оценки контроля перегрузки для интерактивных мультимедиа в реальном времени З. Саркер, В. Сингх, Х. Чжу, М. Рамалхо [Январь 2021] (Статус: ИНФОРМАЦИОННЫЙ)
- RFC8868: Оценка контроля перегрузки для интерактивных мультимедиа в реальном времени В. Сингх, Дж. Отт, С. Холмер [Январь 2021] (Статус: ИНФОРМАЦИОННЫЙ)
- RFC8869: Тестовые случаи оценки интерактивных мультимедиа в реальном времени по беспроводным сетям З. Саркер, Х. Чжу, Дж. Фу [Январь 2021] (Статус: ИНФОРМАЦИОННЫЙ)
- RFC8872: Руководство по использованию функций мультиплексирования RTP для поддержки нескольких медиапотоков М. Вестерлунд, Б. Бурман, К. Перкинс, Х. Альвестранд, Р. Эвен [Январь 2021] (Статус: ИНФОРМАЦИОННЫЙ)
- RFC8888: Обратная связь протокола управления RTP (RTCP)

для контроля перегрузки З. Саркер, К. Перкинс, В. Сингх, М. Рамалхо [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)

ICE, TURN и STUN

- RFC5780: Обнаружение поведения NAT с использованием утилит обхода сеансов для NAT (STUN) Д. Макдональд, Б. Лоуекамп [Май 2010] (Обновлен RFC8553) (Статус: ЭКСПЕРИМЕНТАЛЬНЫЙ)
- RFC8445: Установление связности (ICE): Протокол обхода преобразователей сетевых адресов (NAT) А. Кераннен, К. Холмберг, Дж. Розенберг [Июль 2018] (Устарел RFC5245) (Обновлен RFC8863) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8489: Утилиты обхода сеансов для NAT (STUN) М. Пти-Хюгенен, Г. Салгейро, Дж. Розенберг, Д. Уинг, Р. Мэхи, П. Мэттьюз [Февраль 2020] (Устарел RFC5389) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8656: Обход с использованием ретрансляторов вокруг NAT (TURN): Расширения ретрансляции для утилит обхода сеансов NAT (STUN) Т. Редди, А. Джонстон, П. Мэттьюз, Дж. Розенберг [Февраль 2020] (Устарел RFC5766, RFC6156) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8835: Транспорты для WebRTC Х. Альвестранд [Январь 2021] (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8838: Trickle ICE: Постепенное предоставление кандидатов для протокола установления связности (ICE) Э. Ивов, Дж. Убертти, П. Сент-Андре [Январь 2021] (Обновлен RFC8863) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8839: Процедуры предложения/ответа протокола описания сеанса (SDP) для установления связности (ICE) М. Пти-Хюгенен, С. Нандакумар, К. Холмберг, А. Кераннен, Р. Шпоунт [Январь 2021] (Устарел RFC5245, RFC6336) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)
- RFC8863: Установление связности с терпеливым ожиданием связности (ICE PAC) К. Холмберг, Дж. Убертти [Январь 2021] (Обновляет RFC8445, RFC8838) (Статус: ПРЕДЛАГАЕМЫЙ СТАНДАРТ)