WebRTC For The Curious

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

Contents

WebRTC For The Curious

This book was created by WebRTC implementers to share their hard-earned knowledge with the world. *WebRTC For The Curious* is an Open Source book written for those that are always looking for more. This book doesn't settle for abstraction.

This book is all about protocols and APIs, and will not be talking about any software in particular. We attempt to summarize RFCs and get all undocumented knowledge into one place. This book is not a tutorial, and will not contain much code.

WebRTC is a wonderful technology, but it is difficult to use. This book is vendor agnostic, and we have tried to remove any conflicts of interest.

Who this book is for:

- Developers who don't even know what WebRTC solves, and want to learn more.
- Someone who is already building with WebRTC, but wants to know more beyond the APIs.
- Established developers who need help debugging.
- WebRTC implementer who needs clarification on a specific part.

Designed for multiple readings

This book is designed to be read multiple times. Each chapter is self-contained, so you can jump to any part of the book and not be lost.

Each chapter aims to answer a single question, with three levels of information:

- What needs to be solved?
- How do we solve it? (Including technical details about the solution.)
- Where to learn more.

Each chapter doesn't assume prior knowledge. You can start at any point in the book and begin learning. This book will also recommend resources to go and

learn more. Other books cover individual topics in much greater depth. This book aims to teach you the entire system, at the cost of expert level details.

Freely available and privacy respecting

This book is available on GitHub and WebRTCforTheCurious.com. It is licensed in a way that you can use it however you think is best. You can also download the book in its current version as an ePub or PDF file.

This book is written by individuals, for individuals. It is vendor agnostic, so we will not make recommendations that could be a conflict of interest.

The website will not use analytics or tracking.

Get involved!

We need your help! This book is entirely developed on GitHub and is still being written. We encourage readers to open issues with questions on things we didn't do a good job of covering yet.

License

What is WebRTC?

WebRTC, short for Web Real-Time Communication, is both an API and a Protocol. The WebRTC protocol is a set of rules for two WebRTC agents to negotiate bi-directional secure real-time communication. The WebRTC API then allows developers to use the WebRTC protocol. The WebRTC API is specified only for JavaScript.

A similar relationship would be the one between HTTP and the Fetch API. WebRTC the protocol would be HTTP, and WebRTC the API would be the Fetch API.

The WebRTC protocol is available in other APIs and languages besides JavaScript. You can find servers and domain-specific tools as well for WebRTC. All of these implementations use the WebRTC protocol so that they can interact with each other.

The WebRTC protocol is maintained in the IETF in the rtcweb working group. The WebRTC API is documented in the W3C as webrtc-pc.

Why should I learn WebRTC?

These are the things that WebRTC will give you. This list is not exhaustive, just an example of some things you may appreciate during your journey. Don't worry if you don't know all these terms yet, this book will teach them to you along the way.

- Open standard
- Multiple implementations
- Available in browsers
- Mandatory encryption
- NAT Traversal
- Repurposed existing technology
- Congestion control
- Sub-second latency

The WebRTC Protocol is a collection of other technologies

This is a topic that takes an entire book to explain. However, to start off we break it into four steps.

- Signaling
- Connecting
- Securing
- Communicating

These four steps happen sequentially. The prior step must be 100% successful for the subsequent one to even begin.

One peculiar fact about WebRTC is that each step is actually made up of many other protocols! To make WebRTC, we stitch together many existing technologies. In that sense, WebRTC is more a combination and configuration of well-understood tech that has been around since the early 2000s.

Each of these steps has dedicated chapters, but it is helpful to understand them at a high level first. Since they depend on each other, it will help when explaining further the purpose of each of these steps.

Signaling: How peers find each other in WebRTC

When a WebRTC Agent starts it has no idea who it is going to communicate with and what they are going to communicate about. Signaling solves this issue! Signaling is used to bootstrap the call so that two WebRTC agents can start communicating.

Signaling uses an existing protocol SDP (Session Description Protocol). SDP is a plain-text protocol. Each SDP message is made up of key/value pairs and contains a list of "media sections". The SDP that the two WebRTC Agents exchange contains details like:

- IPs and Ports that the agent is reachable on (candidates).
- How many audio and video tracks the agent wishes to send.
- What audio and video codecs each agent supports.
- Values used while connecting (`uFrag`/`uPwd`).
- Values used while securing (certificate fingerprint).

Note that signaling typically happens "out-of-band"; that is, applications generally don't use WebRTC itself to trade signaling messages. Any architecture suitable for sending messages can be used to relay the SDPs between the connecting peers, and many applications will use their existing infrastructure (like REST endpoints, WebSocket connections, or authentication proxies) to facilitate easy trading of SDPs between the proper clients.

Connecting and NAT Traversal with STUN/TURN

The two WebRTC Agents now know enough details to attempt to connect to each other. WebRTC then uses another established technology called ICE.

ICE (Interactive Connectivity Establishment) is a protocol that pre-dates WebRTC. ICE allows the establishment of a connection between two Agents. These Agents could be on the same network, or on the other side of the world. ICE is the solution to establishing a direct connection without a central server.

The real magic here is 'NAT Traversal' and STUN/TURN Servers. These two concepts are all you need to communicate with an ICE Agent in another subnet. We will explore these topics in depth later.

Once ICE successfully connects, WebRTC then moves on to establishing an encrypted transport. This transport is used for audio, video, and data.

Securing the transport layer with DTLS and SRTP

Now that we have bi-directional communication (via ICE) we need to establish secure communication. This is done through two protocols that pre-date WebRTC. The first protocol is DTLS (Datagram Transport Layer Security) which is just TLS over UDP. TLS is the cryptographic protocol used to secure communication over HTTPS. The second protocol is SRTP (Secure Real-time Transport Protocol).

First, WebRTC connects by doing a DTLS handshake over the connection established by ICE. Unlike HTTPS, WebRTC doesn't use a central authority for certificates. Instead, WebRTC just asserts that the certificate exchanged via DTLS matches the fingerprint shared via signaling. This DTLS connection is then used for DataChannel messages.

WebRTC then uses a different protocol for audio/video transmission called RTP. We secure our RTP packets using SRTP. We initialize our SRTP session by extracting the keys from the negotiated DTLS session. In a later chapter, we discuss why media transmission has its own protocol.

Now we are done! You now have bi-directional and secure communication. If you have a stable connection between your WebRTC Agents, this is all the complexity you may need. Unfortunately, the real world has packet loss and bandwidth limits, and the next section is about how we deal with them.

Communicating with peers via RTP and SCTP

We now have two WebRTC Agents with secure bi-directional communication. Let's start communicating! Again, we use two pre-existing protocols: RTP (Real-time Transport Protocol), and SCTP (Stream Control Transmission Protocol). Use RTP to exchange media encrypted with SRTP, and use SCTP to send and receive DataChannel messages encrypted with DTLS.

RTP is quite minimal but provides what is needed to implement real-time streaming. The important thing is that RTP gives flexibility to the developer, so they can handle latency, loss, and congestion as they please. We will discuss this further in the media chapter.

The final protocol in the stack is SCTP. SCTP allows many delivery options for messages. You can optionally choose to have unreliable, out of order delivery, so you can get the latency needed for real-time systems.

WebRTC, a collection of protocols

WebRTC solves a lot of problems. At first, this may even seem over-engineered. The genius of WebRTC is really the humility. It didn't assume it could solve everything better. Instead, it embraced many existing single purpose technologies and bundled them together.

This allows us to examine and learn each part individually without being overwhelmed. A good way to visualize it is a 'WebRTC Agent' is really just an orchestrator of many different protocols.

How does WebRTC (the API) work

This section shows how the JavaScript API maps to the protocol. This isn't meant as an extensive demo of the WebRTC API, but more to create a mental model of how it all ties together. If you aren't familiar with either, it is ok. This could be a fun section to return to as you learn more!

`new RTCPeerConnection` The `RTCPeerConnection` is the top-level "WebRTC Session". It contains all the protocols mentioned above. The subsystems are all allocated but nothing happens yet.

`addTrack` `addTrack` creates a new RTP stream. A random Synchronization Source (SSRC) will be generated for this stream. This stream will then be inside

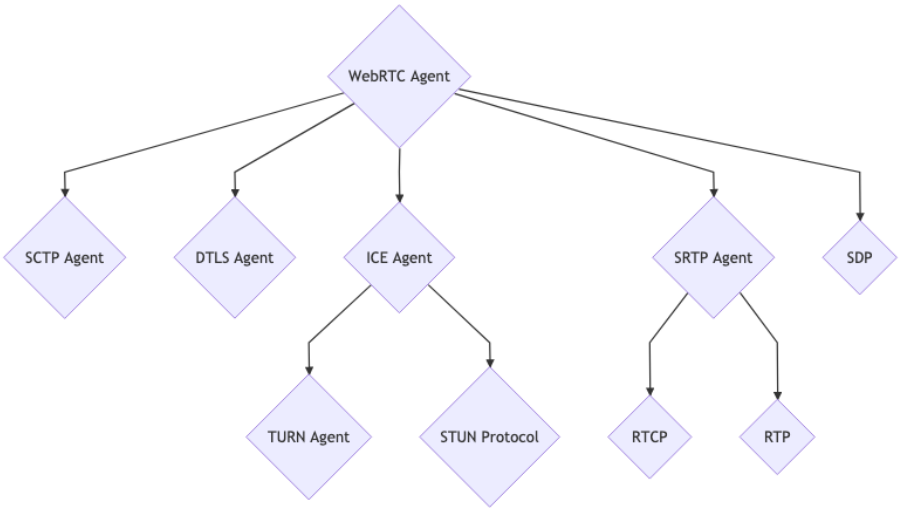


Figure 1: WebRTC Agent

the Session Description generated by `createOffer` inside a media section. Each call to `addTrack` will create a new SSRC and media section.

Immediately after a SRTP Session is established these media packets will start being sent via ICE after being encrypted using SRTP.

`createDataChannel` `createDataChannel` creates a new SCTP stream if no SCTP association exists. By default, SCTP is not enabled but is only started when one side requests a data channel.

Immediately after a DTLS Session is established, the SCTP association will start sending packets via ICE and encrypted with DTLS.

`createOffer` `createOffer` generates a Session Description of the local state to be shared with the remote peer.

The act of calling `createOffer` doesn't change anything for the local peer.

`setLocalDescription` `setLocalDescription` commits any requested changes. `addTrack`, `createDataChannel` and similar calls are all temporary until this call. `setLocalDescription` is called with the value generated by `createOffer`.

Usually, after this call, you will send the offer to the remote peer, and they will call `setRemoteDescription` with it.

`setRemoteDescription` `setRemoteDescription` is how we inform the local agent about the remote candidates' state. This is how the act of 'Signaling' is done with the JavaScript API.

When `setRemoteDescription` has been called on both sides, the WebRTC Agents now have enough info to start communicating Peer-To-Peer (P2P)!

`addIceCandidate` `addIceCandidate` allows a WebRTC agent to add more remote ICE Candidates whenever they want. This API sends the ICE Candidate right into the ICE subsystem and has no other effect on the greater WebRTC connection.

`ontrack` `ontrack` is a callback that is fired when an RTP packet is received from the remote peer. The incoming packets would have been declared in the Session Description that was passed to `setRemoteDescription`.

WebRTC uses the SSRC and looks up the associated `MediaStream` and `MediaStreamTrack`, and fires this callback with these details populated.

`oniceconnectionstatechange` `oniceconnectionstatechange` is a callback that is fired that reflects the state of the ICE Agent. When you have network connectivity or when you become disconnected this is how you are notified.

`onstatechange` `onstatechange` is a combination of ICE Agent and DTLS Agent state. You can watch this to be notified when ICE and DTLS have both completed successfully.

What is WebRTC Signaling?

When you create a WebRTC agent, it knows nothing about the other peer. It has no idea who it is going to connect with or what they are going to send! Signaling is the initial bootstrapping that makes a call possible. After these values are exchanged, the WebRTC agents can communicate directly with each other.

Signaling messages are just text. The WebRTC agents don't care how they are transported. They are commonly shared via Websockets, but that is not a requirement.

How does WebRTC signaling work?

WebRTC uses an existing protocol called the Session Description Protocol. Via this protocol, the two WebRTC Agents will share all the state required to establish a connection. The protocol itself is simple to read and understand. The complexity comes from understanding all the values that WebRTC populates it with.

This protocol is not specific to WebRTC. We will learn the Session Description Protocol first without even talking about WebRTC. WebRTC only really takes advantage of a subset of the protocol, so we are only going to cover what we need. After we understand the protocol, we will move on to its applied usage in WebRTC.

What is the *Session Description Protocol* (SDP)?

The Session Description Protocol is defined in RFC 4566. It is a key/value protocol with a newline after each value. It will feel similar to an INI file. A Session Description contains zero or more Media Descriptions. Mentally you can model it as a Session Description that contains an array of Media Descriptions.

A Media Description usually maps to a single stream of media. So if you wanted to describe a call with three video streams and two audio tracks you would have five Media Descriptions.

How to read the SDP

Every line in a Session Description will start with a single character, this is your key. It will then be followed by an equal sign. Everything after that equal sign is the value. After the value is complete, you will have a newline.

The Session Description Protocol defines all the keys that are valid. You can only use letters for keys as defined in the protocol. These keys all have significant meaning, which will be explained later.

Take this Session Description excerpt:

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

You have two lines. Each with the key `a`. The first line has the value `my-sdp-value`, the second line has the value `second-value`.

WebRTC only uses some SDP keys

Not all key values defined by the Session Description Protocol are used by WebRTC. The following seven keys are the only ones you need to understand right now:

- `v` - Version, should be equal to `0`.
- `o` - Origin, contains a unique ID useful for renegotiations.
- `s` - Session Name, should be equal to `-`.
- `t` - Timing, should be equal to `0 0`.
- `m` - Media Description (`m=<media> <port> <proto> <fmt> ...`), described in detail below.
- `a` - Attribute, a free text field. This is the most common line in WebRTC.
- `c` - Connection Data, should be equal to `IN IP4 0.0.0.0`.

Media Descriptions in a Session Description

A Session Description can contain an unlimited number of Media Descriptions.

A Media Description definition contains a list of formats. These formats map to RTP Payload Types. The actual codec is then defined by an Attribute with the value `rtpmap` in the Media Description. The importance of RTP and RTP Payload Types is discussed later in the Media chapter. Each Media Description can contain an unlimited number of attributes.

Take this Session Description excerpt as an example:

```
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
```

You have two Media Descriptions, one of type audio with fmt `111` and one of type video with the format `96`. The first Media Description has only one attribute. This attribute maps the Payload Type `111` to Opus. The second Media Description has two attributes. The first attribute maps the Payload Type `96` to be VP8, and the second attribute is just `my-sdp-value`.

Full Example

The following brings all the concepts we have talked about together. These are all the features of the Session Description Protocol that WebRTC uses. If you can read this, you can read any WebRTC Session Description!

```
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` are defined, but they do not affect the WebRTC session.
- You have two Media Descriptions. One of type `audio` and one of type `video`.
- Each of those has one attribute. This attribute configures details of the RTP pipeline, which is discussed in the "Media Communication" chapter.

How *Session Description Protocol* and WebRTC work together

The next piece of the puzzle is understanding *how* WebRTC uses the Session Description Protocol.

What are Offers and Answers?

WebRTC uses an offer/answer model. All this means is that one WebRTC Agent makes an "Offer" to start a call, and the other WebRTC Agents "Answers" if it is willing to accept what has been offered.

This gives the answerer a chance to reject unsupported codecs in the Media Descriptions. This is how two peers can understand what formats they are willing to exchange.

Transceivers are for sending and receiving

Transceivers is a WebRTC specific concept that you will see in the API. What it is doing is exposing the "Media Description" to the JavaScript API. Each Media Description becomes a Transceiver. Every time you create a Transceiver a new Media Description is added to the local Session Description.

Each Media Description in WebRTC will have a direction attribute. This allows a WebRTC Agent to declare "I am going to send you this codec, but I am not willing to accept anything back". There are four valid values:

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

SDP Values used by WebRTC

This is a list of some common attributes that you will see in a Session Description from a WebRTC Agent. Many of these values control the subsystems that we haven't discussed yet.

`group:BUNDLE` Bundling is an act of running multiple types of traffic over one connection. Some WebRTC implementations use a dedicated connection per media stream. Bundling should be preferred.

`fingerprint:sha-256` This is a hash of the certificate a peer is using for DTLS. After the DTLS handshake is completed, you compare this to the actual certificate to confirm you are communicating with whom you expect.

setup: This controls the DTLS Agent behavior. This determines if it runs as a client or server after ICE has connected. The possible values are:

- `setup:active` - Run as DTLS Client.
- `setup:passive` - Run as DTLS Server.
- `setup:actpass` - Ask the other WebRTC Agent to choose.

ice-ufrag This is the user fragment value for the ICE Agent. Used for the authentication of ICE Traffic.

ice-pwd This is the password for the ICE Agent. Used for authentication of ICE Traffic.

rtpmap This value is used to map a specific codec to an RTP Payload Type. Payload types are not static, so for every call the offerer decides the payload types for each codec.

fmtp Defines additional values for one Payload Type. This is useful to communicate a specific video profile or encoder setting.

candidate This is an ICE Candidate that comes from the ICE Agent. This is one possible address that the WebRTC Agent is available on. These are fully explained in the next chapter.

ssrc A Synchronization Source (SSRC) defines a single media stream track.

`label` is the ID for this individual stream. `mslabel` is the ID for a container that can have multiple streams inside it.

Example of a WebRTC Session Description

The following is a complete Session Description generated by a WebRTC Client:

```
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:98:14:8D:1
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 57336 ge
a=candidate:foundation 2 udp 1694498815 1.2.3.4 57336 typ srflx raddr 0.0.0.0 rport 57336 ge
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
```

This is what we know from this message:

- We have two media sections, one audio and one video.
- Both of them are `sendrecv` transceivers. We are getting two streams, and we can send two back.
- We have ICE Candidates and Authentication details, so we can attempt to connect.
- We have a certificate fingerprint, so we can have a secure call.

Further Topics

In later versions of this book, the following topics will also be addressed:

- Renegotiation
- Simulcast

Why does WebRTC need a dedicated subsystem for connecting?

Most applications deployed today establish client/server connections. A client/server connection requires the server to have a stable well-known transport address. A client contacts a server, and the server responds.

WebRTC doesn't use a client/server model, it establishes peer-to-peer (P2P) connections. In a P2P connection the task of creating a connection is equally distributed to both peers. This is because a transport address (IP and port) in WebRTC can not be assumed, and may even change during the session. WebRTC will gather all the information it can and will go to great lengths to achieve bi-directional communication between two WebRTC Agents.

Establishing peer-to-peer connectivity can be difficult though. These agents could be in different networks with no direct connectivity. In situations where direct connectivity does exist you can still have other issues. In some cases, your clients don't speak the same network protocols (UDP <-> TCP) or maybe use different IP Versions (IPv4 <-> IPv6).

Despite these difficulties in setting up a P2P connection, you get advantages over traditional Client/Server technology because of the following attributes that WebRTC offers.

Reduced Bandwidth Costs

Since media communication happens directly between peers you don't have to pay for, or host a separate server to relay media.

Lower Latency

Communication is faster when it is direct! When a user has to run everything through your server, it makes transmissions slower.

Secure E2E Communication

Direct Communication is more secure. Since users aren't routing data through your server, they don't even need to trust you won't decrypt it.

How does it work?

The process described above is called Interactive Connectivity Establishment (ICE). Another protocol that pre-dates WebRTC.

ICE is a protocol that tries to find the best way to communicate between two ICE Agents. Each ICE Agent publishes the ways it is reachable, these are known as candidates. A candidate is essentially a transport address of the agent that

it believes the other peer can reach. ICE then determines the best pairing of candidates.

The actual ICE process is described in greater detail later in this chapter. To understand why ICE exists, it is useful to understand what network behaviors we are overcoming.

Networking real-world constraints

ICE is all about overcoming the constraints of real-world networks. Before we explore the solution, let's talk about the actual problems.

Not in the same network

Most of the time the other WebRTC Agent will not even be in the same network. A typical call is usually between two WebRTC Agents in different networks with no direct connectivity.

Below is a graph of two distinct networks, connected over public internet. In each network you have two hosts.

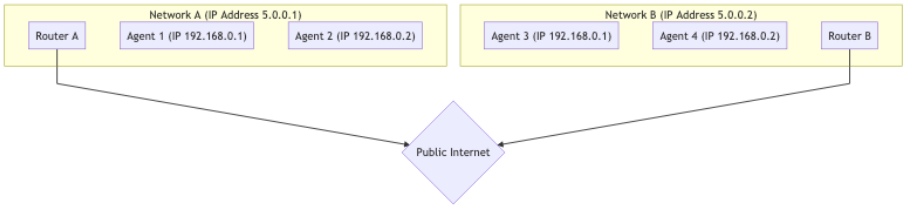


Figure 2: Two networks

For the hosts in the same network it is very easy to connect. Communication between `192.168.0.1 -> 192.168.0.2` is easy to do! These two hosts can connect to each other without any outside help.

However, a host using `Router B` has no way to directly access anything behind `Router A`. How would you tell the difference between `191.168.0.1` behind `Router A` and the same IP behind `Router B`? They are private IPs! A host using `Router B` could send traffic directly to `Router A`, but the request would end there. How does `Router A` know which host it should forward the message to?

Protocol Restrictions

Some networks don't allow UDP traffic at all, or maybe they don't allow TCP. Some networks may have a very low MTU (Maximum Transmission Unit). There are lots of variables that network administrators can change that can make communication difficult.

Firewall/IDS Rules

Another is "Deep Packet Inspection" and other intelligent filtering. Some network administrators will run software that tries to process every packet. Many times this software doesn't understand WebRTC, so it blocks it because it doesn't know what to do, e.g. treating WebRTC packets as suspicious UDP packets on an arbitrary port that is not whitelisted.

NAT Mapping

NAT (Network Address Translation) mapping is the magic that makes the connectivity of WebRTC possible. This is how WebRTC allows two peers in completely different subnets to communicate, addressing the "not in the same network" problem above. While it creates new challenges, let's explain how NAT mapping works in the first place.

It doesn't use a relay, proxy, or server. Again we have `Agent 1` and `Agent 2` and they are in different networks. However, traffic is flowing completely through. Visualized it looks like this:

To make this communication happen you establish a NAT mapping. Agent 1 uses port 7000 to establish a WebRTC connection with Agent 2. This creates a binding of `192.168.0.1:7000` to `5.0.0.1:7000`. This then allows Agent 2 to reach Agent 1 by sending packets to `5.0.0.1:7000`. Creating a NAT mapping like in this example is like an automated version of doing port forwarding in your router.

The downside to NAT mapping is that there isn't a single form of mapping (e.g. static port forwarding), and the behavior is inconsistent between networks. ISPs and hardware manufacturers may do it in different ways. In some cases, network administrators may even disable it.

The good news is the full range of behaviors is understood and observable, so an ICE Agent is able to confirm it created a NAT mapping, and the attributes of the mapping.

The document that describes these behaviors is RFC 4787.

Creating a mapping

Creating a mapping is the easiest part. When you send a packet to an address outside your network, a mapping is created! A NAT mapping is just a temporary

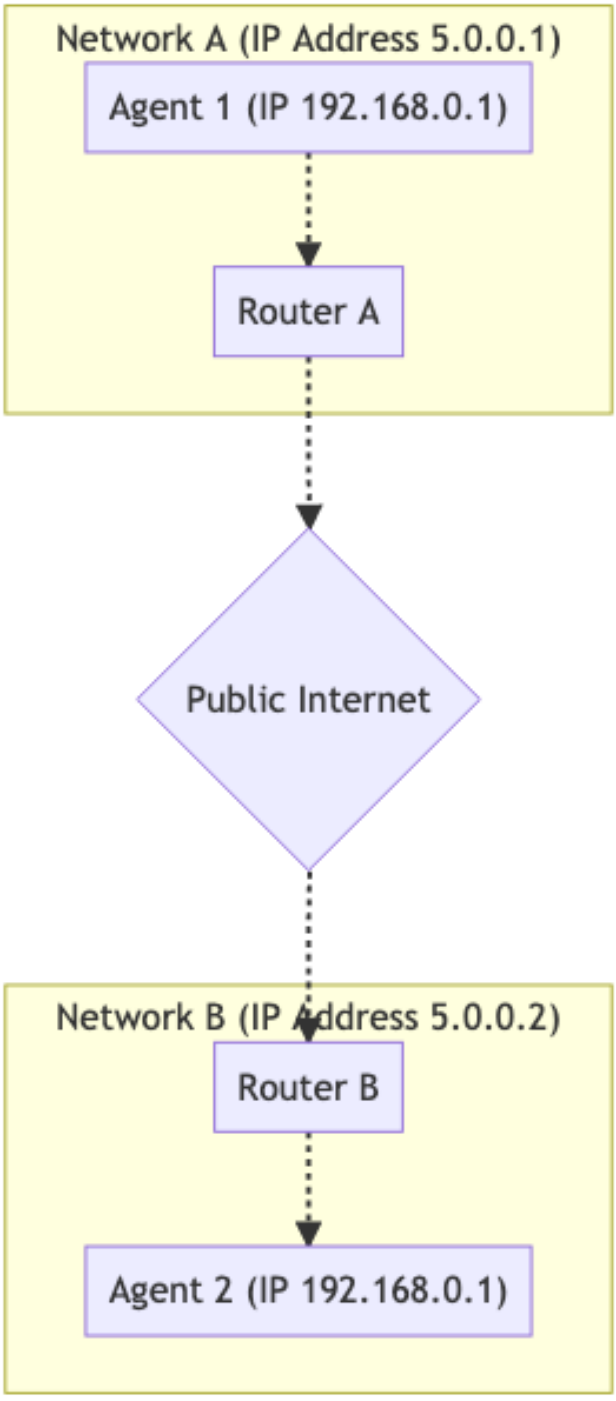


Figure 3: NAT mapping

public IP and port that is allocated by your NAT. The outbound message will be rewritten to have its source address given by the newly mapping address. If a message is sent to the mapping, it will be automatically routed back to the host inside the NAT that created it. The details around mappings is where it gets complicated.

Mapping Creation Behaviors

Mapping creation falls into three different categories:

Endpoint-Independent Mapping One mapping is created for each sender inside the NAT. If you send two packets to two different remote addresses, the NAT mapping will be re-used. Both remote hosts would see the same source IP and port. If the remote hosts respond, it would be sent back to the same local listener.

This is the best-case scenario. For a call to work, at least one side MUST be of this type.

Address Dependent Mapping A new mapping is created every time you send a packet to a new address. If you send two packets to different hosts, two mappings will be created. If you send two packets to the same remote host but different destination ports, a new mapping will NOT be created.

Address and Port Dependent Mapping A new mapping is created if the remote IP or port is different. If you send two packets to the same remote host, but different destination ports, a new mapping will be created.

Mapping Filtering Behaviors

Mapping filtering is the rules around who is allowed to use the mapping. They fall into three similar classifications:

Endpoint-Independent Filtering Anyone can use the mapping. You can share the mapping with multiple other peers, and they could all send traffic to it.

Address Dependent Filtering Only the host the mapping was created for can use the mapping. If you send a packet to host `A` it can respond with as many packets as it wants. If host `B` attempts to send a packet to that mapping, it will be ignored.

Address and Port Dependent Filtering Only the host and port for which the mapping was created for can use that mapping. If you send a packet to host `A:5000` it can respond with as many packets as it wants. If host `A:5001` attempts to send a packet to that mapping, it will be ignored.

Mapping Refresh

It is recommended that if a mapping is unused for 5 minutes it should be destroyed. This is entirely up to the ISP or hardware manufacturer.

STUN

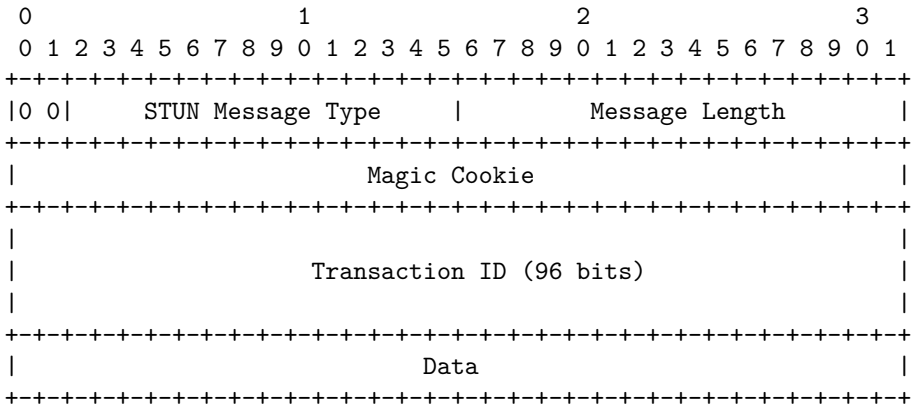
STUN (Session Traversal Utilities for NAT) is a protocol that was created just for working with NATs. This is another technology that pre-dates WebRTC (and ICE!). It is defined by RFC 5389, which also defines the STUN packet structure. The STUN protocol is also used by ICE/TURN.

STUN is useful because it allows the programmatic creation of NAT Mappings. Before STUN, we were able to create a NAT mapping, but we had no idea what the IP and port of it was! STUN not only gives you the ability to create a mapping, but also gives you the details so that you can share them with others, so they can send traffic back to you via the mapping you just created.

Let's start with a basic description of STUN. Later, we will expand on TURN and ICE usage. For now, we are just going to describe the Request/Response flow to create a mapping. Then we will talk about how to get the details of it to share with others. This is the process that happens when you have a `stun:` server in your ICE URLs for a WebRTC PeerConnection. In a nutshell, STUN helps an endpoint behind a NAT figure out what mapping was created by asking a STUN server outside NAT to report what it observes.

Protocol Structure

Every STUN packet has the following structure:



STUN Message Type Each STUN packet has a type. For now, we only care about the following:

- Binding Request - `0x0001`

- Binding Response - `0x0101`

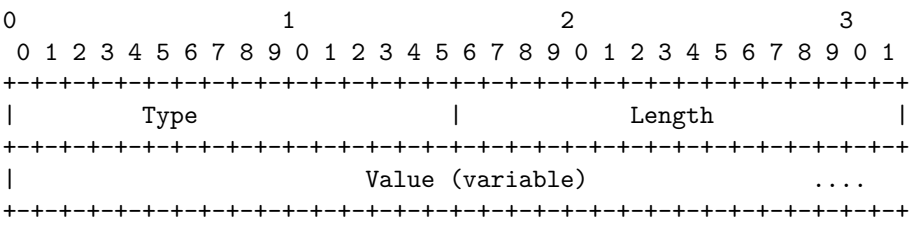
To create a NAT mapping we make a `Binding Request`. Then the server responds with a `Binding Response`.

Message Length This is how long the `Data` section is. This section contains arbitrary data that is defined by the `Message Type`.

Magic Cookie The fixed value `0x2112A442` in network byte order, it helps distinguish STUN traffic from other protocols.

Transaction ID A 96-bit identifier that uniquely identifies a request/response. This helps you pair up your requests and responses.

Data Data will contain a list of STUN attributes. A STUN Attribute has the following structure:



The `STUN Binding Request` uses no attributes. This means a `STUN Binding Request` contains only the header.

The `STUN Binding Response` uses a `XOR-MAPPED-ADDRESS (0x0020)`. This attribute contains an IP and port. This is the IP and port of the NAT mapping that is created!

Create a NAT Mapping

Creating a NAT mapping using STUN just takes sending one request! You send a `STUN Binding Request` to the STUN Server. The STUN Server then responds with a `STUN Binding Response`. This `STUN Binding Response` will contain the `Mapped Address`. The `Mapped Address` is how the STUN Server sees you and is your `NAT mapping`. The `Mapped Address` is what you would share if you wanted someone to send packets to you.

People will also call the `Mapped Address` your `Public IP` or `Server Reflexive Candidate`.

Determining NAT Type

Unfortunately, the `Mapped Address` might not be useful in all cases. If it is `Address Dependent`, only the STUN server can send traffic back to you. If you

shared it and another peer tried to send messages in they will be dropped. This makes it useless for communicating with others. You may find the `Address Dependent` case is in fact solvable, if the STUN server can also forward packets for you to the peer! This leads us to the solution using TURN below.

RFC 5780 defines a method for running a test to determine your NAT Type. This is useful because you would know ahead of time if direct connectivity is possible.

TURN

TURN (Traversal Using Relays around NAT) is defined in RFC 5766 is the solution when direct connectivity isn't possible. It could be because you have two NAT Types that are incompatible, or maybe can't speak the same protocol! TURN can also be used for privacy purposes. By running all your communication through TURN you obscure the client's actual address.

TURN uses a dedicated server. This server acts as a proxy for a client. The client connects to a TURN Server and creates an `Allocation`. By creating an allocation, a client gets a temporary IP/Port/Protocol that can be used to send traffic back to the client. This new listener is known as the `Relayed Transport Address`. Think of it as a forwarding address, you give this out so that others can send you traffic via TURN! For each peer you give the `Relay Transport Address` to, you must create a new `Permission` to allow communication with you.

When you send outbound traffic via TURN it is sent via the `Relayed Transport Address`. When a remote peer gets traffic they see it coming from the TURN Server.

TURN Lifecycle

The following is everything that a client who wishes to create a TURN allocation has to do. Communicating with someone who is using TURN requires no changes. The other peer gets an IP and port, and they communicate with it like any other host.

Allocations Allocations are at the core of TURN. An `allocation` is basically a "TURN Session". To create a TURN allocation you communicate with the TURN `Server Transport Address` (usually port `3478`).

When creating an allocation, you need to provide the following: * Username/Password - Creating TURN allocations require authentication. * Allocation Transport - The `Relayed Transport Address`, can be UDP or TCP. * Even-Port - You can request sequential ports for multiple allocations, not relevant for WebRTC.

If the request succeeded, you get a response with the TURN Server with

the following STUN Attributes in the Data section: * `XOR-MAPPED-ADDRESS` - `Mapped Address` of the `TURN Client`. When someone sends data to the `Relayed Transport Address` this is where it is forwarded to. * `RELAYED-ADDRESS` - This is the address that you give out to other clients. If someone sends a packet to this address, it is relayed to the TURN client. * `LIFETIME` - How long until this TURN Allocation is destroyed. You can extend the lifetime by sending a `Refresh` request.

Permissions A remote host can't send into your `Relayed Transport Address` until you create a permission for them. When you create a permission, you are telling the TURN server that this IP and port is allowed to send inbound traffic.

The remote host needs to give you the IP and port as it appears to the TURN server. This means it should send a `STUN Binding Request` to the TURN Server. A common error case is that a remote host will send a `STUN Binding Request` to a different server. They will then ask you to create a permission for this IP.

Let's say you want to create a permission for a host behind a `Address Dependent Mapping`. If you generate the `Mapped Address` from a different TURN server, all inbound traffic will be dropped. Every time they communicate with a different host it generates a new mapping. Permissions expire after 5 minutes if they are not refreshed.

SendIndication/ChannelData These two messages are for the TURN Client to send messages to a remote peer.

SendIndication is a self-contained message. Inside it is the data you wish to send, and who you wish to send it to. This is wasteful if you are sending a lot of messages to a remote peer. If you send 1,000 messages you will repeat their IP Address 1,000 times!

ChannelData allows you to send data, but not repeat an IP Address. You create a Channel with an IP and port. You then send with the ChannelId, and the IP and port will be populated server side. This is the better choice if you are sending a lot of messages.

Refreshing Allocations will destroy themselves automatically. The TURN Client must refresh them sooner than the `LIFETIME` given when creating the allocation.

TURN Usage

TURN Usage exists in two forms. Usually, you have one peer acting as a "TURN Client" and the other side communicating directly. In some cases you might have TURN usage on both sides, for example because both clients are in networks

that block UDP and therefore the connection to the respective TURN servers happens via TCP.

These diagrams help illustrate what that would look like.

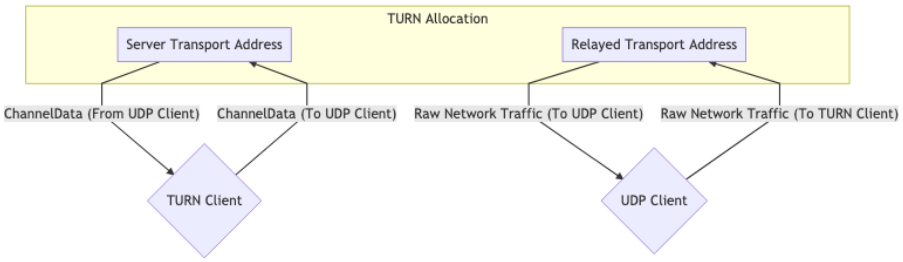


Figure 4: One TURN allocation

One TURN Allocation for Communication

Two TURN Allocations for Communication

ICE

ICE (Interactive Connectivity Establishment) is how WebRTC connects two Agents. Defined in RFC 8445, this is another technology that pre-dates WebRTC! ICE is a protocol for establishing connectivity. It determines all the possible routes between the two peers and then ensures you stay connected.

These routes are known as `Candidate Pairs`, which is a pairing of a local and remote transport address. This is where STUN and TURN come into play with ICE. These addresses can be your local IP Address plus a port, `NAT mapping`, or `Relayed Transport Address`. Each side gathers all the addresses they want to use, exchanges them, and then attempts to connect!

Two ICE Agents communicate using ICE ping packets (or formally called the connectivity checks) to establish connectivity. After connectivity is established, they can send whatever data they want. It will be like using a normal socket. These checks use the STUN protocol.

Creating an ICE Agent

An ICE Agent is either `Controlling` or `Controlled`. The `Controlling` Agent is the one that decides the selected `Candidate Pair`. Usually, the peer sending the offer is the controlling side.

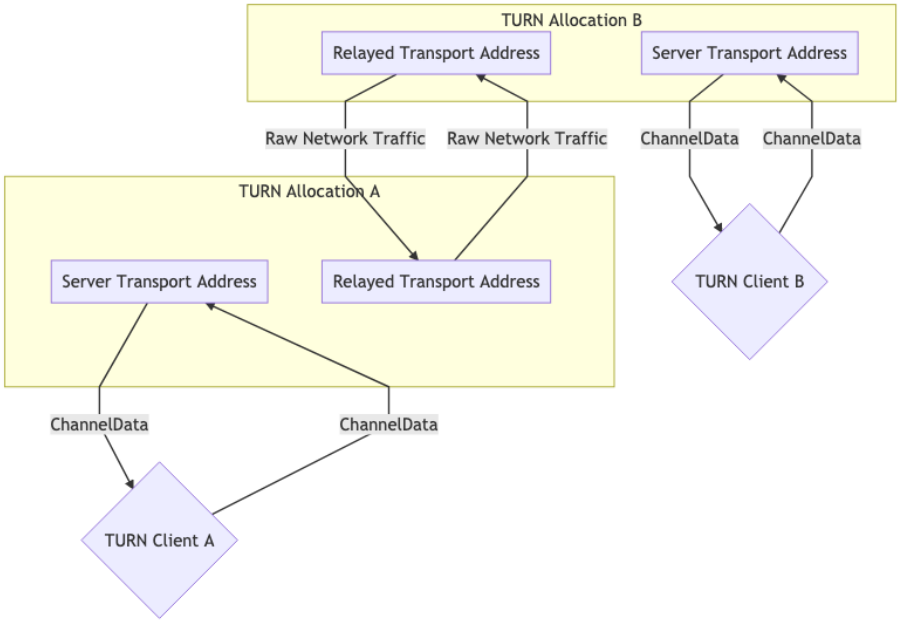


Figure 5: Two TURN allocations

Each side must have a `user fragment` and a `password`. These two values must be exchanged before connectivity checks can even begin. The `user fragment` is sent in plain text and is useful for demuxing multiple ICE Sessions. The `password` is used to generate a `MESSAGE-INTEGRITY` attribute. At the end of each STUN packet, there is an attribute that is a hash of the entire packet using the `password` as a key. This is used to authenticate the packet and ensure it hasn't been tampered with.

For WebRTC, all these values are distributed via the `Session Description` as described in the previous chapter.

Candidate Gathering

We now need to gather all the possible addresses we are reachable at. These addresses are known as candidates. These candidates are also distributed via the `Session Description`.

Host A Host candidate is listening directly on a local interface. This can either be UDP or TCP.

mDNS An mDNS candidate is similar to a host candidate, but the IP address is obscured. Instead of informing the other side about your IP address, you give them a UUID as the hostname. You then set up a multicast listener, and respond if anyone requests the UUID you published.

If you are in the same network as the agent, you can find each other via Multicast. If you are not in the same network, you will be unable to connect (unless the network administrator explicitly configured the network to allow Multicast packets to traverse).

This is useful for privacy purposes. A user could find out your local IP address via WebRTC with a Host candidate (without even trying to connect to you), but with an mDNS candidate, now they only get a random UUID.

Server Reflexive A Server Reflexive candidate is generated by doing a `STUN Binding Request` to a STUN Server.

When you get the `STUN Binding Response`, the `XOR-MAPPED-ADDRESS` is your Server Reflexive Candidate.

Peer Reflexive A Peer Reflexive candidate is when you get an inbound request from an address that isn't known to you. Since ICE is an authenticated protocol, you know the traffic is valid. This just means the remote peer is communicating with you from an address it didn't know about.

This commonly happens when a `Host Candidate` communicates with a `Server Reflexive Candidate`. A new `NAT mapping` was created because you are communicating outside your subnet. Remember we said the connectivity checks are

in fact STUN packets? The format of STUN response naturally allows a peer to report back the peer-reflexive address.

Relay A Relay Candidate is generated by using a TURN Server.

After the initial handshake with the TURN Server you are given a `RELAYED-ADDRESS`, this is your Relay Candidate.

Connectivity Checks

We now know the remote agent's `user fragment`, `password`, and candidates. We can now attempt to connect! Every candidate is paired with each other. So if you have 3 candidates on each side, you now have 9 candidate pairs.

Visually it looks like this:

Candidate Selection

The Controlling and Controlled Agent both start sending traffic on each pair. This is needed if one Agent is behind an `Address Dependent Mapping`, this will cause a `Peer Reflexive Candidate` to be created.

Each `Candidate Pair` that saw network traffic is then promoted to a `Valid Candidate` pair. The Controlling Agent then takes one `Valid Candidate` pair and nominates it. This becomes the `Nominated Pair`. The Controlling and Controlled Agent then attempt one more round of bi-directional communication. If that succeeds, the `Nominated Pair` becomes the `Selected Candidate Pair`! This pair is then used for the rest of the session.

Restarts

If the `Selected Candidate Pair` stops working for any reason (NAT mapping expires, TURN Server crashes) the ICE Agent will go to `Failed` state. Both agents can be restarted and will do the whole process all over again.

What security does WebRTC have?

Every WebRTC connection is authenticated and encrypted. You can be confident that a 3rd party can't see what you are sending or insert bogus messages. You can also be sure that the WebRTC Agent that generated the Session Description is the one you are communicating with.

It is very important that no one tampers with those messages. It is ok if a 3rd party reads the Session Description in transit. However, WebRTC has no protection against it being modified. An attacker could perform a man-in-the-middle attack on you by changing the ICE Candidates and update the Certificate Fingerprint.

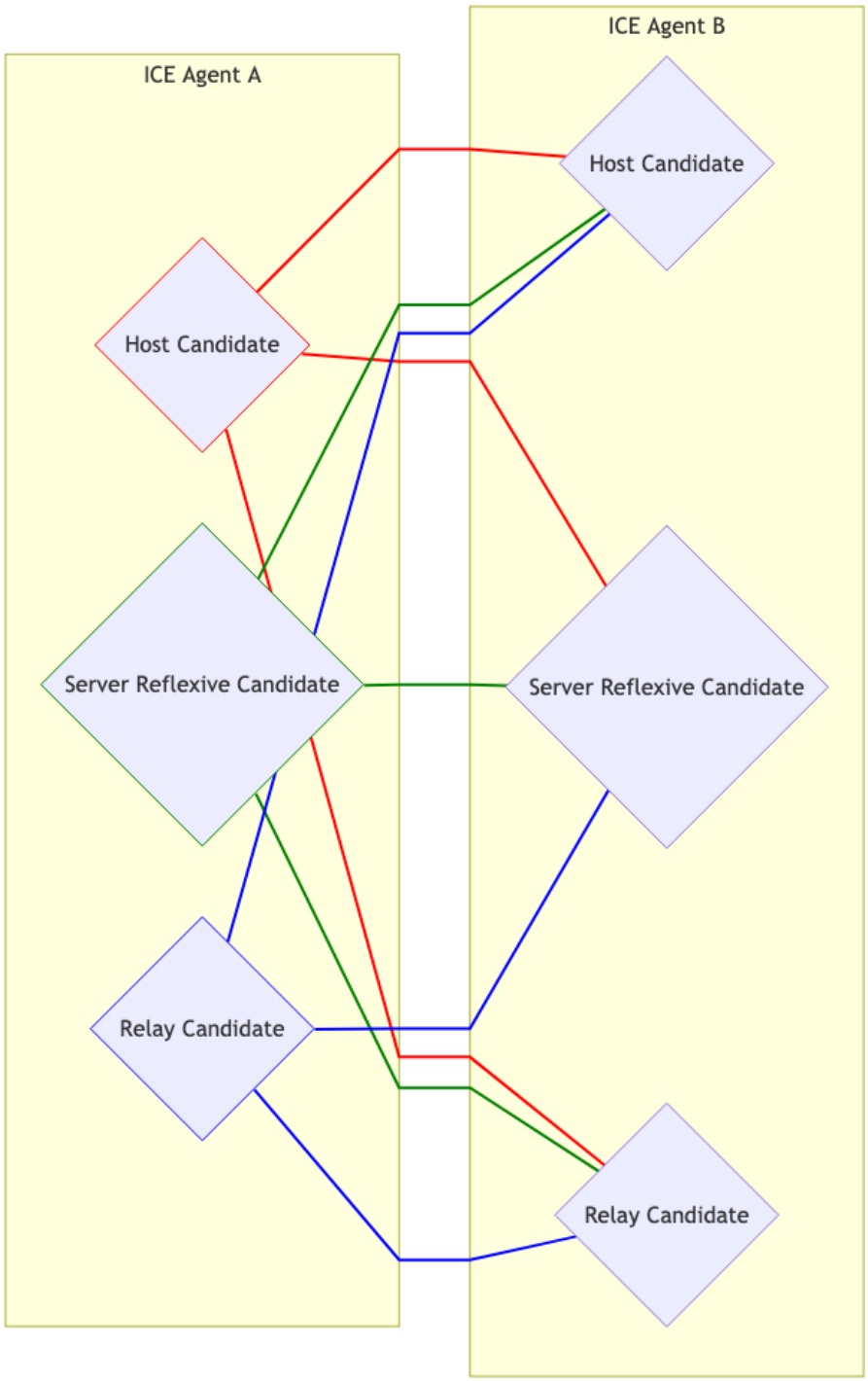


Figure 6: Connectivity checks

How does it work?

WebRTC uses two pre-existing protocols, Datagram Transport Layer Security (DTLS) and the Secure Real-time Transport Protocol (SRTP).

DTLS allows you to negotiate a session and then exchange data securely between two peers. It is a sibling of TLS, the same technology that powers HTTPS, but DTLS uses UDP instead of TCP as the transport layer. That means the protocol has to handle unreliable delivery. SRTP is specifically designed for exchanging media securely. There are some optimizations we can make by using it instead of DTLS.

DTLS is used first. It does a handshake over the connection provided by ICE. DTLS is a client/server protocol, so one side needs to start the handshake. The Client/Server roles are chosen during signaling. During the DTLS handshake, both sides offer a certificate. After the handshake is complete, this certificate is compared to the certificate hash in the Session Description. This is to ensure that the handshake happened with the WebRTC Agent you expected. The DTLS connection is then available to be used for DataChannel communication.

To create an SRTP session we initialize it using the keys generated by DTLS. SRTP does not have a handshake mechanism, so has to be bootstrapped with external keys. Once this is done, media can be exchanged that is encrypted using SRTP!

Security 101

To understand the technology presented in this chapter you will need to understand these terms first. Cryptography is a tricky subject, so it would be worth consulting other sources as well!

Plaintext and Ciphertext Plaintext is the input to a cipher. Ciphertext is the output of a cipher.

Cipher Cipher is a series of steps that takes plaintext to ciphertext. The cipher can then be reversed, so you can take your ciphertext back to plaintext. A cipher usually has a key to change its behavior. Another term for this is encrypting and decrypting.

A simple cipher is ROT13. Each letter is moved 13 characters forward. To undo the cipher you move 13 characters backward. The plaintext `HELLO` would become the ciphertext `URYYB`. In this case, the Cipher is ROT, and the key is 13.

Hash functions A hash function is a one-way process that generates a digest. Given an input, it generates the same output every time. It is important that the output is *not* reversible. If you have an output, you should not be able to determine its input. Hashing is useful when you want to confirm that a message hasn't been tampered with.

A simple hash function would be to only take every other letter. `HELLO` would become `HLO`. You can't assume `HELLO` was the input, but you can confirm that `HELLO` would be a match to the hash digest.

Public/Private Key Cryptography Public/Private Key Cryptography describes the type of ciphers that DTLS and SRTP uses. In this system, you have two keys, a public and private key. The public key is for encrypting messages and is safe to share. The private key is for decrypting, and should never be shared. It is the only key that can decrypt the messages encrypted with the public key.

Diffie–Hellman exchange Diffie–Hellman exchange allows two users who have never met before to create a shared secret securely over the internet. User `A` can send a secret to User `B` without worrying about eavesdropping. This depends on the difficulty of breaking the discrete logarithm problem. You don't need to fully understand how this works, but it helps to know this is what makes the DTLS handshake possible.

Wikipedia has an example of this in action here.

Pseudorandom Function A Pseudorandom Function (PRF) is a pre-defined function to generate a value that appears random. It may take multiple inputs and generate a single output.

Key Derivation Function Key Derivation is a type of Pseudorandom Function. Key Derivation is a function that is used to make a key stronger. One common pattern is key stretching.

Let's say you are given a key that is 8 bytes. You could use a KDF to make it stronger.

Nonce A nonce is an additional input to a cipher. This is used so that you can get different output from the cipher, even if you are encrypting the same message multiple times.

If you encrypt the same message 10 times, the cipher will give you the same ciphertext 10 times. By using a nonce you can get different input, while still using the same key. It is important you use a different nonce for each message! Otherwise, it negates much of the value.

Message Authentication Code A Message Authentication Code is a hash that is placed at the end of a message. A MAC proves that the message comes from the user you expected.

If you don't use a MAC, an attacker could insert invalid messages. After decrypting you would just have garbage because they don't know the key.

Key Rotation Key Rotation is the practice of changing your key on an interval. This makes a stolen key less impactful. If a key is stolen or leaked, fewer data can be decrypted.

DTLS

DTLS (Datagram Transport Layer Security) allows two peers to establish secure communication with no pre-existing configuration. Even if someone is eavesdropping on the conversation, they will not be able to decrypt the messages.

For a DTLS Client and a Server to communicate, they need to agree on a cipher and the key. They determine these values by doing a DTLS handshake. During the handshake, the messages are in plaintext. When a DTLS Client/Server has exchanged enough details to start encrypting it sends a `Change Cipher Spec`. After this message, each subsequent message will be encrypted!

Packet Format

Every DTLS packet starts with a header.

Content Type You can expect the following types:

- `20` - Change Cipher Spec
- `22` - Handshake
- `23` - Application Data

`Handshake` is used to exchange the details to start the session. `Change Cipher Spec` is used to notify the other side that everything will be encrypted. `Application Data` are the encrypted messages.

Version Version can either be `0x0000feff` (DTLS v1.0) or `0x0000fefd` (DTLS v1.2) there is no v1.1.

Epoch The epoch starts at `0`, but becomes `1` after a `Change Cipher Spec`. Any message with a non-zero epoch is encrypted.

Sequence Number Sequence Number is used to keep messages in order. Each message increases the Sequence Number. When the epoch is incremented, the Sequence Number starts over.

Length and Payload The Payload is `Content Type` specific. For a `Application Data` the `Payload` is the encrypted data. For `Handshake` it will be different depending on the message. The length is for how big the `Payload` is.

Handshake State Machine

During the handshake, the Client/Server exchanges a series of messages. These messages are grouped into flights. Each flight may have multiple messages in it (or just one). A Flight is not complete until all the messages in the flight have been received. We will describe the purpose of each message in greater detail below.

ClientHello ClientHello is the initial message sent by the client. It contains a list of attributes. These attributes tell the server the ciphers and features the client supports. For WebRTC this is how we choose the SRTP Cipher as well. It also contains random data that will be used to generate the keys for the session.

HelloVerifyRequest HelloVerifyRequest is sent by the server to the client. It is to make sure that the client intended to send the request. The Client then re-sends the ClientHello, but with a token provided in the HelloVerifyRequest.

ServerHello ServerHello is the response by the server for the configuration of this session. It contains what cipher will be used when this session is over. It also contains the server random data.

Certificate Certificate contains the certificate for the Client or Server. This is used to uniquely identify who we were communicating with. After the handshake is over we will make sure this certificate when hashed matches the fingerprint in the `SessionDescription`.

ServerKeyExchange/ClientKeyExchange These messages are used to transmit the public key. On startup, the client and server both generate a keypair. After the handshake these values will be used to generate the `Pre-Master Secret`.

CertificateRequest A CertificateRequest is sent by the server notifying the client that it wants a certificate. The server can either Request or Require a certificate.

ServerHelloDone ServerHelloDone notifies the client that the server is done with the handshake.

CertificateVerify CertificateVerify is how the sender proves that it has the private key sent in the Certificate message.

ChangeCipherSpec ChangeCipherSpec informs the receiver that everything sent after this message will be encrypted.

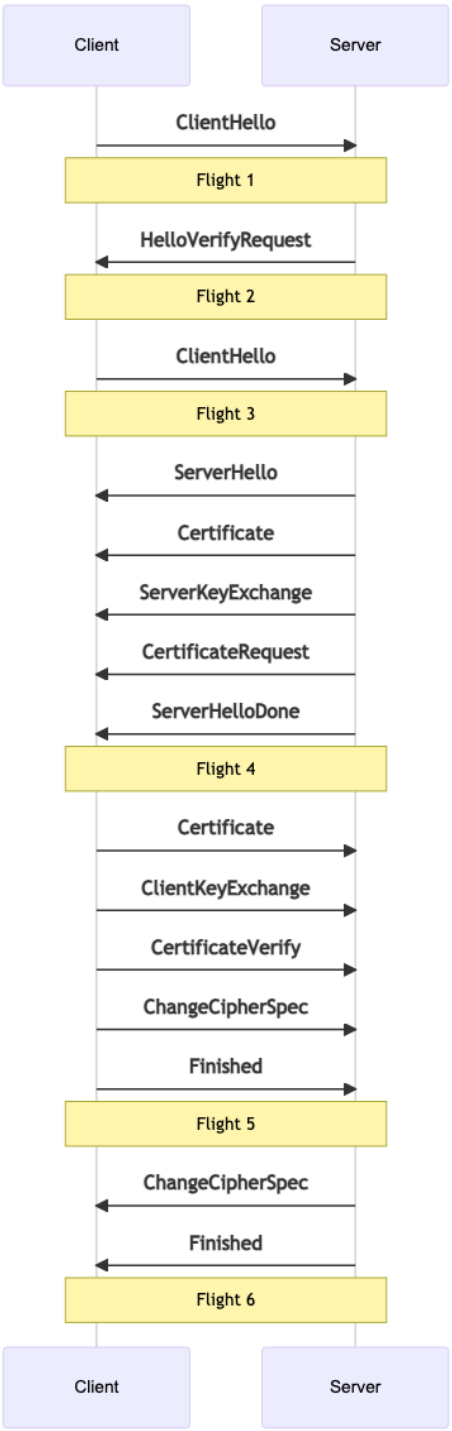


Figure 7: Handshake

Finished Finished is encrypted and contains a hash of all messages. This is to assert that the handshake was not tampered with.

Key Generation

After the Handshake is complete, you can start sending encrypted data. The Cipher was chosen by the server and is in the ServerHello. How was the key chosen though?

First we generate the `Pre-Master Secret`. To obtain this value Diffie–Hellman is used on the keys exchanged by the `ServerKeyExchange` and `ClientKeyExchange`. The details differ depending on the chosen Cipher.

Next the `Master Secret` is generated. Each version of DTLS has a defined `Pseudorandom function`. For DTLS 1.2 the function takes the `Pre-Master Secret` and random values in the `ClientHello` and `ServerHello`. The output from running the `Pseudorandom Function` is the `Master Secret`. The `Master Secret` is the value that is used for the Cipher.

Exchanging ApplicationData

The workhorse of DTLS is `ApplicationData`. Now that we have an initialized cipher, we can start encrypting and sending values.

`ApplicationData` messages use a DTLS header as described earlier. The `Payload` is populated with ciphertext. You now have a working DTLS Session and can communicate securely.

DTLS has many more interesting features like renegotiation. They are not used by WebRTC, so they will not be covered here.

SRTP

SRTP is a protocol designed specifically for encrypting RTP packets. To start an SRTP session you specify your keys and cipher. Unlike DTLS it has no handshake mechanism. All the configuration and keys were generated during the DTLS handshake.

DTLS provides a dedicated API to export the keys to be used by another process. This is defined in RFC 5705.

Session Creation

SRTP defines a Key Derivation Function that is used on the inputs. When creating an SRTP Session the inputs are run through this to generate our keys for our SRTP Cipher. After this you can move on to processing media.

Exchanging Media

Each RTP packet has a 16 bit SequenceNumber. These Sequence Numbers are used to keep packets in order, like a Primary Key. During a call these will rollover. SRTP keeps track of it and calls this the rollover counter.

When encrypting a packet SRTP uses the rollover counter and sequence number as a nonce. This is to ensure that even if you send the same data twice, the ciphertext will be different. This is important to prevent an attacker from identifying patterns or attempting a replay attack.

Why is networking so important in Real-time communication?

Networks are the limiting factor in Real-time communication. In an ideal world we would have unlimited bandwidth and packets would arrive instantaneously. This isn't the case though. Networks are limited, and the conditions could change at anytime. Measuring and observing network conditions is also a difficult problem. You can get different behaviors depending on hardware, software and the configuration of it.

Real-time communication also poses a problem that doesn't exist in most other domains. For a web developer it isn't fatal if your website is slower on some networks. As long as all the data arrives, users are happy. With WebRTC, if your data is late it is useless. No one cares about what was said in a conference call 5 seconds ago. So when developing a realtime communication system, you have to make a trade-off. What is my time limit, and how much data can I send?

This chapter covers the concepts that apply to both data and media communication. In later chapters we go beyond the theoretical and discuss how WebRTC's media and data subsystems solve these problems.

What are the attributes of the network that make it difficult?

Code that effectively works across all networks is complicated. You have lots of different factors, and they can all affect each other subtly. These are the most common issues that developers will encounter.

Bandwidth Bandwidth is the maximum rate of data that can be transferred across a given path. It is important to remember this isn't a static number either. The bandwidth will change along the route as more (or less) people use it.

Transmission Time and Round Trip Time Transmission Time is how long it takes for a packet to arrive to its destination. Like Bandwidth this isn't constant. The Transmission Time can fluctuate at anytime.

```
transmission_time = receive_time - send_time
```

To compute transmission time, you need clocks on sender and receiver synchronized with millisecond precision. Even a small deviation would produce an unreliable transmission time measurement. Since WebRTC is operating in highly heterogeneous environments, it is next to impossible to rely on perfect time synchronization between hosts.

Round-trip time measurement is a workaround for imperfect clock synchronization.

Instead of operating on distributed clocks a WebRTC peer sends a special packet with its own timestamp `sendertime1`. A cooperating peer receives the packet and reflects the timestamp back to the sender. Once the original sender gets the reflected time it subtracts the timestamp `sendertime1` from the current time `sendertime2`. This time delta is called "round-trip propagation delay" or more commonly round-trip time.

```
rtt = sendertime2 - sendertime1
```

Half of the round trip time is considered to be a good enough approximation of transmission time. This workaround is not without drawbacks. It makes the assumption that it takes an equal amount of time to send and receive packets. However on cellular networks, send and receive operations may not be time-symmetrical. You may have noticed that upload speeds on your phone are almost always lower than download speeds.

```
transmission_time = rtt/2
```

The technicalities of round-trip time measurement are described in greater detail in RTCP Sender and Receiver Reports chapter.

Jitter Jitter is the fact that `Transmission Time` may vary for each packet. Your packets could be delayed, but then arrive in bursts.

Packet Loss Packet Loss is when messages are lost in transmission. The loss could be steady, or it could come in spikes. This could be because of the network type like satellite or Wi-Fi. Or it could be introduced by the software along the way.

Maximum Transmission Unit Maximum Transmission Unit is the limit on how large a single packet can be. Networks don't allow you to send one giant message. At the protocol level, messages might have to be split into multiple smaller packets.

The MTU will also differ depending on what network path you take. You can use a protocol like Path MTU Discovery to figure out the largest packet size you can send.

Congestion

Congestion is when the limits of the network have been reached. This is usually because you have reached the peak bandwidth that the current route can handle. Or it could be operator imposed like hourly limits your ISP configures.

Congestion exhibits itself in many different ways. There is no standardized behavior. In most cases when congestion is reached the network will drop excess packets. In other cases the network will buffer. This will cause the Transmission Time for your packets to increase. You could also see more jitter as your network becomes congested. This is a rapidly changing area and new algorithms for congestion detection are still being written.

Dynamic

Networks are incredibly dynamic and conditions can change rapidly. During a call you may send and receive hundreds of thousands of packets. Those packets will be traveling through multiple hops. Those hops will be shared by millions of other users. Even in your local network you could have HD movies being downloaded or maybe a device decides to download a software update.

Having a good call isn't as simple as measuring your network on startup. You need to be constantly evaluating. You also need to handle all the different behaviors that come from a multitude of network hardware and software.

Solving Packet Loss

Handling packet loss is the first problem to solve. There are multiple ways to solve it, each with their own benefits. It depends on what you are sending and how latency tolerant you are. It is also important to note that not all packet loss is fatal. Losing some video might not be a problem, the human eye might not even able to perceive it. Losing a users text messages are fatal.

Let's say you send 10 packets, and packets 5 and 6 are lost. Here are the ways you can solve it.

Acknowledgments

Acknowledgments is when the receiver notifies the sender of every packet they have received. The sender is aware of packet loss when it gets an acknowledgment for a packet twice that isn't final. When the sender gets an `ACK` for packet 4 twice, it knows that packet 5 has not been seen yet.

Selective Acknowledgments

Selective Acknowledgments is an improvement upon Acknowledgments. A receiver can send a `SACK` that acknowledges multiple packets and notifies the sender of gaps. Now the sender can get a `SACK` for packet 4 and 7. It then knows it needs to re-send packets 5 and 6.

Negative Acknowledgments

Negative Acknowledgments solve the problem the opposite way. Instead of notifying the sender what it has received, the receiver notifies the sender what has been lost. In our case a `NACK` will be sent for packets 5 and 6. The sender only knows packets the receiver wishes to have sent again.

Forward Error Correction

Forward Error Correction fixes packet loss pre-emptively. The sender sends redundant data, meaning a lost packet doesn't affect the final stream. One popular algorithm for this is Reed–Solomon error correction.

This reduces the latency/complexity of sending and handling Acknowledgments. Forward Error Correction is a waste of bandwidth if the network you are in has zero loss.

Solving Jitter

Jitter is present in most networks. Even inside a LAN you have many devices sending data at fluctuating rates. You can easily observe jitter by pinging another device with the `ping` command and noticing the fluctuations in round-trip latency.

To solve jitter, clients use a JitterBuffer. The JitterBuffer ensures a steady delivery time of packets. The downside is that JitterBuffer adds some latency to packets that arrive early. The upside is that late packets don't cause jitter. Imagine that during a call, you see the following packet arrival times:

```
* 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
```

In this case, around 1.8 ms would be a good choice. Packets that arrive late will use our window of latency. Packets that arrive early will be delayed a bit and can fill the window depleted by late packets. This means we no longer have stuttering and provide a smooth delivery rate for the client.

JitterBuffer operation

Every packet gets added to the jitter buffer as soon as it is received. Once there are enough packets to reconstruct the frame, packets that make up the frame are released from the buffer and emitted for decoding. The decoder, in turn, decodes

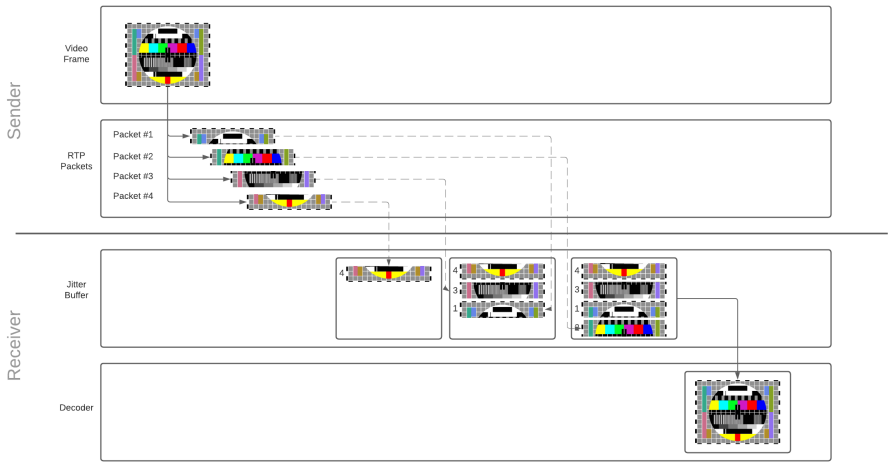


Figure 8: JitterBuffer

and draws the video frame on the user's screen. Since the jitter buffer has a limited capacity, packets that stay in the buffer for too long will be discarded.

Read more on how video frames are converted to RTP packets, and why reconstruction is necessary in the media communication chapter.

`jitterBufferDelay` provides a great insight into your network performance and its influence on playback smoothness. It is a part of WebRTC statistics API relevant to the receiver's inbound stream. The delay defines the amount of time video frames spend in the jitter buffer before being emitted for decoding. A long jitter buffer delay means your network is highly congested.

Detecting Congestion

Before we can even resolve congestion, we need to detect it. To detect it we use a congestion controller. This is a complicated subject, and is still rapidly changing. New algorithms are still being published and tested. At a high level they all operate the same. A congestion controller provides bandwidth estimates given some inputs. These are some possible inputs:

- **Packet Loss** - Packets are dropped as the network becomes congested.
- **Jitter** - As network equipment becomes more overloaded packets queuing will cause the times to be erratic.
- **Round Trip Time** - Packets take longer to arrive when congested. Unlike jitter, the Round Trip Time just keeps increasing.
- **Explicit Congestion Notification** - Newer networks may tag packets as at risk for being dropped to relieve congestion.

These values need to be measured continuously during the call. Utilization of the

network may increase or decrease, so the available bandwidth could constantly be changing.

Resolving Congestion

Now that we have an estimated bandwidth we need to adjust what we are sending. How we adjust depends on what kind of data we want to send.

Sending Slower

Limiting the speed at which you send data is the first solution to preventing congestion. The Congestion Controller gives you an estimate, and it is the sender's responsibility to rate limit.

This is the method used for most data communication. With protocols like TCP this is all done by the operating system and completely transparent to both users and developers.

Sending Less

In some cases we can send less information to satisfy our limits. We also have hard deadlines on the arrival of our data, so we can't send slower. These are the constraints that Real-time media falls under.

If we don't have enough bandwidth available, we can lower the quality of video we send. This requires a tight feedback loop between your video encoder and congestion controller.

What do I get from WebRTC's media communication?

WebRTC allows you to send and receive an unlimited amount of audio and video streams. You can add and remove these streams at anytime during a call. These streams could all be independent, or they could be bundled together! You could send a video feed of your desktop, and then include audio and video from your webcam.

The WebRTC protocol is codec agnostic. The underlying transport supports everything, even things that don't exist yet! However, the WebRTC Agent you are communicating with may not have the necessary tools to accept it.

WebRTC is also designed to handle dynamic network conditions. During a call your bandwidth might increase, or decrease. Maybe you suddenly experience lots of packet loss. The protocol is designed to handle all of this. WebRTC responds to network conditions and tries to give you the best experience possible with the resources available.

How does it work?

WebRTC uses two pre-existing protocols RTP and RTCP, both defined in RFC 1889.

RTP (Real-time Transport Protocol) is the protocol that carries the media. It was designed to allow for real-time delivery of video. It does not stipulate any rules around latency or reliability, but gives you the tools to implement them. RTP gives you streams, so you can run multiple media feeds over one connection. It also gives you the timing and ordering information you need to feed a media pipeline.

RTCP (RTP Control Protocol) is the protocol that communicates metadata about the call. The format is very flexible and allows you to add any metadata you want. This is used to communicate statistics about the call. It is also used to handle packet loss and to implement congestion control. It gives you the bi-directional communication necessary to respond to changing network conditions.

Latency vs Quality

Real-time media is about making trade-offs between latency and quality. The more latency you are willing to tolerate, the higher quality video you can expect.

Real World Limitations

These constraints are all caused by the limitations of the real world. They are all characteristics of your network that you will need to overcome.

Video is Complex

Transporting video isn't easy. To store 30 minutes of uncompressed 720 8-bit video you need about 110 GB. With those numbers, a 4-person conference call isn't going to happen. We need a way to make it smaller, and the answer is video compression. That doesn't come without downsides though.

Video 101

We aren't going to cover video compression in depth, but just enough to understand why RTP is designed the way it is. Video compression encodes video into a new format that requires fewer bits to represent the same video.

Lossy and Lossless compression

You can encode video to be lossless (no information is lost) or lossy (information may be lost). Because lossless encoding requires more data to be sent to a peer, making for a higher latency stream and more dropped packets, RTP typically uses lossy compression even though the video quality won't be as good.

Intra and Inter frame compression

Video compression comes in two types. The first is intra-frame. Intra-frame compression reduces the bits used to describe a single video frame. The same techniques are used to compress still pictures, like the JPEG compression method.

The second type is inter-frame compression. Since video is made up of many pictures we look for ways to not send the same information twice.

Inter-frame types

You then have three frame types:

- **I-Frame** - A complete picture, can be decoded without anything else.
- **P-Frame** - A partial picture, containing only changes from the previous picture.
- **B-Frame** - A partial picture, is a modification of previous and future pictures.

The following is visualization of the three frame types.

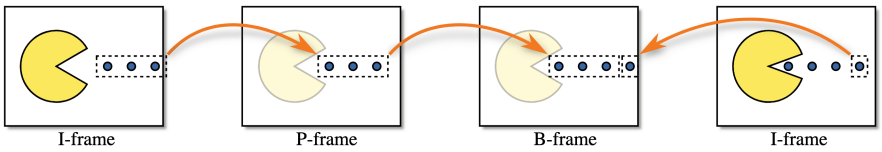


Figure 9: Frame types

Video is delicate

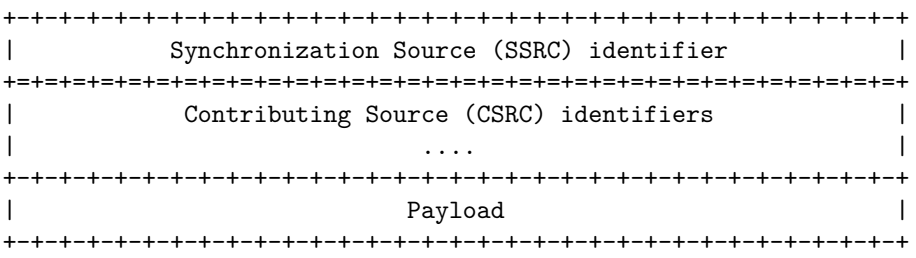
Video compression is incredibly stateful, making it difficult to transfer over the internet. What happens If you lose part of an I-Frame? How does a P-Frame know what to modify? As video compression gets more complex, this is becoming even more of a problem. Luckily RTP and RTCP have the solution.

RTP

Packet Format

Every RTP packet has the following structure:

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       Sequence Number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
```



Version (V) `Version` is always `2`

Padding (P) `Padding` is a bool that controls if the payload has padding.

The last byte of the payload contains a count of how many padding bytes were added.

Extension (X) If set, the RTP header will have extensions. This is described in greater detail below.

CSRC count (CC) The amount of `CSRC` identifiers that follow after the `SSRC`, and before the payload.

Marker (M) The marker bit has no pre-set meaning, and can be used however the user likes.

In some cases it is set when a user is speaking. It is also commonly used to mark a keyframe.

Payload Type (PT) `Payload Type` is a unique identifier for what codec is being carried by this packet.

For WebRTC the `Payload Type` is dynamic. VP8 in one call may be different from another. The offerer in the call determines the mapping of `Payload Types` to codecs in the `Session Description`.

Sequence Number `Sequence Number` is used for ordering packets in a stream. Every time a packet is sent the `Sequence Number` is incremented by one.

RTP is designed to be useful over lossy networks. This gives the receiver a way to detect when packets have been lost.

Timestamp The sampling instant for this packet. This is not a global clock, but how much time has passed in the media stream. Several RTP packages can have the same timestamp if they for example are all part of the same video frame.

Synchronization Source (SSRC) An `SSRC` is the unique identifier for this stream. This allows you to run multiple streams of media over a single RTP stream.

Contributing Source (CSRC) A list that communicates what `SSRC`es contributed to this packet.

This is commonly used for talking indicators. Let's say server side you combined multiple audio feeds into a single RTP stream. You could then use this field to say "Input stream A and C were talking at this moment".

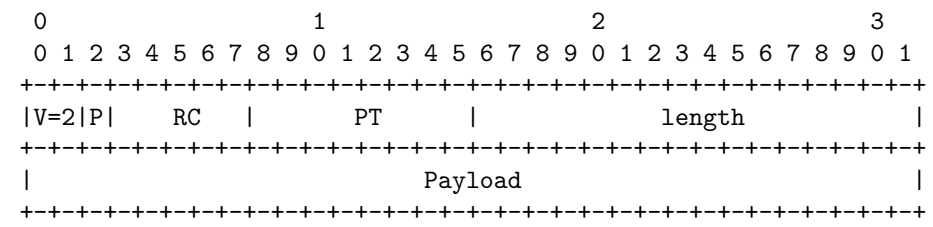
Payload The actual payload data. Might end with the count of how many padding bytes were added, if the padding flag is set.

Extensions

RTCP

Packet Format

Every RTCP packet has the following structure:



Version (V) `Version` is always 2.

Padding (P) `Padding` is a bool that controls if the payload has padding.

The last byte of the payload contains a count of how many padding bytes were added.

Reception Report Count (RC) The number of reports in this packet. A single RTCP packet can contain multiple events.

Packet Type (PT) Unique Identifier for what type of RTCP Packet this is. A WebRTC Agent doesn't need to support all these types, and support between Agents can be different. These are the ones you may commonly see though:

- `192` - Full INTRA-frame Request (`FIR`)
- `193` - Negative ACKnowledgements (`NACK`)
- `200` - Sender Report
- `201` - Receiver Report

- `205` - Generic RTP Feedback
- `206` - Payload Specific Feedback

The significance of these packet types will be described in greater detail below.

Full INTRA-frame Request (FIR) and Picture Loss Indication (PLI)

Both `FIR` and `PLI` messages serve a similar purpose. These messages request a full key frame from the sender. `PLI` is used when partial frames were given to the decoder, but it was unable to decode them. This could happen because you had lots of packet loss, or maybe the decoder crashed.

According to RFC 5104, `FIR` shall not be used when packets or frames are lost. That is `PLI`s job. `FIR` requests a key frame for reasons other than packet loss - for example when a new member enters a video conference. They need a full key frame to start decoding video stream, the decoder will be discarding frames until key frame arrives.

It is a good idea for a receiver to request a full key frame right after connecting, this minimizes the delay between connecting, and an image showing up on the user's screen.

`PLI` packets are a part of Payload Specific Feedback messages.

In practice, software that is able to handle both `PLI` and `FIR` packets will act the same way in both cases. It will send a signal to the encoder to produce a new full key frame.

Negative Acknowledgment

A `NACK` requests that a sender re-transmits a single RTP packet. This is usually caused by an RTP packet getting lost, but could also happen because it is late.

`NACK`s are much more bandwidth efficient than requesting that the whole frame get sent again. Since RTP breaks up packets into very small chunks, you are really just requesting one small missing piece. The receiver crafts an RTCP message with the SSRC and Sequence Number. If the sender does not have this RTP packet available to re-send, it just ignores the message.

Sender and Receiver Reports

These reports are used to send statistics between agents. This communicates the amount of packets actually received and jitter.

The reports can be used for diagnostics and congestion control.

How RTP/RTCP solve problems together

RTP and RTCP then work together to solve all the problems caused by networks. These techniques are still constantly changing!

Forward Error Correction

Also known as FEC. Another method of dealing with packet loss. FEC is when you send the same data multiple times, without it even being requested. This is done at the RTP level, or even lower with the codec.

If the packet loss for a call is steady then FEC is a much lower latency solution than NACK. The round trip time of having to request, and then re-transmit the missing packet can be significant for NACKs.

Adaptive Bitrate and Bandwidth Estimation

As discussed in the Real-time networking chapter, networks are unpredictable and unreliable. Bandwidth availability can change multiple times throughout a session. It is not uncommon to see available bandwidth change dramatically (orders of magnitude) within a second.

The main idea is to adjust encoding bitrate based on predicted, current, and future available network bandwidth. This ensures that video and audio signal of the best possible quality is transmitted, and the connection does not get dropped because of network congestion. Heuristics that model the network behavior and tries to predict it is known as Bandwidth estimation.

There is a lot of nuance to this, so let's explore in greater detail.

Communicating Network Status

The first roadblock with implementing Congestion Control is that UDP and RTP don't communicate network status. As a sender I have no idea when my packets are arriving or if they are arriving at all!

RTP/RTCP has three different solutions to this problem. They all have their pros and cons. What you use will depend on what clients you are working with, the type of topology you are working with, or even just how much development time you have available.

Receiver Reports

Receiver Reports are RTCP messages, the original way to communicate network status. You can find them in RFC 3550. They are sent on a schedule for each SSRC and contain the following fields:

- **Fraction Lost** - What percentage of packets have been lost since the last Receiver Report.
- **Cumulative Number of Packets Lost** - How many packets have been lost during the entire call.
- **Extended Highest Sequence Number Received** - What was the last Sequence Number received, and how many times has it rolled over.
- **Interarrival Jitter** - The rolling Jitter for the entire call.

- **Last Sender Report Timestamp** - Last known time on sender, used for round-trip time calculation.

Sender and Receiver reports (SR and RR) work together to compute round-trip time.

The sender includes its local time, `sendertime1` in SR. When the receiver gets an SR packet, it sends back RR. Among other things, the RR includes `sendertime1` just received from the sender. There will be a delay between receiving the SR and sending the RR. Because of that, the RR also includes a "delay since last sender report" time - `DLSR`. The `DLSR` is used to adjust the round-trip time estimate later on in the process. Once the sender receives the RR it subtracts `sendertime1` and `DLSR` from the current time `sendertime2`. This time delta is called round-trip propagation delay or round-trip time.

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

Round-trip time in plain English: - I send you a message with my clock's current reading, say it is 4:20pm, 42 seconds and 420 milliseconds. - You send me this same timestamp back. - You also include the time elapsed from reading my message to sending the message back, say 5 milliseconds. - Once I receive the time back, I look at the clock again. - Now my clock says 4:20pm, 42 seconds 690 milliseconds. - It means that it took 265 milliseconds (690 - 420 - 5) to reach you and return back to me. - Therefore, the round-trip time is 265 milliseconds.

TMMBR, TMMBN and REMB

The next generation of Network Status messages all involve receivers messaging senders via RTCP with explicit bitrate requests.

- **Temporary Maximum Media Stream Bit Rate Request** - A mantissa/exponent of a requested bitrate for a single SSRC.
- **Temporary Maximum Media Stream Bit Rate Notification** - A message to notify that a TMMBR has been received.
- **Receiver Estimated Maximum Bitrate** - A mantissa/exponent of a requested bitrate for the entire session.

TMMBR and TMMBN came first and are defined in RFC 5104. REMB came later, there was a draft submitted in draft-alvestrand-rmcat-remb, but it was never standardized.

A session that uses REMB would look like the following:

Browsers use a simple rule of thumb for incoming bandwidth estimation: 1. Tell the encoder to increase bitrate if the current packet loss is less than 2%. 2. If packet loss is higher than 10%, decrease bitrate by half of the current packet loss percentage.

```
if (packetLoss < 2%) video_bitrate *= 1.08
if (packetLoss > 10%) video_bitrate *= (1 - 0.5*lossRate)
```

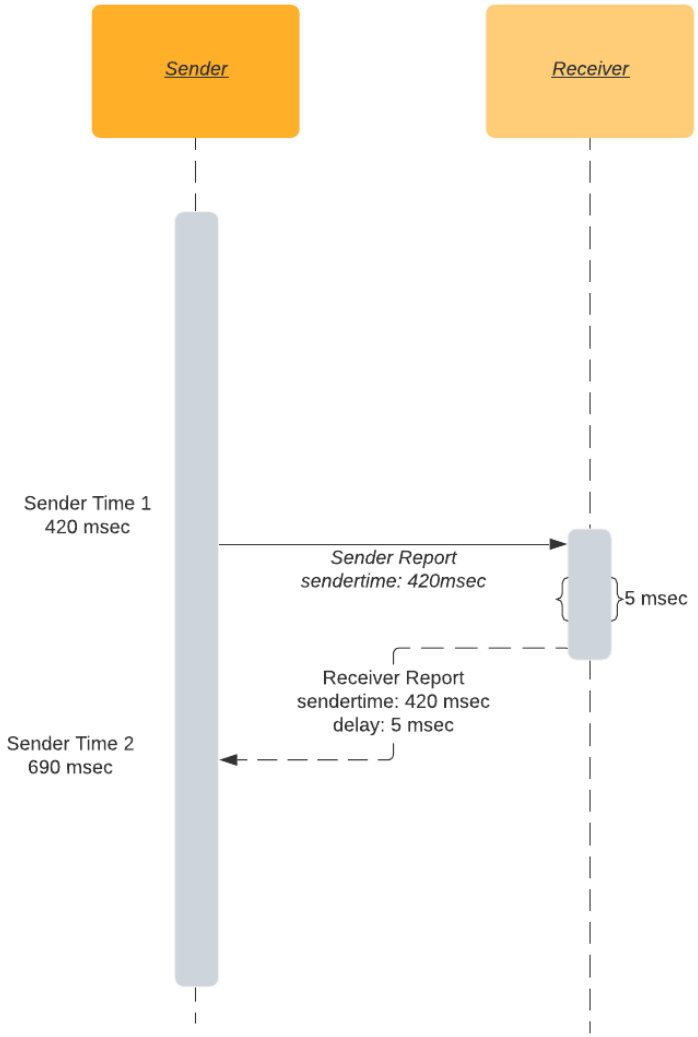


Figure 10: Round-trip time

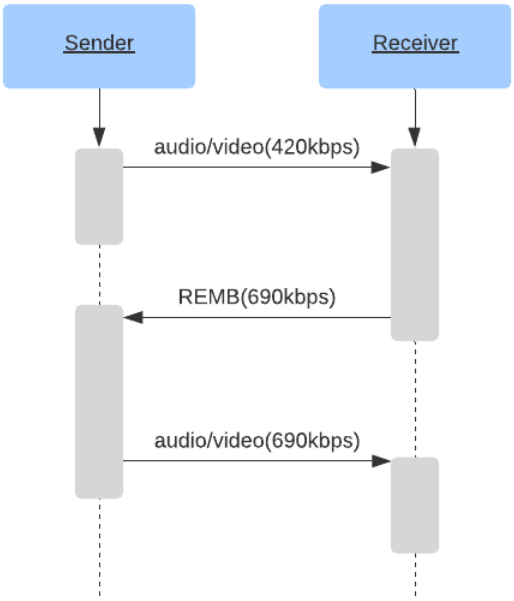


Figure 11: REMB

This method works great on paper. The Sender receives estimation from the receiver, sets encoder bitrate to the received value. Tada! We've adjusted to the network conditions.

However in practice, the REMB approach has multiple drawbacks.

Encoder inefficiency is one of them. When you set a bitrate for the encoder, it won't necessarily output the exact bitrate you requested. It may output less or more bits, depending on the encoder settings and the frame being encoded.

For example, using the x264 encoder with `tune=zerolatency` can significantly deviate from the specified target bitrate. Here is a possible scenario:

- Let's say we start off by setting the bitrate to 1000 kbps.
- The encoder outputs only 700 kbps, because there is not enough high frequency features to encode. (AKA - "staring at a wall".)
- Let's also imagine that the receiver gets the 700 kbps video at zero packet loss. It then applies REMB rule 1 to increase the incoming bitrate by 8%.
- The receiver sends a REMB packet with a 756 kbps suggestion (700 kbps * 1.08) to the sender.
- The sender sets the encoder bitrate to 756 kbps.
- The encoder outputs an even lower bitrate.
- This process continues to repeat itself, lowering the bitrate to the absolute minimum.

You can see how this would cause heavy encoder parameter tuning, and surprise users with unwatchable video even on a great connection.

Transport Wide Congestion Control

Transport Wide Congestion Control is the latest development in RTCP network status communication.

TWCC uses a quite simple principle:

Unlike in REMB, a TWCC receiver doesn't try to estimate its own incoming bitrate. It just lets the sender know which packets were received and when. Based on these reports, the sender has a very up-to-date idea of what is happening in the network.

- The sender creates an RTP packet with a special TWCC header extension, containing a list of packet sequence numbers.
- The receiver responds with a special RTCP feedback message letting the sender know if and when each packet was received.

The sender keeps track of sent packets, their sequence numbers, sizes and timestamps. When the sender receives RTCP messages from the receiver, it compares the send inter-packet delays with receive delays. If the receive delays increase, it means network congestion is happening, and the sender must act on it.

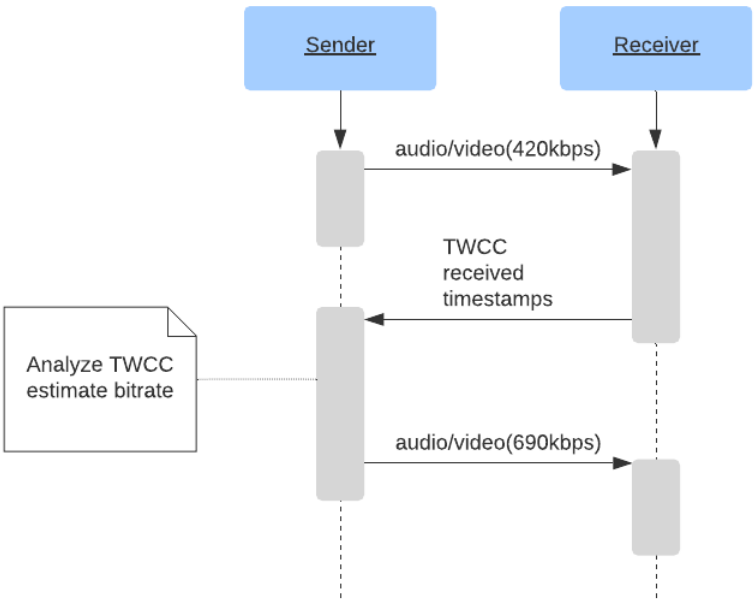


Figure 12: TWCC

In the diagram below, the median interpacket delay increase is +20 msec, a clear indicator of network congestion happening.

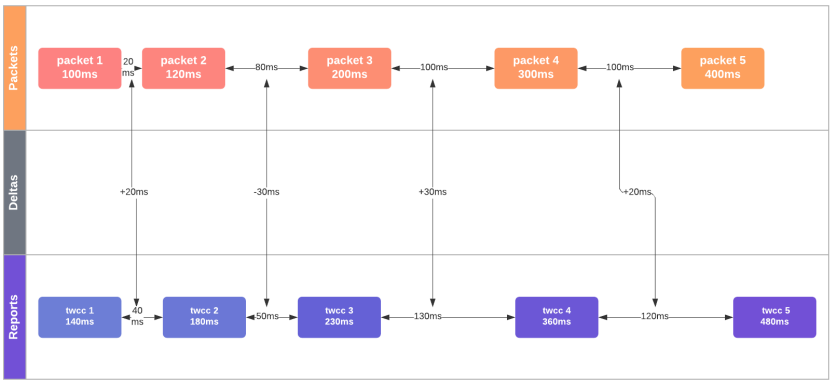


Figure 13: TWCC with delay

TWCC provides the raw data, and an excellent view into real time network conditions: - Almost instant packet loss statistics, not only the percentage lost, but the exact packets that were lost. - Accurate send bitrate. - Accurate receive bitrate. - A jitter estimate. - Differences between send and receive packet delays.

A trivial congestion control algorithm to estimate the incoming bitrate on the receiver from the sender is to sum up packet sizes received, and divide it by the remote time elapsed.

Generating a Bandwidth Estimate

Now that we have information around the state of the network we can make estimates around the bandwidth available. In 2012 the IETF started the RMCAT (RTP Media Congestion Avoidance Techniques) working group. This working group contains multiple submitted standards for congestion control algorithms. Before then, all congestion controller algorithms were proprietary.

The most deployed implementation is “A Google Congestion Control Algorithm for Real-Time Communication” defined in draft-alvestrand-rmcat-congestion. In can run in two passes. First a “loss based” pass that just uses Receiver Reports. If TWCC is available, it will also take that additional data into consideration. It predicts the current and future network bandwidth by using a Kalman filter.

There are several alternatives to GCC, for example NADA: A Unified Congestion Control Scheme for Real-Time Media and SCReAM - Self-Clocked Rate Adaptation for Multimedia.

What do I get from WebRTC's data communication?

WebRTC provides data channels for data communication. Between two peers you can open 65,534 data channels. A data channel is datagram based, and each has its own durability settings. By default, each data channel has guaranteed ordered delivery.

If you are approaching WebRTC from a media background data channels might seem wasteful. Why do I need this whole subsystem when I could just use HTTP or WebSockets?

The real power with data channels is that you can configure them to behave like UDP with unordered/lossy delivery. This is necessary for low latency and high performance situations. You can measure the backpressure and ensure you are only sending as much as your network supports.

How does it work?

WebRTC uses the Stream Control Transmission Protocol (SCTP), defined in RFC 2960. SCTP is a transport layer protocol that was intended as an alternative to TCP or UDP. For WebRTC we use it as an application layer protocol which runs over our DTLS connection.

SCTP gives you streams and each stream can be configured independently. WebRTC data channels are just thin abstractions around them. The settings around durability and ordering are just passed right into the SCTP Agent.

Data channels have some features that SCTP can't express, like channel labels. To solve that WebRTC uses the Data Channel Establishment Protocol (DCEP) which is defined in RFC 8832. DCEP defines a message to communicate the channel label and protocol.

DCEP

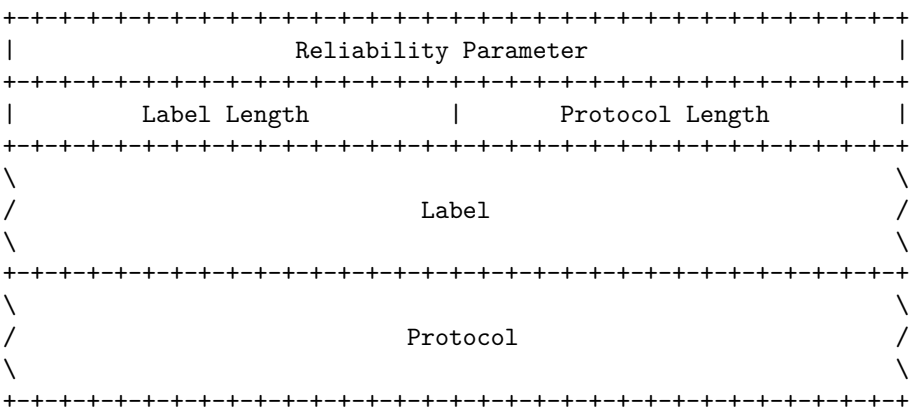
DCEP only has two messages `DATA_CHANNEL_OPEN` and `DATA_CHANNEL_ACK`. For each data channel that is opened the remote must respond with an ack.

DATA_CHANNEL_OPEN

This message is sent by the WebRTC Agent that wishes to open a channel.

Packet Format

```
 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 |  Channel Type |            Priority           |
```



Message Type Message Type is a static value of `0x03`.

Channel Type Channel Type controls durability/ordering attributes of the channel. It may have the following values:

- `DATA_CHANNEL_RELIABLE` (`0x00`) - No messages are lost and will arrive in order
- `DATA_CHANNEL_RELIABLE_UNORDERED` (`0x80`) - No messages are lost, but they may arrive out of order.
- `DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT` (`0x01`) - Messages may be lost after trying the requested amount of times, but they will arrive in order.
- `DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT_UNORDERED` (`0x81`) - Messages may be lost after trying the requested amount of times and may arrive out of order.
- `DATA_CHANNEL_PARTIAL_RELIABLE_TIMED` (`0x02`) - Messages may be lost if they don't arrive in the requested amount of time, but they will arrive in order.
- `DATA_CHANNEL_PARTIAL_RELIABLE_TIMED_UNORDERED` (`0x82`) - Messages may be lost if they don't arrive in the requested amount of time and may arrive out of order.

Priority The priority of the data channel. Data channels having a higher priority will be scheduled first. Large lower-priority user messages will not delay the sending of higher-priority user messages.

Reliability Parameter If the data channel type is `DATA_CHANNEL_PARTIAL_RELIABLE`, the suffixes configures the behavior:

- `REXMIT` - Defines how many times the sender will re-send the message before giving up.
- `TIMED` - Defines for how long time (in ms) the sender will re-send the message before giving up.

Label A UTF-8-encoded string containing the name of the data channel. This string may be empty.

Protocol If this is an empty string, the protocol is unspecified. If it is a non-empty string, it should specify a protocol registered in the "WebSocket Subprotocol Name Registry", defined in RFC 6455.

DATA_CHANNEL_ACK

This message is sent by the WebRTC Agent to acknowledge that this data channel has been opened.

Packet Format

```
 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 is the real power behind WebRTC data channels. It provides all these features of the data channel:

- Multiplexing
- Reliable delivery using a TCP-like retransmission mechanism
- Partial-reliability options
- Congestion Avoidance
- Flow Control

To understand SCTP we will explore it in three parts. The goal is that you will know enough to debug and learn the deep details of SCTP on your own after this chapter.

Concepts

SCTP is a feature rich protocol. This section is only going to cover the parts of SCTP that are used by WebRTC. Features in SCTP that are not used by WebRTC include multi-homing and path selection.

With over twenty years of development SCTP can be hard to fully grasp.

Association

Association is the term used for an SCTP Session. It is the state that is shared between two SCTP Agents while they communicate.

Streams

A stream is one bi-directional sequence of user data. When you create a data channel you are actually just creating a SCTP stream. Each SCTP Association contains a list of streams. Each stream can be configured with different reliability types.

WebRTC only allows you to configure on stream creation, but SCTP actually allows changing the configuration at anytime.

Datagram Based

SCTP frames data as datagrams and not as a byte stream. Sending and receiving data feels like using UDP instead of TCP. You don't need to add any extra code to transfer multiple files over one stream.

SCTP messages don't have size limits like UDP. A single SCTP message can be multiple gigabytes in size.

Chunks

The SCTP protocol is made up of chunks. There are many different types of chunks. These chunks are used for all communication. User data, connection initialization, congestion control, and more are all done via chunks.

Each SCTP packet contains a list of chunks. So in one UDP packet you can have multiple chunks carrying messages from different streams.

Transmission Sequence Number

The Transmission Sequence Number (TSN) is a global unique identifier for DATA chunks. A DATA chunk is what carries all the messages a user wishes to send. The TSN is important because it helps a receiver determine if packets are lost or out of order.

If the receiver notices a missing TSN, it doesn't give the data to the user until it is fulfilled.

Stream Identifier

Each stream has a unique identifier. When you create a data channel with an explicit ID, it is actually just passed right into SCTP as the stream identifier. If you don't pass an ID the stream identifier is chosen for you.

Payload Protocol Identifier

Each DATA chunk also has a Payload Protocol Identifier (PPID). This is used to uniquely identify what type of data is being exchanged. SCTP has many PPIDs, but WebRTC is only using the following five:

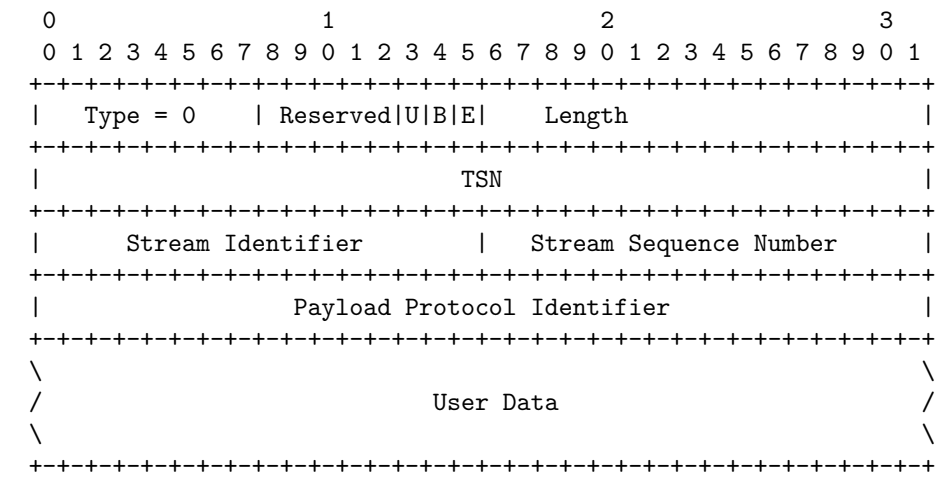
- `WebRTC DCEP` (`50`) - DCEP messages.
- `WebRTC String` (`51`) - Datachannel string messages.
- `WebRTC Binary` (`53`) - Datachannel binary messages.
- `WebRTC String Empty` (`56`) - Datachannel string messages with 0 length.
- `WebRTC Binary Empty` (`57`) - Datachannel binary messages with 0 length.

Protocol

The following are some of the chunks used by the SCTP protocol. This is not an exhaustive demonstration. This provides enough structures for the state machine to make sense.

Each Chunk starts with a `type` field. Before a list of chunks, you will also have a header.

DATA Chunk



The DATA chunk is how all user data is exchanged. When you send anything over the data channel, this is how it is exchanged.

`U` bit is set if this is an unordered packet. We can ignore the Stream Sequence Number.

`B` and `E` are the beginning and end bits. If you want to send a message that is too large for one DATA chunk it needs to be fragmented. With the the `B` and `E` bit and Sequence Numbers SCTP is able to express this.

- `B=1`, `E=0` - First piece of a fragmented user message.
- `B=0`, `E=0` - Middle piece of a fragmented user message.
- `B=0`, `E=1` - Last piece of a fragmented user message.
- `B=1`, `E=1` - Unfragmented message.

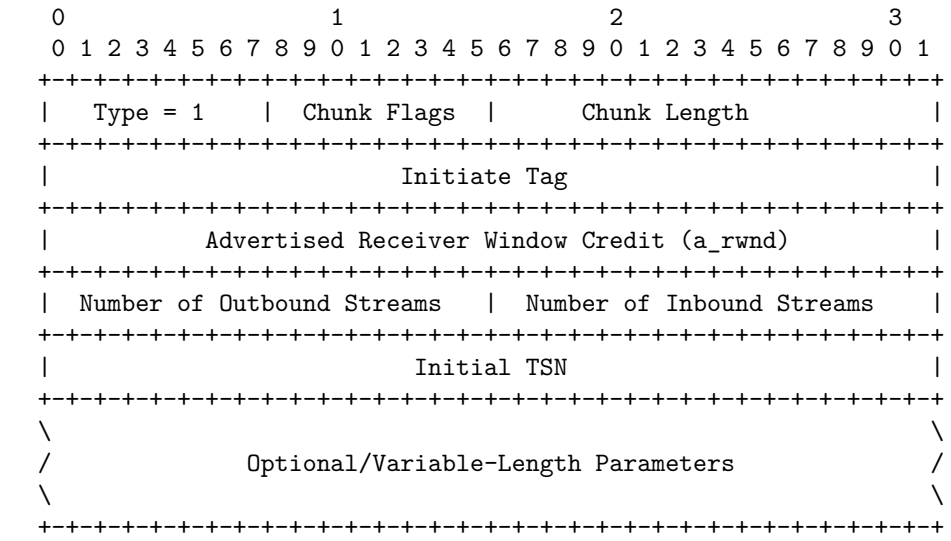
`TSN` is the Transmission Sequence Number. It is the global unique identifier for this message. After 4,294,967,295 messages this will wrap around.

`Stream Identifier` is the unique identifier for the stream this data belongs too.

`Payload Protocol Identifier` is the type of data that is flowing through this stream. For WebRTC, it is going to be DCEP, String or Binary.

`User Data` is what you are sending. All data you send via a WebRTC data channel is transmitted via a DATA chunk.

INIT Chunk



The INIT chunk starts the process of creating an association.

`Initiate Tag` is used for cookie generation. Cookies are used for Man-In-The-Middle and Denial of Service protection. They are described in greater detail in the state machine section.

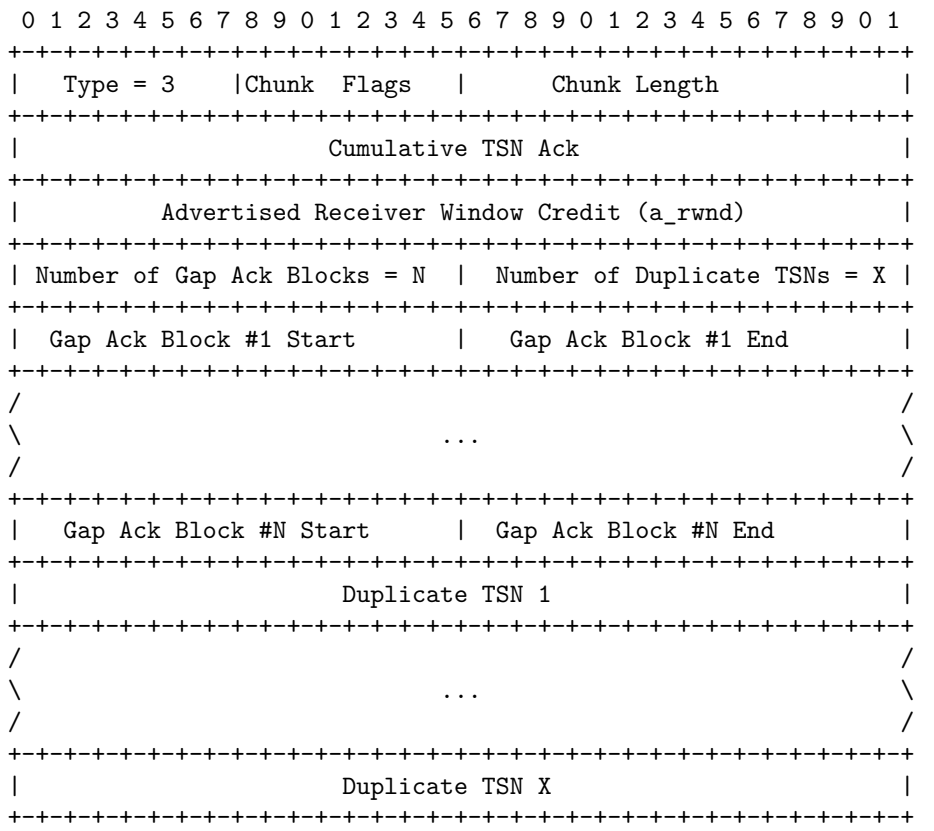
`Advertised Receiver Window Credit` is used for SCTP's Congestion Control. This communicates how large of a buffer the receiver has allocated for this association.

`Number of Outbound/Inbound Streams` notifies the remote of how many streams this agent supports.

`Initial TSN` is a random `uint32` to start the local TSN at.

`Optional Parameters` allows SCTP to introduce new features to the protocol.

SACK Chunk



The SACK (Selective Acknowledgment) Chunk is how a receiver notifies a sender it has gotten a packet. Until a sender gets a SACK for a TSN it will re-send the DATA chunk in question. A SACK does more than just update the TSN though.

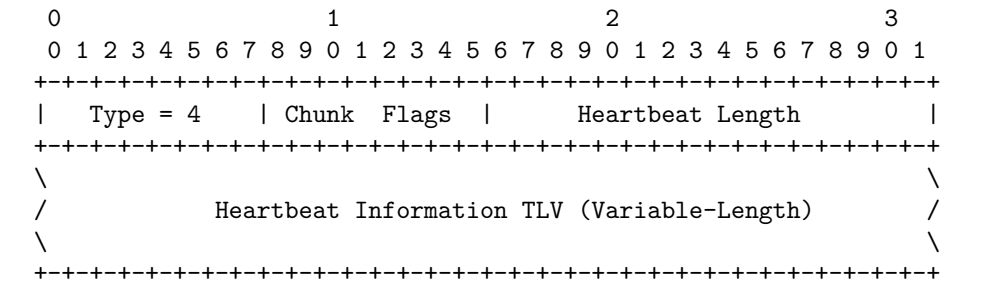
`Cumulative TSN ACK` the highest TSN that has been received.

`Advertised Receiver Window Credit` receiver buffer size. The receiver may change this during the session if more memory becomes available.

`Ack Blocks` TSNs that have been received after the `Cumulative TSN ACK`. This is used if there is a gap in packets delivered. Let's say DATA chunks with TSNs `100`, `102`, `103` and `104` are delivered. The `Cumulative TSN ACK` would be `100`, but `Ack Blocks` could be used to tell the sender it doesn't need to resend `102`, `103` or `104`.

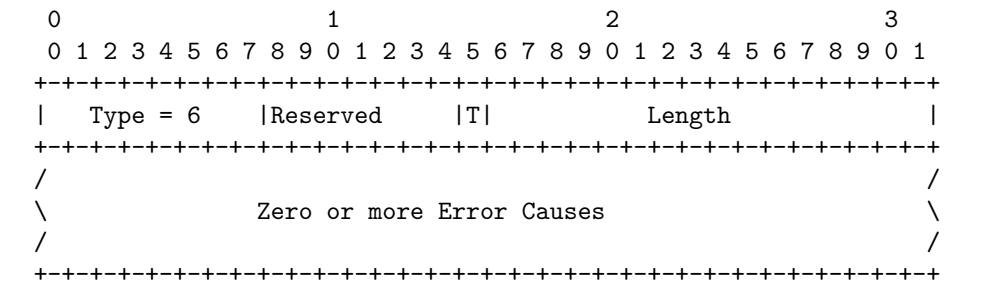
`Duplicate TSN` informs the sender that it has received the following DATA chunks more than once.

HEARTBEAT Chunk



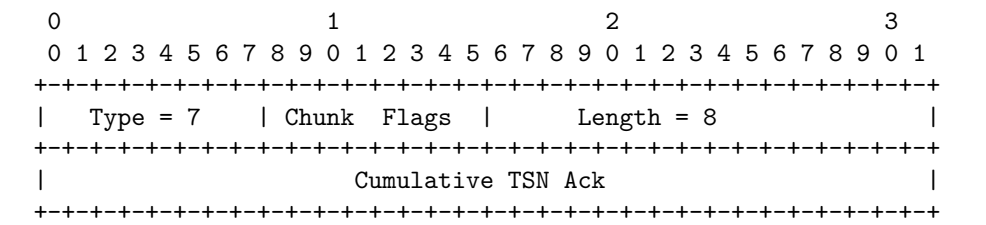
The HEARTBEAT Chunk is used to assert the remote is still responding. Useful if you aren't sending any DATA chunks and need to keep a NAT mapping open.

ABORT Chunk



An ABORT chunk abruptly shuts down the association. Used when one side enters an error state. Gracefully ending the connection uses the SHUTDOWN chunk.

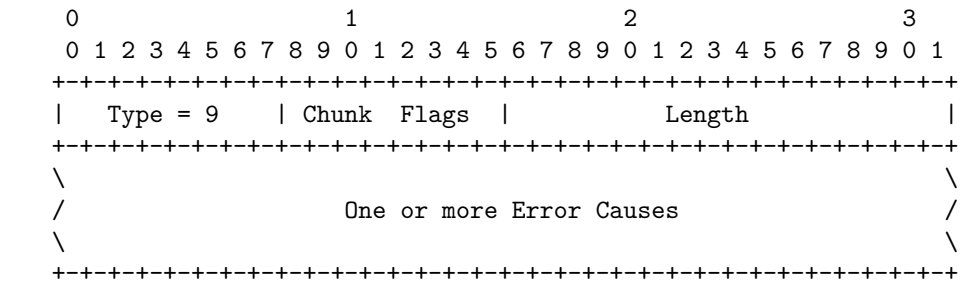
SHUTDOWN Chunk



The SHUTDOWN Chunk starts a graceful shutdown of the SCTP association. Each agent informs the remote of the last TSN it sent. This ensures that no packets are lost. WebRTC doesn't do a graceful shutdown of the SCTP association. You need to tear down each data channel yourself to handle it gracefully.

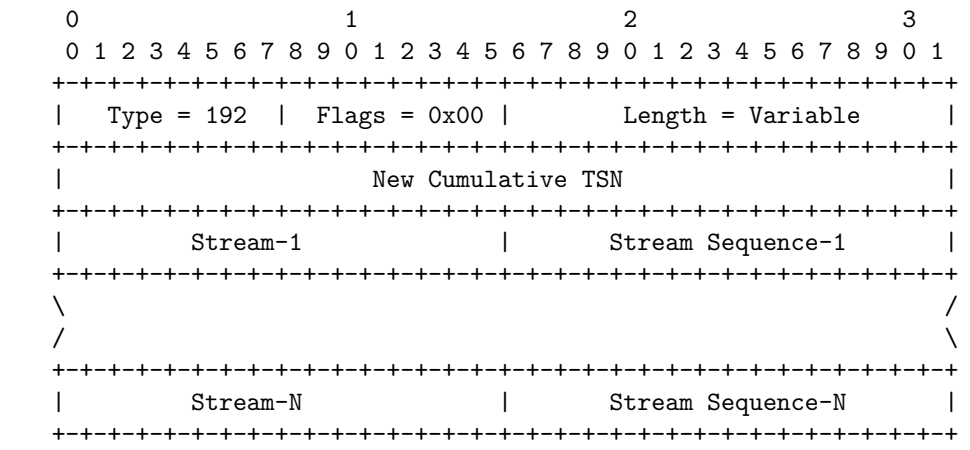
`Cumulative TSN ACK` is the last TSN that was sent. Each side knows not to terminate until they have received the DATA chunk with this TSN.

ERROR Chunk



An ERROR chunk is used to notify the remote SCTP Agent that a non-fatal error has occurred.

FORWARD TSN Chunk



The `FORWARD TSN` chunk moves the global TSN forward. SCTP does this, so you can skip some packets you don't care about anymore. Let's say you send `10 11 12 13 14 15` and these packets are only valid if they all arrive. This data is also real-time sensitive, so if it arrives late it isn't useful.

If you lose `12` and `13` there is no reason to send `14` and `15`! SCTP uses the `FORWARD TSN` chunk to achieve that. It tells the receiver that `14` and `15` aren't going to be delivered anymore.

`New Cumulative TSN` this is the new TSN of the connection. Any packets before this TSN will not be retained.

`Stream` and `Stream Sequence` are used to jump the `Stream Sequence Number` number ahead. Refer back to the DATA Chunk for the significance of this field.

State Machine

These are some interesting parts of the SCTP state machine. WebRTC doesn't use all the features of the SCTP state machine, so we have excluded those parts. We also have simplified some components to make them understandable on their own.

Connection Establishment Flow

The `INIT` and `INIT ACK` chunks are used to exchange the capabilities and configurations of each peer. SCTP uses a cookie during the handshake to validate the peer it is communicating with. This is to ensure that the handshake is not intercepted and to prevent DoS attacks.

The `INIT ACK` chunk contains the cookie. The cookie is then returned to its creator using the `COOKIE ECHO`. If cookie verification is successful the `COOKIE ACK` is sent and DATA chunks are ready to be exchanged.

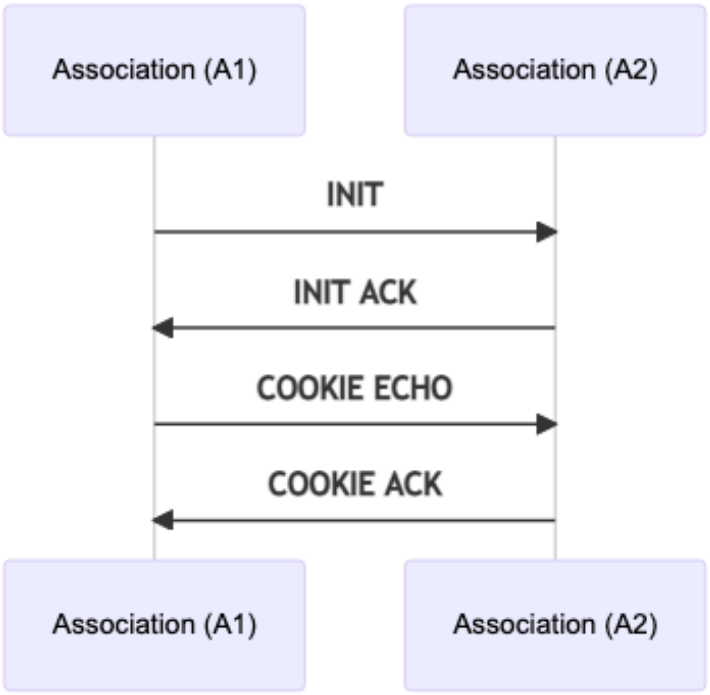


Figure 14: Connection establishment

Connection Teardown Flow

SCTP uses the `SHUTDOWN` chunk. When an agent receives a `SHUTDOWN` chunk it will wait until it receives the requested `Cumulative TSN ACK`. This allows a user to ensure that all data is delivered even if the connection is lossy.

Keep-Alive Mechanism

SCTP uses the `HEARTBEAT REQUEST` and `HEARTBEAT ACK` Chunks to keep the connection alive. These are sent on a configurable interval. SCTP also performs an exponential backoff if the packet hasn't arrived.

The `HEARTBEAT` chunk also contains a time value. This allows two associations to compute trip time between two agents.

Applied WebRTC

Now that you know how WebRTC works it is time to build with it! This chapter explores what people are building with WebRTC, and how they are building it. You will learn all the interesting things that are happening with WebRTC. The power of WebRTC comes at a cost. Building production grade WebRTC services is challenging. This chapter will try to explain those challenges before you hit them.

By Use Case

Many think WebRTC is just a technology for conferencing in the web browser. It is so much more than that though! WebRTC is used for a wide range of use cases. New use cases showing up all the time. In this chapter we will list some common ones and how WebRTC is revolutionizing them.

Conferencing

Conferencing is the original use case for WebRTC. The protocol contains a few necessary features that no other protocol offers in the browser. You could build a conferencing system with WebSockets and it may work in optimal conditions. If you want something that can be deployed in real world network conditions, WebRTC is the best choice.

WebRTC provides congestion control and adaptive bitrate for media. As the conditions of the network change, users will still get the best experience possible. Developers don't have to write any additional code to measure these conditions either.

Participants can send and receive multiple streams. They can also add and remove those streams at any time during the call. Codecs are negotiated as well. All of this functionality is provided by the browser, no custom code is required to be written by the developer.

Conferencing also benefits from data channels. Users can send metadata or share documents. You can create multiple streams and configure them if you need performance more than reliability.

Broadcasting

Lots of new projects are starting to appear in the broadcast space that use WebRTC. The protocol has a lot to offer for both the publisher and consumer of media.

WebRTC being in the browser makes it easy for users to publish video. It removes the requirement for users to download a new client. Any platform that has a web browser can publish video. Publishers can then send multiple tracks and modify or remove them at anytime. This is a huge improvement over legacy protocols that only allowed one audio or one video track per connection.

WebRTC gives developers greater control over the latency versus quality trade-offs. If it is more important that latency never exceeds a certain threshold, and you are willing to tolerate some decoding artifacts. You can configure the viewer to play media as soon as it arrives. With other protocols that run over TCP, that isn't as easy. In the browser you can request data and that is it.

Remote Access

Remote Access is when you remotely access another computer via WebRTC. You could have complete control of the remote host, or maybe just a single application. This is great for running computationally expensive tasks when the local hardware can't do it. Like running a new video game, or CAD software. WebRTC was able to revolutionize the space in three ways.

WebRTC can be used to remotely access a host that isn't world routable. With NAT Traversal you can access a computer that is only available via STUN. This is great for security and privacy. Your users don't have to route video through an ingest, or a "jump box". NAT Traversal also makes deployments easier. You don't have to worry about port forwarding or setting up a static IP ahead of time.

Data channels are really powerful as well in this scenario. They can be configured so that only the latest data is accepted. With TCP run the risk of encountering Head-of-line blocking. An old mouse click or keypress could arrive late, and block the subsequent ones from being accepted. WebRTC's data channels are designed to handle this and can be configured to not retry on lost packets. You can also measure the backpressure and make sure that you aren't sending more data than your network supports.

WebRTC being available in the browser has been a huge quality of life improvement. You don't have to download a proprietary client to start the session. More and more clients are coming with WebRTC bundled, smart TVs are getting full web browsers now.

File Sharing and Censorship Circumvention

File Sharing and Censorship Circumvention are dramatically different problems. However, WebRTC solves the same problems for them both. It makes them both easily available and harder to block.

The first problem that WebRTC solves is getting the client. If you want to join a file sharing network, you need to download the client. Even if the network is distributed, you still need to get the client first. In a restricted network the download will often be blocked. Even if you can download it, the user may not be able to install and run the client. WebRTC is available in every web browser already making it readily available.

The second problem that WebRTC solves is your traffic being blocked. If you use a protocol that is just for file sharing or censorship circumvention it is much easier to block it. Since WebRTC is a general purpose protocol, blocking it would impact everyone. Blocking WebRTC might prevent other users of the network from joining conference calls.

Internet of Things

Internet of Things (IoT) covers a few different use cases. For many this means network connected security cameras. Using WebRTC you can stream the video to another WebRTC peer like your phone or a browser. Another use case is having devices connect and exchange sensor data. You can have two devices in your LAN exchange climate, noise or light readings.

WebRTC has a huge privacy advantage here over legacy video stream protocols. Since WebRTC supports P2P connectivity the camera can send the video directly to your browser. There is no reason for your video to be sent to a 3rd party server. Even when video is encrypted, an attacker can make assumptions from the metadata of the call.

Interoperability is another advantage for the IoT space. WebRTC is available in lots of different languages; C#, C++, C, Go, Java, Python, Rust and TypeScript. This means you can use the language that works best for you. You also don't have to turn to proprietary protocols or formats to be able to connect your clients.

Media Protocol Bridging

You have existing hardware and software that is producing video, but you can't upgrade it yet. Expecting users to download a proprietary client to watch videos is frustrating. The answer is to run a WebRTC bridge. The bridge translates between the two protocols so users can use the browser with your legacy setup.

Many of the formats that developers bridge with use the same protocols as WebRTC. SIP is commonly exposed via WebRTC and allows users to make phone calls from their browser. RTSP is used in lots of legacy security cameras. They

both use the same underlying protocols (RTP and SDP) so it is computationally cheap to run. The bridge is just required to add or remove things that are WebRTC specific.

Data Protocol Bridging

A web browser is only able to speak a constrained set of protocols. You can use HTTP, WebSockets, WebRTC and QUIC. If you want to connect to anything else, you need to use a protocol bridge. A protocol bridge is a server that converts foreign traffic into something the browser can access. A popular example is using SSH from your browser to access a server. WebRTC's data channels have two advantages over the competition.

WebRTC's data channels allow unreliable and unordered delivery. In cases where low latency is critical this is needed. You don't want new data to be blocked by old data, this is known as head-of-line blocking. Imagine you are playing a multiplayer First-person shooter. Do you really care where the player was two seconds ago? If that data didn't arrive in time, it doesn't make sense to keep trying to send it. Unreliable and unordered delivery allows you to get your data as soon as it arrives.

Data channels also provide feedback pressure. This tells you if you are sending data faster than your connection can support. You then have two choices when this happens. The data channel can either be configured to buffer and deliver the data late, or you can drop the data that hasn't arrived in real-time.

Teleoperation

Teleoperation is the act of controlling a device remotely via WebRTC data channels, and sending the video back via RTP. Developers are driving cars remotely via WebRTC today! This is used to control robots at construction sites and deliver packages. Using WebRTC for these problems makes sense for two reasons.

The ubiquity of WebRTC makes it easy to give users control. All the user needs is a web browser and an input device. Browsers even support taking input from joysticks and gamepads. WebRTC completely removes the need to install an additional client on the user's device.

Distributed CDN

Distributed CDNs are a subset of file sharing. The files being distributed are configured by the CDN operator instead. When users join the CDN network they can download and share the allowed files. Users get all the same benefits as file sharing.

These CDNs work great when you are at an office with poor external connectivity, but great LAN connectivity. You can have one user download a video, and then

share it with everyone else. Since everyone isn't attempting to fetch the same file via the external network, the transfer will complete faster.

WebRTC Topologies

WebRTC is a protocol for connecting two agents, so how are developers connecting hundreds of people at once? There are a few different ways you can do it, and they all have pros and cons. These solutions broadly fall into two categories; Peer-to-Peer or Client/Server. WebRTC's flexibility allows us to create both.

One-To-One

One-to-One is the first connection type you will use with WebRTC. You connect two WebRTC Agents directly and they can send bi-directional media and data. The connection looks like this.

Figure 15: One-to-One

Full Mesh

Full mesh is the answer if you want to build a conference call or a multiplayer game. In this topology each user establishes a connection with every other user directly. This allows you to build your application, but it comes with some downsides.

In a Full Mesh topology each user is connected directly. That means you have to encode and upload video independently for each member of the call. The network conditions between each connection will be different, so you can't reuse the same video. Error handling is also difficult in these deployments. You need to carefully consider if you have lost complete connectivity, or just connectivity with one remote peer.

Because of these concerns, a Full Mesh is best used for small groups. For anything larger a client/server topology is best.

Hybrid Mesh

Hybrid Mesh is an alternative to Full Mesh that can alleviate some of the Full Mesh's issues. In a Hybrid Mesh connections aren't established between every user. Instead, media is relayed through peers in the network. This means that the creator of the media doesn't have to use as much bandwidth to distribute media.

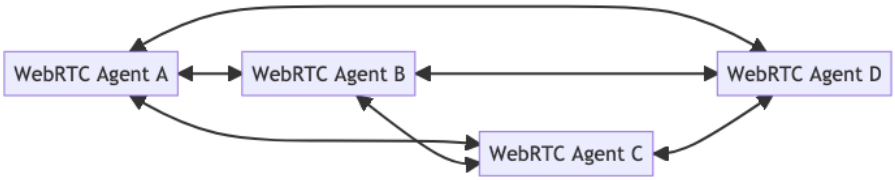


Figure 16: Full mesh

This does have some downsides. In this set up, the original creator of the media has no idea who its video is being sent too, and if it arrived successfully. You also will have an increase in latency with every hop in your Hybrid Mesh network.

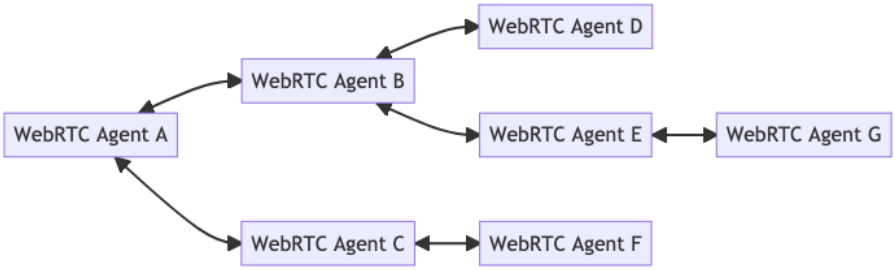


Figure 17: Hybrid mesh

Selective Forwarding Unit

An SFU (Selective Forwarding Unit) also solves the issues of Full Mesh, but in an entirely different way. An SFU implements a client/server topology, instead of P2P. Each WebRTC peer connects to the SFU and uploads its media. The SFU then forwards this media out to each connected client.

With an SFU each WebRTC Agent only has to encode and upload their video once. The burden of distributing it to all the viewers is on the SFU. Connectivity with an SFU is much easier than P2P as well. You can run an SFU on a world routable address, making it much easier for clients to connect. You don't need to worry about NAT Mappings. You do still need to make sure your SFU is available via TCP (either via ICE-TCP or TURN).

Building a simple SFU can be done in a weekend. Building a good SFU that can handle all types of clients is never ending. Tuning the Congestion Control, Error Correction and Performance is a never ending task.

MCU

A MCU (Multi-point Conferencing Unit) is a client/server topology like an SFU, but composites the output streams. Instead of distributing the outbound media

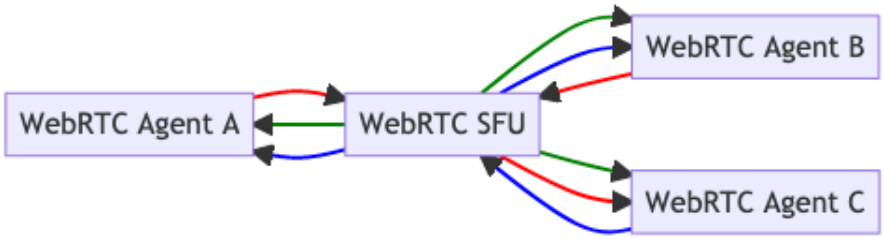


Figure 18: Selective Forwarding Unit

unmodified it re-encodes them as one feed.

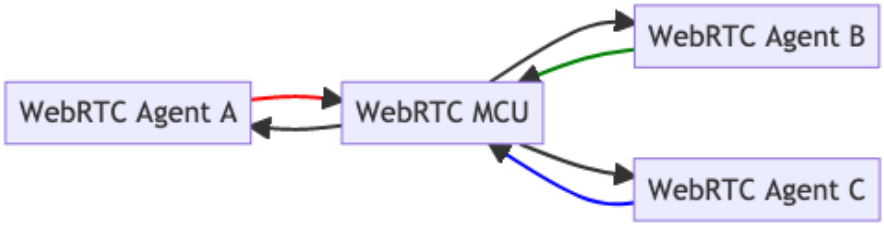


Figure 19: Multi-point Conferencing Unit

Debugging

Debugging WebRTC can be a daunting task. There are a lot of moving parts, and they all can break independently. If you aren't careful, you can lose weeks of time looking at the wrong things. When you do finally find the part that is broken, you will need to learn a bit to understand why.

This chapter will get you in the mindset to debug WebRTC. It will show you how to break down the problem. After we know the problem, we will give a quick tour of the popular debugging tools.

Isolate The Problem

When debugging, you need to isolate where the issue is coming from. Start from the beginning of the...

Signaling Failure

Networking Failure Test your STUN server using netcat:

1. Prepare the **20-byte** binding request packet:

```
echo -ne "\x00\x01\x00\x00\x21\x12\xA4\x42TESTTESTTEST" | hexdump -C
00000000  00 01 00 00 21 12 a4 42  54 45 53 54 54 45 53 54  |....!..BTESTTEST|
00000010  54 45 53 54                                       |TEST|
00000014
```

Interpretation:

- `00 01` is the message type.
- `00 00` is the length of the data section.
- `21 12 a4 42` is the magic cookie.
- and `54 45 53 54 54 45 53 54 54 45 53 54` (Decodes to ASCII: `TESTTESTTEST`) is the 12-byte transaction ID.

2. Send the request and wait for the **32 byte** response:

```
stunserver=stun1.l.google.com;stunport=19302;listenport=20000;echo -ne "\x00\x01\x00\x0
00000000  01 01 00 0c 21 12 a4 42  54 45 53 54 54 45 53 54  |....!..BTESTTEST|
00000010  54 45 53 54 00 20 00 08  00 01 6f 32 7f 36 de 89  |TEST. ....o2.6..|
00000020
```

Interpretation:

- `01 01` is the message type
- `00 0c` is the length of the data section which decodes to 12 in decimal
- `21 12 a4 42` is the magic cookie
- and `54 45 53 54 54 45 53 54 54 45 53 54` (Decodes to ASCII: `TESTTESTTEST`) is the 12-byte transaction ID.
- `00 20 00 08 00 01 6f 32 7f 36 de 89` is the 12-byte data, interpretation:
  - `00 20` is the type: `XOR-MAPPED-ADDRESS`
  - `00 08` is the length of the value section which decodes to 8 in decimal
  - `00 01 6f 32 7f 36 de 89` is the data value, interpretation:
    - `00 01` is the address type (IPv4)
    - `6f 32` is the XOR-mapped port
    - `7f 36 de 89` is the XOR-mapped IP address

Decoding the XOR-mapped section is cumbersome, but we can trick the stun server to perform a dummy XOR-mapping, by supplying an (invalid) dummy magic cookie set to `00 00 00 00`:

```
stunserver=stun1.l.google.com;stunport=19302;listenport=20000;echo -ne "\x00\x01\x00\x00\x0
00000000  01 01 00 0c 00 00 00 00  54 45 53 54 54 45 53 54  |........TESTTEST|
00000010  54 45 53 54 00 01 00 08  00 01 4e 20 5e 24 7a cb  |TEST......N ^$z.|
00000020
```

XOR-ing against the dummy magic cookie is idempotent, so the port and address will be in clear in the response. This will not work in all situations, because some routers manipulate the passing packets, cheating on the IP address. If we look at the returned data value (last eight bytes):

- `00 01 4e 20 5e 24 7a cb` is the data value, interpretation:
  - `00 01` is the address type (IPv4)
  - `4e 20` is the mapped port, which decodes to 20000 in decimal
  - `5e 24 7a cb` is the IP address, which decodes to `94.36.122.203` in dotted-decimal notation.

Security Failure

Media Failure

Data Failure

Tools of the trade

netcat (nc) netcat is command-line networking utility for reading from and writing to network connections using TCP or UDP. It is typically available as the `nc` command.

tcpdump tcpdump is a command-line data-network packet analyzer.

Common commands: - Capture UDP packets to and from port 19302, print a hexdump of the packet content:

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

- Same, but save packets in a PCAP (packet capture) file for later inspection:

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

  The PCAP file can be opened with the Wireshark application: `wireshark stun.pcap`

wireshark

webrtc-internals Chrome comes with a built-in WebRTC statistics page available at chrome://webrtc-internals.

Latency

How do you know you have high latency? You may have noticed that your video is lagging, but do you know precisely how much it is lagging? To be able to reduce this latency, you have to start by measuring it first.

True latency is supposed to be measured end-to-end. That means not just the latency of the network path between the sender and the receiver, but the combined latency of camera capture, frame encoding, transmission, receiving, decoding and displaying, as well as possible queueing between any of these steps.

End-to-end latency is not a simple sum of latencies of each component.

While you could theoretically measure the latency of the components of a live video transmission pipeline separately and then add them together, in practice, at least some components will be either inaccessible for instrumentation, or produce significantly different results when measured outside the pipeline. Variable queue depths between pipeline stages, network topology and camera exposure changes are just a few examples of components affecting end-to-end latency.

The intrinsic latency of each component in your live-streaming system can change and affect downstream components. Even the content of captured video affects latency. For example, many more bits are required for high frequency features such as tree branches, compared to a low frequency clear blue sky. A camera with auto exposure turned on may take *much* longer than the expected 33 milliseconds to capture a frame, even if when the capture rate is set to 30 frames per second. Transmission over the network, especially so cellular, is also very dynamic due to changing demand. More users introduce more chatter on the air. Your physical location (notorious low signal zones) and multiple other factors increase packet loss and latency. What happens when you send a packet to a network interface, say WiFi adapter or an LTE modem for delivery? If it can not be immediately delivered it is queued on the interface, the larger the queue the more latency such network interface introduces.

Manual end-to-end latency measurement

When we talk about end-to-end latency, we mean the time between an event happening and it being observed, meaning video frames appearing on the screen.

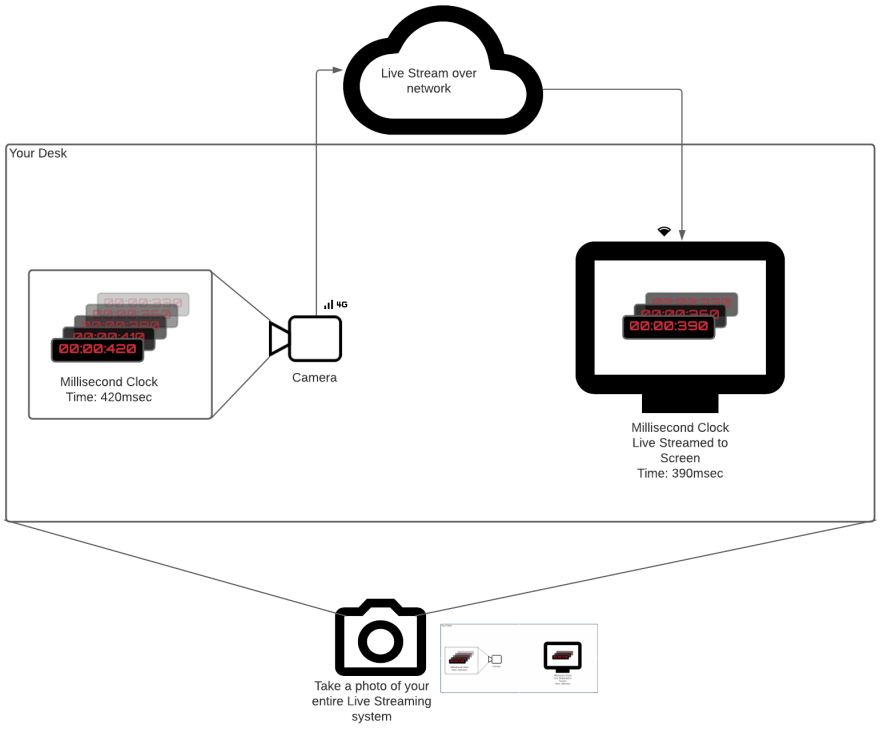
```
EndToEndLatency = T(observe) - T(happen)
```

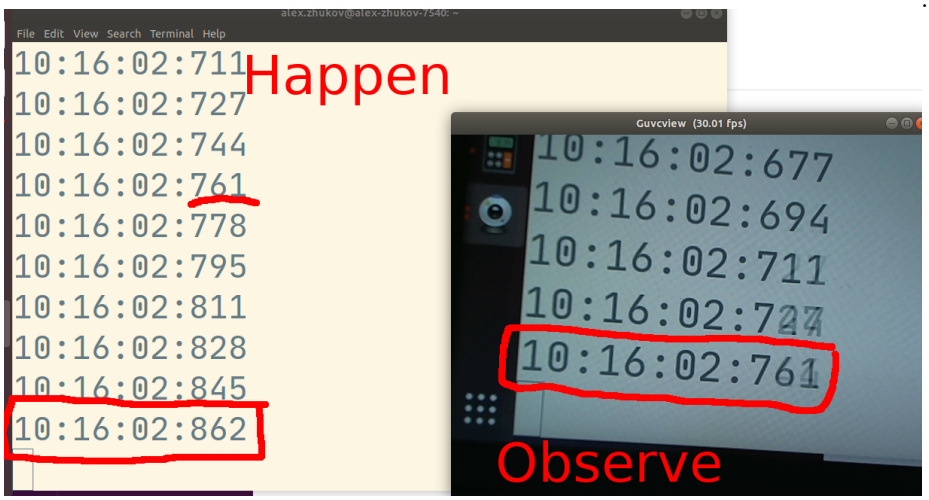
A naive approach is to record the time when an event happens and subtract it from the time at observation. However, as precision goes down to milliseconds time synchronization becomes an issue. Trying to synchronize clocks across distributed systems is mostly futile, even a small error in time sync produces unreliable latency measurement.

A simple workaround for clock sync issues is to use the same clock. Put sender and receiver in the same frame of reference.

Imagine you have a ticking millisecond clock or any other event source really. You want to measure latency in a system that live streams the clock to a remote screen by pointing a camera at it. An obvious way to measure time between the millisecond timer ticking (`Thappen`) and video frames of the clock appear on screen (`Tobserve`) is the following: - Point your camera at the millisecond clock. - Send video frames to a receiver that is in the same physical location. - Take a picture (use your phone) of the millisecond timer and the received video on screen. - Subtract two times.

That is the most true-to-yourself end-to-end latency measurement. It accounts for all components latencies (camera, encoder, network, decoder) and does not rely on any clock synchronization.





In the photo above measured end-to-end latency is 101 msec. Event happening right now is 10:16:02.862, but the live-streaming system observer sees 10:16:02.761.

Automatic end-to-end latency measurement

As of the time of writing (May 2021) the WebRTC standard for end-to-end delay is being actively discussed. Firefox implemented a set of APIs to let users create automatic latency measurement on top of standard WebRTC APIs. However in this paragraph, we discuss the most compatible way to automatically measure latency.

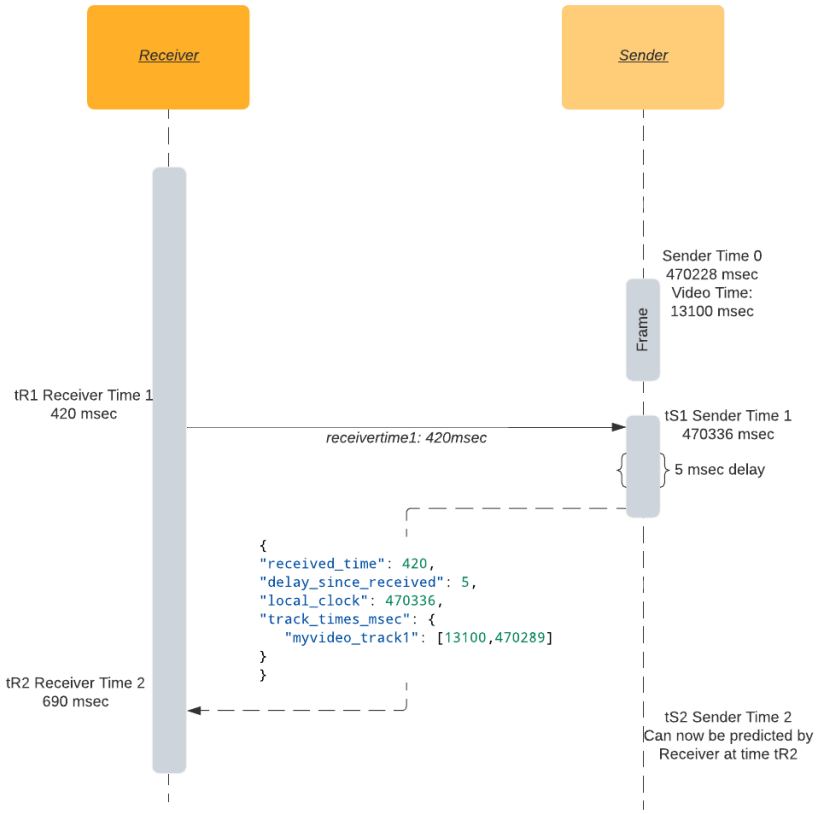


Figure 20: NTP Style Latency Measurement

Roundtrip time in a nutshell: I send you my time `tR1`, when I receive back my `tR1` at time `tR2`, I know round trip time is `tR2 - tR1`.

Given a communication channel between sender and receiver (e.g. DataChannel), the receiver may model the sender's monotonic clock by following the steps below: 1. At time `tR1`, the receiver sends a message with its local monotonic clock timestamp. 2. When it is received at the sender with local time `tS1`, the sender responds with a copy of `tR1` as well as the sender's `tS1` and the sender's

video track time `tSV1`. 3. At time `tR2` on the receiving end, round trip time is calculated by subtracting the message's send and receive times: `RTT = tR2 - tR1`. 4. Round trip time `RTT` together with sender local timestamp `tS1` is enough to create an estimation of the sender's monotonic clock. Current time on the sender at time `tR2` would be equal to `tS1` plus half of round trip time. 5. Sender's local clock timestamp `tS1` paired with video track timestamp `tSV1` together with round trip time `RTT` is therefore enough to sync receiver video track time to the sender video track.

Now that we know how much time has passed since the last known sender video frame time `tSV1`, we can approximate the latency by subtracting the currently displayed video frame's time (`actual_video_time`) from the expected time:

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

This method's drawback is that it does not include the camera's intrinsic latency. Most video systems consider the frame capture timestamp to be the time when the frame from the camera is delivered to the main memory, which will be a few moments after the event being recorded actually happened.

Example latency estimation A sample implementation opens a `latency` data channel on the receiver and periodically sends the receiver's monotonic timer timestamps to the sender. The sender responds back with a JSON message and the receiver calculates the latency based the message.

```
{
    "received_time": 64714,       // Timestamp sent by receiver, sender reflects the timesta
    "delay_since_received": 46,   // Time elapsed since last `received_time` received on ser
    "local_clock": 1597366470336, // The sender's current monotonic clock time.
    "track_times_msec": {
        "myvideo_track1": [
            13100,        // Video frame RTP timestamp (in milliseconds).
            1597366470289 // Video frame monotonic clock timestamp.
        ]
    }
}
```

Open the data channel on the receiver:

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

Send the receiver's time `tR1` periodically. This example uses 2 seconds for no particular reason:

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

Handle incoming message from receiver on sender:

```
// Assuming event.data is a string like "1234567".
tR1 = event.data
now = Math.trunc(performance.now());
tSV1 = 42000; // Current frame RTP timestamp converted to millisecond timescale.
tS1 = 1597366470289; // Current frame monotonic clock timestamp.
msg = {
  "received_time": tR1,
  "delay_since_received": 0,
  "local_clock": now,
  "track_times_msec": {
    "myvideo_track1": [tSV1, tS1]
  }
}
dataChannel.send(JSON.stringify(msg));
```

Handle incoming message from the sender and print the estimated latency to the `console`:

```
let tR2 = performance.now();
let fromSender = JSON.parse(event.data);
let tR1 = fromSender['received_time'];
let delay = fromSender['delay_since_received']; // How much time that has passed between the
let senderTimeFromResponse = fromSender['local_clock'];
let rtt = tR2 - delay - tR1;
let networkLatency = rtt / 2;
let senderTime = (senderTimeFromResponse + delay + networkLatency);
VIDEO.requestVideoFrameCallback((now, framemeta) => {
    // Estimate current time of the sender.
    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); // Convert RTP timeb
    let latency = expectedVideoTimeMsec - actualVideoTimeMsec;
    console.log('latency', latency, 'msec');
});
```

Actual video time in browser

> `<video>.requestVideoFrameCallback()` allows web authors to be notified when a frame has been presented for composition.

Until very recently (May 2020), it was next to impossible to reliably get a timestamp of the currently displayed video frame in browsers. Workaround methods based on `video.currentTime` existed, but were not particularly precise.

Both the Chrome and Mozilla browser developers supported the introduction of a new W3C standard, `HTMLVideoElement.requestVideoFrameCallback()`, that adds an API callback to access the current video frame time. While the addition sounds trivial, it has enabled multiple advanced media applications on the web that require audio and video synchronization. Specifically for WebRTC, the callback will include the `rtpTimestamp` field, the RTP timestamp associated with the current video frame. This should be present for WebRTC applications, but absent otherwise.

Latency Debugging Tips

Since debugging is likely to affect the measured latency, the general rule is to simplify your setup to the smallest possible one that can still reproduce the issue. The more components you can remove, the easier it will be to figure out which component is causing the latency problem.

Camera latency Depending on camera settings camera latency may vary. Check auto exposure, auto focus and auto white balance settings. All the "auto" features of web cameras take some extra time to analyse the captured image before making it available to the WebRTC stack.

If you are on Linux, you can use the `v4l2-ctl` command line tool to control camera settings:

```
# Disable autofocus:
v4l2-ctl -d /dev/video0 -c focus_auto=0
# Set focus to infinity:
v4l2-ctl -d /dev/video0 -c focus_absolute=0
```

You can also use the graphical UI tool `guvcview` to quickly check and tweak camera settings.

Encoder latency Most modern encoders will buffer some frames before outputting an encoded one. Their first priority is a balance between the quality of the produced picture and bitrate. Multipass encoding is an extreme example of an encoder's disregard for output latency. During the first pass encoder ingests the entire video and only after that starts outputting frames.

However, with proper tuning people have achieved sub-frame latencies. Make sure your encoder does not use excessive reference frames or rely on B-frames. Every codec's latency tuning settings are different, but for x264 we recommend using `tune=zerolatency` and `profile=baseline` for the lowest frame output latency.

Network latency Network latency is the one you can arguably do least about, other than upgrading to a better network connection. Network latency is very much like the weather - you can't stop the rain, but you can check the forecast

and take an umbrella. WebRTC is measuring network conditions with millisecond precision. Important metrics are: - Round-trip time. - Packet loss and packet retransmissions.

Round-Trip Time

The WebRTC stack has a built-in network round trip time (RTT) measurement mechanism. A good-enough approximation of latency is half of the RTT. It assumes that it takes the same time to send and receive a packet, which is not always the case. RTT sets the lower bound on the end-to-end latency. Your video frames can not reach the receiver faster than `RTT/2`, no matter how optimized your camera to encoder pipeline is.

The built-in RTT mechanism is based on special RTCP packets called sender/receiver reports. Sender sends its time reading to receiver, the receiver in turn reflects the same timestamp to the sender. Thereby the sender knows how much time it took for the packet to travel to the receiver and return back. Refer to Sender/Receiver Reports chapter for more details of RTT measurement.

Packet loss and packet retransmissions

Both RTP and RTCP are protocols based on UDP, which does not have any guarantee of ordering, successful delivery, or non-duplication. All of the above can and does happen in real world WebRTC applications. An unsophisticated decoder implementation expects all packets of a frame to be delivered for the decoder to successfully reassemble the image. In presence of packet loss decoding artifacts may appear if packets of a P-frame are lost. If I-frame packets are lost then all of its dependent frames will either get heavy artifacts or won't be decoded at all. Most likely this will make the video "freeze" for a moment.

To avoid (well, at least to try to avoid) video freezing or decoding artifacts, WebRTC uses negative acknowledgement messages (NACK). When the receiver does not get an expected RTP packet, it returns a NACK message to tell the sender to send the missing packet again. The receiver *waits* for the retransmission of the packet. Such retransmissions cause increased latency. The number of NACK packets sent and received is recorded in WebRTC's built-in stats fields outbound stream nackCount and inbound stream nackCount.

You can see nice graphs of inbound and outbound `nackCount` on the webrtc internals page. If you see the `nackCount` increasing, it means the network is experiencing high packet loss, and the WebRTC stack is doing its best to create a smooth video/audio experience despite that.

When packet loss is so high that the decoder is unable to produce an image, or subsequent dependent images like in the case of a fully lost I-frame, all future P-frames will not be decoded. The receiver will try to mitigate that by sending a special Picture Loss Indication message (PLI). Once the sender receives a `PLI`, it will produce a new I-frame to help the receiver's decoder. I-frames are normally larger in size than P-frames. This increases the number of packets that need to

be transmitted. Like with NACK messages, the receiver will need to wait for the new I-frame, introducing additional latency.

Watch for `pliCount` on the webrtc internals page. If it increases, tweak your encoder to produce less packets or enable a more error resilient mode.

Receiver side latency Latency will be affected by packets arriving out of order. If the bottom half of the image packet comes before the top you would have to wait for the top before decoding. This is explained in the Solving Jitter chapter in great detail.

You can also refer to the built-in jitterBufferDelay metric to see how long a frame was held in the receive buffer, waiting for all of its packets until it was released to the decoder.

History

This section is ongoing, and we don't have all the facts yet. We are conducting interviews to build a history of digital communication.

RTP

RTP and RTCP is the protocol that handles all media transport for WebRTC. It was defined in RFC 1889 in January 1996. We are very lucky to have one of the authors Ron Frederick talk about it himself. Ron recently uploaded Network Video tool, a project that informed RTP.

In his own words:

In October of 1992, I began to experiment with the Sun VideoPix frame grabber card, with the idea of writing a network videoconferencing tool based upon IP multicast. It was modeled after "vat" – an audioconferencing tool developed at LBL, in that it used a similar lightweight session protocol for users joining into conferences, where you simply sent data to a particular multicast group and watched that group for any traffic from other group members.

In order for the program to really be successful, it needed to compress the video data before putting it out on the network. My goal was to make an acceptable looking stream of data that would fit in about 128 kbps, or the bandwidth available on a standard home ISDN line. I also hoped to produce something that was still watchable that fit in half this bandwidth. This meant I needed approximately a factor of 20 in compression for the particular image size and frame rate I was working with. I was able to achieve this compression and filed for a patent on the techniques I used, later granted as patent US5485212A: Software video compression for teleconferencing.

In early November of 1992, I released the videoconferencing tool "nv" (in binary form) to the Internet community. After some initial testing, it was used to

videocast parts of the November Internet Engineering Task Force all around the world. Approximately 200 subnets in 15 countries were capable of receiving this broadcast, and approximately 50-100 people received video using "nv" at some point in the week.

Over the next couple of months, three other workshops and some smaller meetings used "nv" to broadcast to the Internet at large, including the Australian NetWorkshop, the MCNC Packet Audio and Video workshop, and the MultiG workshop on distributed virtual realities in Sweden.

A source code release of "nv" followed in February of 1993, and in March I released a version of the tool where I introduced a new wavelet-based compression scheme. In May of 1993, I added support for color video.

The network protocol used for "nv" and other Internet conferencing tools became the basis of the Realtime Transport Protocol (RTP), standardized through the Internet Engineering Task Force (IETF), first published in RFCs 1889-1890 and later revised in RFCs 3550-3551 along with various other RFCs that covered profiles for carrying specific formats of audio and video.

Over the next couple of years, work contined on "nv", porting the tool to a number of additional hardware platforms and video capture devices. It continued to be used as one of the primary tools for broadcasting conferences on the Internet at the time, including being selected by NASA to broadcast live coverage of shuttle missions online.

In 1994, I added support in "nv" for supporting video compression algorithms developed by others, including some hardware compression schemes such as the CellB format supported by the SunVideo video capture card. This also allowed "nv" to send video in CUSeeMe format, to send video to users running CUSeeMe on Macs and PCs.

The last publicly released version of "nv" was version 3.3beta, released in July of 1994. I was working on a "4.0alpha" release that was intended to migrate "nv" over to version 2 of the RTP protocol, but this work was never completed due to my moving on to other projects. A copy of the 4.0 alpha code is included in the Network Video tool archive for completeness, but it is unfinished and there are known issues with it, particularly in the incomplete RTPv2 support.

The framework provided in "nv" later went on to become the basis of video conferencing in the "Jupiter multi-media MOO" project at Xerox PARC, which eventually became the basis for a spin-off company "PlaceWare", later acquired by Microsoft. It was also used as the basis for a number of hardware video conferencing projects that allowed sending of full NTSC broadcast quality video over high-bandwidth Ethernet and ATM networks. I also later used some of this code as the basis for "Mediastore", which was a network-based video recording and playback service.

Do you remember the motivations/ideas of the other people on the draft?

We were all researchers working on IP multicast, and helping to create the Internet multicast backbone (aka MBONE). The MBONE was created by Steve Deering (who first developed IP multicast), Van Jacobson, and Steve Casner. Steve Deering and I had the same advisor at Stanford, and Steve ended up going to work at Xerox PARC when he left Stanford, I spent a summer at Xerox PARC as an intern working on IP multicast-related projects and continued to work for them part time while at Stanford and later full time. Van Jacobson and Steve Casner were two of the four authors on the initial RTP RFCs, along with Henning Schulzrinne and myself. We all had MBONE tools that we were working on that allowed for various forms of online collaboration, and trying to come up with a common base protocol all these tools could use was what led to RTP.

Multicast is super fascinating. WebRTC is entirely unicast, mind expanding on that?

Before getting to Stanford and learning about IP multicast, I had a long history working on ways to use computers as a way for people to communicate with one another. This started in the early 80s for me where I ran a dial-up bulletin board system where people could log on and leave messages for one another, both private (sort of the equivalent of e-mail) and public (discussion groups). Around the same time, I also learned about the online service provider CompuServe. One of the cool features on CompuServe was something called a "CB Simulator" where people could talk to one another in real-time. It was all text-based, but it had a notion of "channels" like a real CB radio, and multiple people could see what others typed, as long as they were in the same channel. I built my own version of CB which ran on a timesharing system I had access to which let users on that system send messages to one another in real-time, and over the next few years I worked with friends to develop more sophisticated versions of real-time communication tools on several different computer systems and networks. In fact, one of those systems is still operational, and I use it talk every day to folks I went to college with 30+ years ago!

All of those tools were text based, since computers at the time generally didn't have any audio/video capabilities, but when I got to Stanford and learned about IP multicast, I was intrigued by the notion of using multicast to get something more like a true "radio" where you could send a signal out onto the network that wasn't directed at anyone in particular, but everyone who tuned to that "channel" could receive it. As it happened, the computer I was porting the IP multicast code to what was the first generation SPARC-station from Sun, and it actually had built-in telephone-quality audio hardware! You could digitize sound from a microphone and play it back over built-in speakers (or via a headphone output). So, my first thought was to figure out how to send that audio out onto the network in real-time using IP multicast, and see if I could build a "CB radio" equivalent with actual audio instead of text.

There were some tricky things to work out, like the fact that the computer could only play one audio stream at a time, so if multiple people were talking you

needed to mathematically "mix" multiple audio streams into one before you could play it, but that could all be done in software once you understood how the audio sampling worked. That audio application led me to working on the MBONE and eventually moving from audio to video with "nv".

Anything that got left out of the protocol that you wish you had added? Anything in the protocol you regret?

I wouldn't say I regret it, but one of the big complaints people ended up having about RTP was the complexity of implementing RTCP, the control protocol that ran in parallel with the main RTP data traffic. I think that complexity was a large part of why RTP wasn't more widely adopted, particularly in the unicast case where there wasn't as much need for some of RTCP's features. As network bandwidth became less scarce and congestion wasn't as big a problem, a lot of people just ended up streaming audio & video over plain TCP (and later HTTP), and generally speaking it worked "well enough" that it wasn't worth dealing with RTP.

Unfortunately, using TCP or HTTP meant that multi-party audio and video applications had to send the same data over the network multiple times, to each of the peers that needed to receive it, making it much less efficient from a bandwidth perspective. I sometimes wish we had pushed harder to get IP multicast adopted beyond just the research community. I think we could have seen the transition from cable and broadcast television to Internet-based audio and video much sooner if we had.

What things did you imagine being built with RTP? Do have any cool RTP projects/ideas that got lost to time?

One of the fun things I built was a version of the classic "Spacewar" game which used IP multicast. Without having any kind of central server, multiple clients could each run the spacewar binary and start broadcasting their ship's location, velocity, the direction it was facing, and similar information for any "bullets" it had fired, and all of the other instances would pick up that information and render it locally, allowing users to all see each other's ships and bullets, with ships "exploding" if they crashed into each other or bullets hit them. I even made the "debris" from the explosion a live object that could take out other ships, sometimes leading to fun chain reactions!

In the spirit of the original game, I rendered it using simulated vector graphics, so you could do things like zooming your view in & out and everything would scale up/down. The ships themselves were a bunch of line segments in vector form that I had some of my colleagues at PARC helped me to design, so everyone's ship had a unique look to it.

Basically, anything that could benefit from a real-time data stream that didn't need perfect in-order delivery could benefit from RTP. So, in addition to audio & video we could build things like a shared whiteboard. Even file transfers could benefit from RTP, especially in conjunction with IP multicast.

Imagine something like BitTorrent but where you didn't need all the data going point-to-point between peers. The original seeder could send a multicast stream to all of the leeches at once, and any packet losses along the way could be quickly cleaned up by a retransmission from any peer that successfully received the data. You could even scope your retransmission requests so that some peer nearby delivered the copy of the data, and that too could be multicast to others in that region, since a packet loss in the middle of the network would tend to mean a bunch of clients downstream of that point all missed the same data.

Why did you have to roll your own video compression. Was nothing else available at the time?

At the time I began to build "nv", the only systems I know of that did video-conferencing were very expensive specialized hardware. For instance, Steve Casner had access to a system from BBN that was called "DVC" (and later commercialized as "PictureWindow"). The compression required specialized hardware, but the decompression could be done in software. What made "nv" somewhat unique was that both compression and decompression was being done in software, with the only hardware requirement being something to digitize an incoming analog video signal.

Many of the basic concepts about how to compress video existed by then, with things like the MPEG-1 standard appearing right around the same time "nv" did, but real-time encoding with MPEG-1 was definitely NOT possible at the time. The changes I made were all about taking those basic concepts and approximating them with much cheaper algorithms, where I avoided things like cosine transforms and floating point, and even avoided integer multiplications since those were very slow on SPARC-stations. I tried to do everything I could with just additions/subtractions and bit masking and shifting, and that got back enough speed to still feel somewhat like video.

Within a year or two of the release of "nv", there were many different audio and video tools to choose from, not only on the MBONE but in other places like the CU-SeeMe tool built on the Mac. So, it was clearly an idea whose time had come. I actually ended up making "nv" interoperate with many of these tools, and in a few cases other tools picked up my "nv" codecs, so they could interoperate when using my compression scheme.

SDP

ICE

SRTP

SCTP

DTLS

FAQ

{{<details “Why does WebRTC use UDP?”>}} NAT Traversal requires UDP. Without NAT Traversal establishing a P2P connection wouldn’t be possible. UDP doesn’t provide “guaranteed delivery” like TCP, so WebRTC provides it at the user level.

See [Connecting]({{< ref “03-connecting” >}}) for more info. {{

}}

{{<details “How many DataChannels can I have?”>}} 65536 channels as stream identifier has 16 bits. You can close and open a new one at any time. {{

}}

{{<details “Does WebRTC impose bandwidth limits?”>}} Both DataChannels and RTP use congestion control. This means that WebRTC actively measures your bandwidth and attempts to use the optimal amount. It is a balance between sending as much as possible, without overwhelming the connection. {{

}}

{{<details “Can I send binary data?”>}} Yes, you can send both text and binary data via DataChannels. {{

}}

{{<details “What latency can I expect with WebRTC?”>}} For un-tuned media you can expect sub-500 milliseconds. If you are willing to tune or sacrifice quality for latency developers have gotten sub-100ms

DataChannels support “Partial-reliability” option which can reduce latency caused by data retransmissions over a lossy connection. If configured properly it has been shown to beat TCP TLS connections. {{

}}

{{<details “Why would I want unordered delivery for DataChannels?”>}} When newer information obsoletes the old such as positional information of an object, or each message is independent from the others and need to avoid head-of-line blocking delay. {{

}}

{{<details “Can I send audio or video over a DataChannel?”>}} You could send any data over DataChannel. In a browser case, it is your responsibility to decode the data and pass it to a media player for rendering, where it is automatically done when you use media channels. {{

}}