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 is difficult to use. This book is vendor agnostic, and we have tried to remove any conflicts of interest.

Who is this book 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 Developer who needs 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?
- Technical details about the solution.
- Where to go 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.

Non-commercial 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 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/languages besides JavaScript. You can find servers and domain-specific tools as well for WebRTC. All of these implementations use the WebRTC protocol so 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 but is some of the things you may appreciate during your journey. Don't worry if you don't know some of 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

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.

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). SRTP is used to encrypt media exchanged over RTP, and SCTP is used to send 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 different 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.

{{}} graph TB

webrtc{WebRTC Agent}

sctp{SCTP Agent} dtls{DTLS Agent} ice{ICE Agent} stun{STUN Protocol} turn{TURN Agent} srtp{SRTP Agent} sdp{SDP} rtp{RTP} rtcp{RTCP}

webrtc –> ice webrtc –> dtls webrtc –> srtp webrtc –> sdp webrtc –> sctp

ice –> turn ice –> stun

srtp –> rtcp srtp –> rtp

{{}}

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 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 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 a 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 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 keyes 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, 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 then 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 fmt `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 codecs, Media Descriptions. This is how two peers can understand what 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 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 a RTP Payload Type. Payload types are not static so 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 of 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
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
```

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. If you have more questions please submit a Pull Request!

- Renegotiation
- Simulcast

Why does WebRTC need a dedicated subsystem for connecting?

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

WebRTC doesn't use client/server, 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 IP Versions (IPv4 <-> IPv6).

Despite these diffculties 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 transporting it (no third-party server charging for it).

Lower Latency

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

{{}} graph TB subgraph netb ["Network B (IP Address 5.0.0.2)"] b3["Agent 3 (IP 192.168.0.1)"] b4["Agent 4 (IP 192.168.0.2)"] routerb["Router B"] end

subgraph neta ["Network A (IP Address 5.0.0.1)"] routera["Router A"] a1["Agent 1 (IP 192.168.0.1)"] a2["Agent 2 (IP 192.168.0.2)"] end

pub{Public Internet} routera–>pub routerb–>pub

{{}}

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 too?

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 filterings. 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, addresing 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:

{{}} graph TB subgraph netb ["Network B (IP Address 5.0.0.2)"] b2["Agent 2 (IP 192.168.0.1)"] routerb["Router B"] end

subgraph neta ["Network A (IP Address 5.0.0.1)"] routera["Router A"] a1["Agent 1 (IP 192.168.0.1)"] end

pub{Public Internet}

a1-.->routera; routera-.->pub; pub-.->routerb; routerb-.->b2; {{}}

To make this communication happen you establish a NAT Mapping. Agent 1 uses port 7000 to established 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 forwardinging 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 public IP/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/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 back 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 back 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

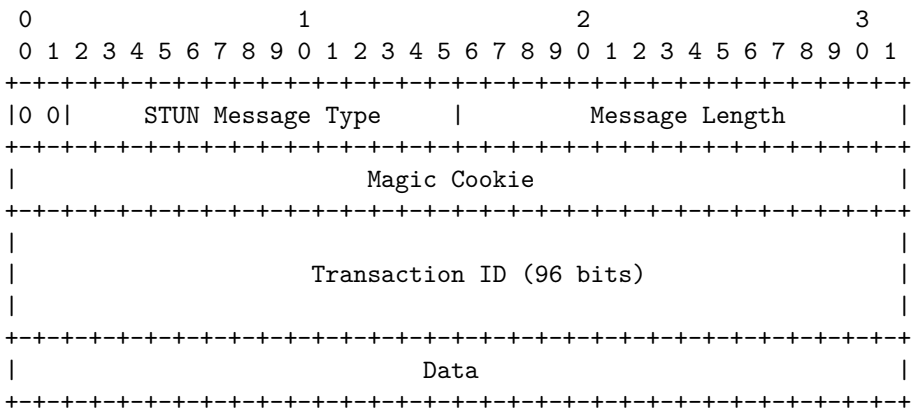
STUN (Session Traversal Utilities for NAT) is a protocol that was created just for working with NATs. This is another technology that pre-dates WebRTC (and ICE!). It is defined by RFC 5389, which also defines the STUN packet structure. The STUN protocol is also used by ICE/TURN.

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

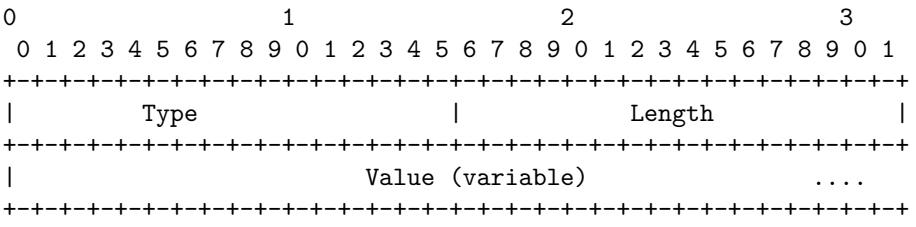
To create a NAT Mapping we make a `Binding Request`. Then the server responds with a `Binding Response`.

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/Port. This is the IP/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 would be useful because you would know ahead of time if direct connectivity was 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 is also important 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 send into to get 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 others can send you traffic via TURN! For each peer you give the `Relay Transport Address` to, you must create a `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/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/decide 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

If the request succeeded, you get a response with the TURN Server with the follow 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/Port is allowed to send inbound traffic.

The remote host needs to give you the IP/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.

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 too. 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/Port. You then send with the ChannelId, and the IP/Port is populated server side. This is the better choice if you are sending lots 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.

One TURN Allocations for Communication {{}} graph TB

subgraph turn [“TURN Allocation”] serverport[“Server Transport Address”] relayport[“Relayed Transport Address”] end

turnclient{TURN Client} peer{UDP Client}

turnclient–>|“ChannelData (To UDP Client)”|serverport serverport–>|“ChannelData (From UDP Client)”|turnclient

peer–>|“Raw Network Traffic (To TURN Client)”|relayport relayport–>|“Raw Network Traffic (To UDP Client)”|peer {{}}

Two TURN Allocations for Communication {{}} graph TB

subgraph turna[“TURN Allocation A”] serverportA[“Server Transport Address”] relayportA[“Relayed Transport Address”] end

subgraph turnb[“TURN Allocation B”] serverportB[“Server Transport Address”] relayportB[“Relayed Transport Address”] end

turnclientA{TURN Client A} turnclientB{TURN Client B}

turnclientA–>|“ChannelData”|serverportA serverportA–>|“ChannelData”|turnclientA

turnclientB–>|“ChannelData”|serverportB serverportB–>|“ChannelData”|turnclientB

relayportA–>|“Raw Network Traffic”|relayportB relayportB–>|“Raw Network Traffic”|relayportA {{}}

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, exchange them, and then attempt to connect!

Two ICE Agents communicate using ICE ping packets (or formally called the connectivity checks), and establish connectivity. After connectivity is established they can send whatever they want. It will feel 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.

Each side must have a `user fragment` and a `password`. These two values must be exchanged before connectivity checks to 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 {{}} graph LR

subgraph agentA["ICE Agent A"] hostA{Host Candidate} serverreflexiveA{Server Reflexive Candidate} relayA{Relay Candidate} end

style hostA fill:#ECECFF,stroke:red style serverreflexiveA fill:#ECECFF,stroke:green style relayA fill:#ECECFF,stroke:blue

subgraph agentB["ICE Agent B"] hostB{Host Candidate} serverreflexiveB{Server Reflexive Candidate} relayB{Relay Candidate} end

hostA — hostB hostA — serverreflexiveB hostA — relayB linkStyle 0,1,2 stroke-width:2px,fill:none,stroke:red;

serverreflexiveA — hostB serverreflexiveA — serverreflexiveB serverreflexiveA — relayB linkStyle 3,4,5 stroke-width:2px,fill:none,stroke:green;

relayA — hostB relayA — serverreflexiveB relayA — relayB linkStyle 6,7,8 stroke-width:2px,fill:none,stroke:blue; {{}}

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.

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 transport layer. That means the the protocol has to handle unreliable delivery. SRTP is specifially 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 a 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!

Cipher Cipher is a series of steps that takes plaintext to ciphertext. The cipher then can 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.

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

Hash Hash 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 a 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.

A simple hash 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.

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.

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

Nonce A nonce is additional input to a cipher. This is so 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/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:

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

`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 exchange 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,

{{}} sequenceDiagram participant C as Client participant S as Server

```
C->>S: ClientHello
Note over C,S: Flight 1

S->>C: HelloVerifyRequest
Note over C,S: Flight 2

C->>S: ClientHello
Note over C,S: Flight 3

S->>C: ServerHello
S->>C: Certificate
S->>C: ServerKeyExchange
S->>C: CertificateRequest
S->>C: ServerHelloDone
Note over C,S: Flight 4

C->>S: Certificate
```

```
C->>S: ClientKeyExchange
C->>S: CertificateVerify
C->>S: ChangeCipherSpec
C->>S: Finished
Note over C,S: Flight 5

S->>C: ChangeCipherSpec
S->>C: Finished
Note over C,S: Flight 6
```

{{}}

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 key-pair. 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.

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 a 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 re-negotiation. They are not used by WebRTC, so they will not be covered here.

SRTP

SRTP is a protocol designed specifially for encrypting RTP packets. To start a 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 a 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, it will ciphertext will be different. This is important to prevent an attacker from identifying patterns or attempting a replay attack.

TODO

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/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 all the sudden 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.

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.

When you attempt to send more data then available bandwidth you will experience network congestion.

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

Jitter Jitter is the fact that `Transmission Time` may vary. Some times you will see packets arrive in bursts. Any piece of hardware along the network path can introduce issues.

Packet Loss Packet Loss is when messages are lost in transmission. The loss could be steady, or it could come in spikes. This is also a common issue, especially on wireless networks!

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, you need to split your data into multiple small 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.

Media 101

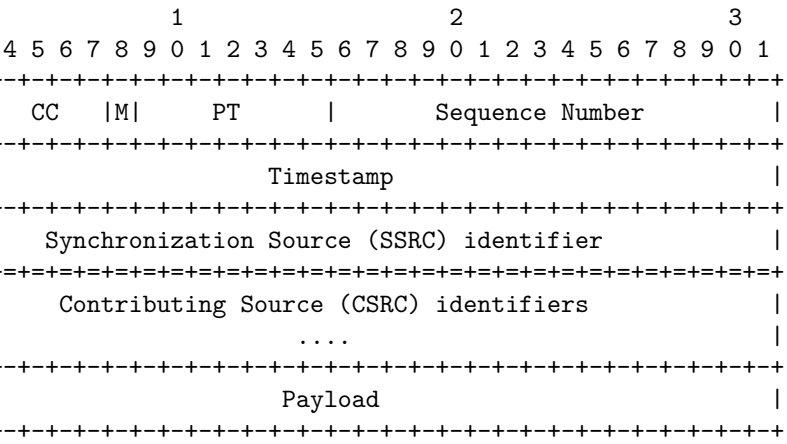
Codec

Frame Types

RTP

Packet Format

Every RTP 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.

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 is up to the user.

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 then 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.

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

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

This is commonly used for talking indicators. Lets 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

Mapping Payload Types to Codecs

RTCP

Packet Format

Every RTCP 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|    RC   |       PT      |             length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Payload                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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.

- Full INTRA-frame Request (FIR) - `192`
- Negative ACKnowledgements (NACK) - `193`
- Sender Report - `200`
- Receiver Report - `201`
- Generic RTP Feedback - `205`

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

Full INTRA-frame Request

This RTCP message notifies the sender that it needs to send a full image. This is for when the encoder is giving you partial frames, but you aren't able to decode them.

This could happen because you had lots of packet loss, or maybe the decoder crashed.

Negative ACKnowledgements

A NACK requests that a sender re-transmits a single RTP Packet. This is usually caused when a RTP Packet is lost, but could also happen because it is late.

NACKs are much more bandwidth efficent then 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.

Sender/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 or basic Congestion Control.

Generic RTP Feedback

How RTP/RTCP solve problems

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

Negative Acknowledgment

Also known as a NACK. This is one method of dealing with packet loss with RTP.

A NACK is a RTCP message sent back to a sender to request re-transmission. The receiver crafts a 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.

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 packet can be significant for NACKs.

Adaptive bitrate and Bandwidth Estimation

A common problem of modern IP networks, both wireless and wired, is unpredictable and unreliable bandwidth. Network conditions are changing dynamically multiple times throughout a session. It is not uncommon to see available bandwidth change drammatically (orders of magnitude) within a second.

Unpredictability in wired networks can be caused by changing demand for bandwidth across the network, routing changes, limitations of transfer medium (fiber channel vs ethernet vs dsl) and more.

In addition to issues seen in wired networks, the nature of radio signal transmission itself, interference from multiple sources, distance to cell towers or Wi-Fi access points and physical obstacles (read walls) are some reasons for unpredictable wireless network characteristics.

WebRTC has several mechanisms to help deliver video/audio signals to the receiver despite changing network conditions. The main idea is to adjust encoding bitrate based on predicted, current, and future available network bandwidth. This ensures that video/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.

REMB A widely supported albeit never fully standardized and now considered deprecated method is called REMB (Receiver Estimated Maximum Bitrate). REMB is a special RTCP packet the receiver sends to the sender, notifying the sender of available bandwidth. There is no(?) standard method defined to estimate bandwidth associated with REMB, so the actual values are implementation dependent. A good starting point for research into details of REMB is the Chrome source code.

The only useful payload in the packet is the bitrate, measured in bits per second. Notable ambiguity and a source of confusion is that REMB is defined as the *total* bitrate, while it is common to see WebRTC libraries use it to constrain video encoding bitrate only.

{{< figure src="/images/05-remb.png">}} REMB

Congestion Control

Experienced WebRTC practitioners say that REMB's approach leaves scars, angry looks and even laughs from Google engineers.

Congestion Control is the act of adjusting the media depending on the attributes of the network. If you don't have a lot of bandwidth, you need to send lower quality video.

Congestion Control improves WebRTC experience by providing a more fine grained control and monitoring of network connection and conditions.

TWCC Transport-Wide Congestion Control (TWCC) is an advanced congestion control specification implemented in most browsers.

TWCC uses a quite simple principle:

{{< figure src="/images/05-twcc-idea.png">}} TWCC

Unlike in REMB, a TWCC receiver doesn't try to estimate it's own incoming bitrate. It just lets the sender know which packets where 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 happenning and the sender must act on it.

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

{{< figure src="/images/05-twcc.png">}} TWCC

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 algrorithm 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.

More sophisticated congestion control algorithms like A Google Congestion Control Algorithm for Real-Time Communication, GCC for short, are built on top of the raw TWCC data. GCC was proposed by Google and implemented in Chrome. 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.

Q: How can I tell that TWCC is supported and enabled?

A: Look at the SDP offer/answer. If you see the lines below, you have TWCC negotiated on your connection:

```
a=extmap:5 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions
```

and

```
a=rtcp-fb:96 transport-cc
```

JitterBuffer

What is Data

Functional Overview

Data channel can deliver any types of data. If you wish, you can send audio or video data over the data channel too. But if you need to playback media in real-time, using media channels (see [Media Communication]({{< ref "06-media-communication.md" >}})) that uses RTP/RTCP protocols are the better options.

What are the applications of data channel?

Applicability of the data channel is unlimited! The data channel solves packet loss, congestion problems and many other stuff for you and provide you with

very simple API to deliver you your data to your peer in real-time. You just need to focus on what your application wants to achieve, and be creative.

Here are some example applications using the data channel: - Real-time network games - Serverless home automation from remote - Text chat - Animation (series of still images) - P2P CDN (Content Delivery Network) - Monitoring data from IoT - Continuous ingestion of time-series data from sensor devices - Real-time image recognition with AI from cameras

Protocol Stack Overview

The data channel is comprised of the following 3 layers: * Data Channel Layer * DCEP (Data channel Establishment Protocol) Layer * SCTP (Stream Control Transmission Protocol) Layer

{{< figure src="/images/06-datachan-proto-stack.png">}}

Data Channel Layer This layer provides API. The API largely mimics that of WebSockets, masking it relatively easy for web developers to repurpose existing client-server websocket based code to work over the P2P datachannel. A single peerconnection can have many datachannels to a peer saving you the bother multiplexing and demultiplexing different data sources and sinks. Just create a datachannel per class of data you need to exchange and the peerconnection does the rest. Unlike websockets datachannels have lables which are assigned by the creator and can be used by recipient to figure out which datachannel is which.

Data Channel Establishment Protocol Layer This layer is responsible for the data channel handshake with the peer. It uses a SCTP stream as a control channel to negotiate capabilities such as ordered delivery, maxRetransmits/maxPacketLifeTime (a.k.a. "Partial-reliability"), send the channel's lable etc.

SCTP Protocol Layer This is the heart of the data channel. What it does includes:

- Channel multiplexing (In SCTP, channels are called "streams")
- Reliable delivery with TCP-like retransmission mechanism
- Partial-reliability options
- Congestion Avoidance
- Flow Control

Data Channel API

Connection / Teardown

Data Channel Options

Flow Control API

Data Channel in Depth

SCTP

Connection establishment flow

Connection teardown flow

Keep-alive mechanism

How does a user message get sent?

TSN and retransmission

Congestion avoidance

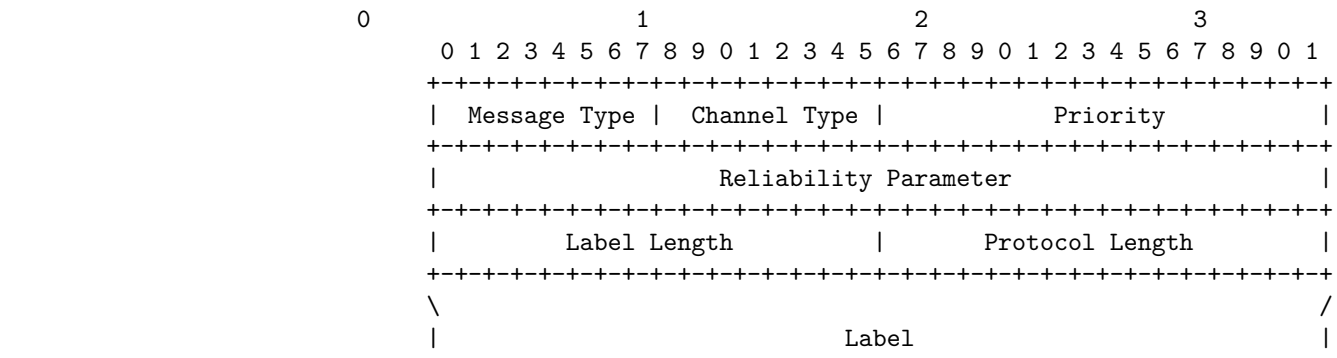
Selective ACK

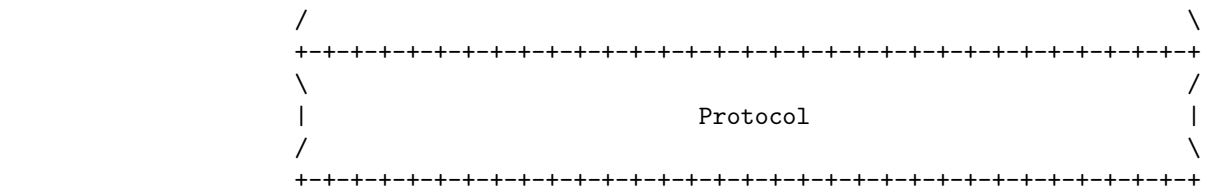
Fast retransmission/recovery

Partial Reliability

DCEP

DCEP is a simple 2 packet protocol which describes the behaviour of a given datachannel. It does not require a network round trip, so subsequent data for the channel can be included in the same packet. This makes establishing datachanels fast and cheap. #### Open/ACK handshake





Here is a list of chanel types and the associated meaning of the reliability parameter that will be applied when packets are lost or misordered.

DATA_CHANNEL_OPEN
Message Type: 0x03

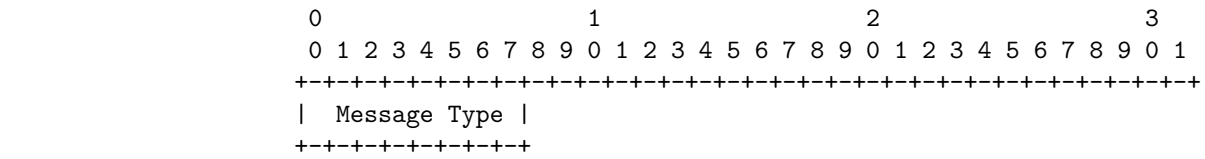
Channel Type	Reliability Parameter
DATA_CHANNEL_RELIABLE	Ignored
DATA_CHANNEL_RELIABLE_UNORDERED	Ignored
DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT	Number of RTX
DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT_UNORDERED	Number of RTX
DATA_CHANNEL_PARTIAL_RELIABLE_TIMED	Lifetime in ms
DATA_CHANNEL_PARTIAL_RELIABLE_TIMED_UNORDERED	Lifetime in ms

Reliable is largely equivelent to TCP.

Reliable_Unordered requires a special mention. It means that when a packet is dropped by the network only the message that the packet is part of will be delayed until it is resent. Other messages, which were sent later may be delivered earlier. You can use this mode when the sequence that messages are received is less important than the speed of their arrival. In this mode message delivery still enforced, but the ordering is not. This mode can be thought of as a halfway house between TCP and UDP.

The remaining modes are equivelent to a UDP based protocols with differing retry/failure rules.

DATA_CHANNEL_ACK
Message Type: 0x02



There is no DCEP NACK. DataChannel opens always succeed, because they are carried on the same SCTP stream as the data which as already suceeded! #### PPID The SCTP PPID for WebRTC DCEP is 50. DCEP messages are processed apart from the rest of the data in the SCTP stream and not passed back up to the web application or service. #### Parameter exchange In theory it is possible to create a datachannel using out-of-band non-DCEP negotiation in SDP. Don't. ## Reference to RFCs https://datatracker.ietf.org/doc/rfc8832/?include_text=1

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 as a cost. Building production grade WebRTC services is challenging. This chapter will try and explain those challenges before you hit them.

By Use Case

The technologies behind WebRTC aren't just for video chatting – since WebRTC is a generic real-time framework, applications are limitless. Some of the most common categories include:

Conferencing

Conferencing was the use case that WebRTC was originally designed for. You can have two users directly connect to each other. Once they are connected they can share their webcams, or maybe their desktop. Participants can send and receive as many streams as they want. They can also add and remove those streams at any time.

Going beyond just media DataChannels are very useful for building a conferencing experience. Users can send metadata or share documents. You can create multiple streams and have multiple conversations going at once.

Conferencing becomes more difficult as more users join the call. How you scale is entirely up to you. The WebRTC topologies section covers this further.

Broadcasting

WebRTC can also be used to broadcast video streams one-to-many.

Remote Control

File-Transfer

A desktop application could be created to capture a screenshot. Once the screenshot is captured on the clipboard of Device A, it could generate a temporary link for another device to access. Once Device B opens the link, a PeerConnection could be created back to Device A, the data could be streamed using the `DataChannel` part of WebRTC and once the transmission is successful the connection could tear down. This would effectively create a peer-to-peer secure way to transfer files over the internet.

Distributed CDN

IoT

When a video doorbell detects movement, it could supply the cameras RTP stream and initiated a new `PeerConnection` with a central server for recording or send a push notification to a mobile device to ask it to connect as a peer. This would establish a real-time communication between the front door and the mobile app. The mobile app could send real time audio back to the doorbell or it could initiate secure remote controls over WebRTC.

Protocol Bridging

WebRTC Topologies

Regardless of whether you use WebRTC for voice, video or DataChannel capabilities, everything starts with a `PeerConnection`. How peers connect to one another is a key design consideration in any WebRTC application, and there are many established approaches.

Client-Server

The low-latency nature of WebRTC protocol is great for calls, and it's common to see conferences arranged in p2p mesh configuration (for low latency), or peering through an SFU (Selective Forwarding Unit) to improve call quality. Since codec support varies by browser, many conferencing servers allow browsers to broadcast using proprietary or non-free codecs like h264, and then re-encode to an open standard like VP8 at the server level; when the SFU performs an encoding task beyond just forwarding packets, it is now called an MCU (Multi-point Conferencing Unit). While SFU are notoriously fast and efficient and great for conferences, MCU can be very resource intensive! Some conferencing servers even perform heavy tasks like compositing (combining together) A/V streams,

customized for each caller, to minimize client bandwidth use by sending only a single stream of all the other callers.

SFU (Selective Forwarding Unit)

MCU (Multi-point Conferencing Unit)

Peer-To-Peer

One-To-One

P2P Mesh

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 it.

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 begining of t

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` (which 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` (which 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 tipically 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 hexdump of the packet content:

  sudo tcpdump 'udp port 19302' -xx

- same but save packets in PCAP (packet capture) file for later inspection

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

  the PCAP file can be opened with the wireshark GUI: `wireshark stun.pcap`

wireshark

webrtc-internals

History

This section is ongoing and we don't have all the facts yet. We are conducting interviews and 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 was the first generation SPARCstation 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 SPARCstations. 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. {{

}}