frozen mountain

Anton Venema
Chief Technology Officer

Anton is a leading expert on Real Time Communications solutions, and the visionary lead architect behind IceLink, WebSync and LiveSwitch.

Setting Up a Connection

The Role of Signaling

Introduction

Nobody would argue that WebRTC has had a significant impact on industries that depend on real-time communications. From education to healthcare to telecommunications, WebRTC jumped into the forefront of business development plans as a cost-effective and user-friendly option for video streaming. In a world where plugins were viewed increasingly as annoying and intrusive security risks, web developers suddenly had a brand new plugin-free tool in their belt allowing them to create immersive applications that brought people together in new and exciting ways.

It sounds great (and it is), but there is a reason that it took so long for the web development community to agree on a standard for peer-to-peer media streaming. It's hard - very hard. Streaming live video from a camera to another device may sound simple, but there are many moving parts in a real-time peer-driven media application. Networks are messy and complicated to navigate. Codecs are often proprietary and incompatible. Each application's requirements are as unique as the next. There is a continual drive to reduce connectivity time and improve performance. It has been years since WebRTC first surfaced, and the web community is still not yet in agreement on all points.

In this series of white papers, we will examine WebRTC and ORTC in detail in the hopes of coming to a better understanding of the technology itself. We will look at the technologies used, the design decisions made, the impact on real-world applications, and how it has and is evolving into new areas.

Signaling Basics

A good place to start is perhaps where the WebRTC API draws a line in the sand, so to speak - signaling. If you already know what signaling is, feel free to skip ahead a bit. For those who are new to real-time communications, signaling is the process by which two or more media endpoints (e.g. a mobile phone, web browser, or server) exchange information with each other outside of a peer connection.

In general, any exchange of real-time messages between peers can be considered signaling. For example, a simple text chat server that uses WebSync to send messages between participants is a signaling server.

Presence notifications, call notifications, etc. are all part of this, and most signaling systems are used for much more than what is required for WebRTC.

In the WebRTC community, however, the term signaling is often used to refer specifically to the initial exchange of "session descriptions" between peers prior to the establishment of a peer connection. If nothing else, signaling has to perform this one job. Signaling would not be required if session descriptions were not required.

Session descriptions are, however, required. To know why, we need to consider a couple points:

1. Generally, the peers do not know each other's capabilities.
2. Generally, the peers do not know each other's network addresses.

Peer Capabilities

Creating a peer connection requires each peer to understand some things about how the connection will be set up and what will be sent over that connection. Specifically, the peers need to know things like:

1. The types of streams to set up (e.g. audio, video, data).
2. The number of network transports required (e.g. one per connection, one per stream).
3. The encryption configuration (e.g. SDES, DTLS).
4. The role of each peer (e.g. active/client vs. passive/server).
5. A way to identify each other (e.g. ICE username fragment and password).
6. For audio and video, the stream direction (e.g. send-only, receive-only, send-receive).
7. For audio and video, the supported codecs (e.g. Opus, VP8, H.264)
8. For some codecs, the format details (e.g. Opus packet time, H.264 profile).

All of these bits of information play a role in how the connection gets set up, who initiates things, and how to handle media flowing over the connection once it is established. By including this information in the session descriptions, the peers can make educated decisions about how to set up their side to work best with the other side.

A clever person might point out that all of this information can be coded ahead of time into the application. Consider a simple audio chat application where the developer has addressed each of these requirements programmatically in advance, e.g.

1. One audio stream.
2. One network transport.
3. DTLS encryption with a hard-coded certificate.
4. Client role goes to the user whose application username appears earlier alphabetically.
5. ICE username fragment and password is derived from the remote application username.
6. Direction is send-receive.
7. Codec is Opus.
8. Opus packet time is 20ms.

Technically, this satisfies all the requirements, and while it may pose a security risk to hard-code a DTLS certificate, there isn't anything fundamentally wrong with this approach except that it is inflexible (which may not be a problem). This approach starts to break down, however, when we consider that the peers also need to know each other's network addresses.

Peer Network Addresses

In a typical client-server scenario, a client opens a connection to a server through the use of some socket or networking API. The client provides the API with an IP address (or DNS hostname, which resolves to an IP address) and port where the server is actively listening for new connections. This doesn't work for peer connections.

In a typical peer-to-peer scenario, neither side knows what the IP address and port of the other is, and so neither side knows where to direct their media flow. Where do we start? It's impossible to do anything if one side doesn't have any network address information for the other side. How can a connection be opened to a server when the server address isn't known?

By gathering local IP addresses and ports ("candidates") ahead of time and including them in the session description, the signaling process allows both sides to send and receive enough information about each other's networks to form candidate pairs, at least one of which will describe a valid route between the two endpoints.

Yet again, a clever person might point out that this information can be coded ahead of time if both of the endpoints have predictable IP addresses (or hostnames) and ports. This is correct, but is a very limited use case and further limits flexibility. While this could be useful for connections between servers, it requires yet more configuration for the server-side applications and can quickly get out of hand in large, scalable environments.

Offers and Answers

A key feature of session descriptions is that they are negotiable. To make this work, signaling requires one side to create and send an "offer" and the other side to create and send an "answer". The offer and answer are both session descriptions, where the offer can be viewed as a requested or proposed session, and the answer can be viewed as an update to or confirmation of the session details, including rejecting certain offered capabilities. The offerer has the opportunity to advertise its desired streams and capabilities, and the answerer is able to factor in its own capabilities in the response, which acts as the definitive arrangement for the connection.

In general, the signaling portion of the call flow follows 3 logical segments:

1. **Offerer.**
   - Create offer (set local description).
   - Send offer to remote peer and wait...
2. **Answerer.**
   - Receive offer (set remote description).
   - Create answer (set local description).
     WebRTC engine can proceed with connection.
   - Send answer to remote peer.
3. **Offerer.**
   - Receive answer (set remote description).
   - WebRTC engine can proceed with connection.

More simply put: A sends offer > B receives offer, sends answer > A receives answer

Once both parties have both a local description and a remote description, the underlying WebRTC engine has everything it needs to establish the connection.

Session Description Protocol

No discussion of offers and answers can avoid the topic of the Session Description Protocol (SDP). SDP was and is a bit of a dividing point in the WebRTC community and it's use as an extension of the API through "munging" was one of the driving factors behind the development of the ORTC specification. Simply put, SDP is a way to describe a session in a text blob that can be easily serialized and sent back and forth between peers. It describes streams, peer capabilities, and peer network addresses in a simple form that has been used for years by SIP and VoIP networks.

Over time, SDP has become extremely robust in its ability to accurately describe the capabilities of each side in a peer connection. Here's a sample SDP blob that describes a WebRTC- and ORTC-compatible audio/video session:

```
v=0
o=- 3867256703365317120 1993119804 IN IP4 127.0.0.1
s=IceLink
t=0 0
a=ice-options:trickle
m=audio 9 UDP/TLS/RTP/SAVPF 111 0 8
c=IN IP4 0.0.0.0
a=ice-ufrag:c1305c9c
a=ice-pwd:106b2df401337e11bb088f19e370e28b
a=candidate:db6a2d72ed7745926dafbf1c7f306f00 1 udp 2122294015 192.168.1.3 64044 typ host
a=fingerprint:sha-256 4B:01:C4:39:2A:A1:E4:84:9D:DA:09:4E:94:1F:C8:50:B8:FC:C2:6B:9D:D3:EA:7B:0B:84:E5:07:6D:9B:40:46
a=setup:active
a=rtcp-mux
a=rtcp:9 IN IP4 0.0.0.0
a=sendrecv
a=rtpmap:111 opus/48000/2
a=fmtp:111 useinbandfec=1
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=ssrc:2517803111 cname:ba4b4170-4067-452b-bd97-de89e7a25795
m=video 9 UDP/TLS/RTP/SAVPF 100 107
c=IN IP4 0.0.0.0
a=ice-ufrag:52264b40
a=ice-pwd:0a6cf043ed7a30d320d229f825683186
a=candidate:db6a2d72ed7745926dafbf1c7f306f00 1 udp 2122294015 192.168.1.3 52620 typ host
a=fingerprint:sha-256 4B:01:C4:39:2A:A1:E4:84:9D:DA:09:4E:94:1F:C8:50:B8:FC:C2:6B:9D:D3:EA:7B:0B:84:E5:07:6D:9B:40:46
a=setup:active
a=rtcp-mux
a=rtcp:9 IN IP4 0.0.0.0
a=sendrecv
a=rtpmap:100 VP8/90000
a=rtcp-fb:100 nack pli
a=rtcp-fb:100 nack
a=rtpmap:107 H264/90000
a=rtcp-fb:107 nack pli
a=fmtp:107 packetization-mode=1;profile-level-id=42e01f
a=rtcp-fb:107 nack
a=ssrc:1143685949 cname:ba4b4170-4067-452b-bd97-de89e7a25795
```

It's a bit verbose, and not all of it is used by WebRTC, but it's broad compatibility with existing SIP infrastructure made it a priority for the WebRTC community to adopt. We can learn a lot about this WebRTC endpoint from the SDP blob:

1. **It wants to run two streams over the connection - one audio and one video:**
   ```
   m=audio ...
   ...
   m=video ...
   ```
2. **It wants to use DTLS for encryption:**
   ```
   a=fingerprint:sha-256 ...
   ```
3. **It wants to take on the client role:**
   ```
   a=setup:active
   ```
4. **It supports the Opus (48000Hz stereo), PCMU (8000Hz mono), and PCMA (8000Hz mono) codecs for audio and the VP8 and H.264 codecs for video:**
   ```
   a=rtpmap:111 opus/48000/2
   a=rtpmap:0 PCMU/8000
   a=rtpmap:8 PCMA/8000
   ...
   a=rtpmap:100 VP8/90000
   a=rtpmap:107 H264/90000
   ```
5. **It supports both picture loss indication (PLI) and generic NACK (negative acknowledgement) feedback for VP8 and H.264:**
   ```
   a=rtcp-fb:100 nack pli
   a=rtcp-fb:100 nack
   a=rtcp-fb:107 nack pli
   a=rtcp-fb:107 nack
   ```
6. **It supports in-band forward error correction for Opus:**
   ```
   a=fmtp:111 useinbandfec=1
   ```
7. **It supports the baseline profile (42) constrained (e0) 3.1 level (1f) for H.264 with packetization mode 1:**
   ```
   a=fmtp:107 packetization-mode=1;profile-level-id=42e01f
   ```
8. **It wants to both send and receive media:**
   ```
   a=sendrecv
   ```
9. **It wants to mux RTP and RTCP traffic over the same network path, but use separate network paths for each stream:**
   ```
   a=rtcp-mux
   ```
10. **It is listening on address 192.168.1.3 and port 64044 for audio and port 52620 for video.**
    ```
    a=candidate:db6a2d72ed7745926dafbf1c7f306f00 1 udp 2122294015 192.168.1.3 64044 typ host
    ...
    a=candidate:db6a2d72ed7745926dafbf1c7f306f00 1 udp 2122294015 192.168.1.3 52620 typ host
    ```

If this was a session description offered to us, we'd have more than enough information to form an answer and start connecting to the remote peer, using signaling to send the answer SDP back to the offering peer so they can finish their own preparations and accept the connection. so they can finish their own preparations and accept the connection.

Trickle ICE

Once the session descriptions have been exchanged, the peers can put together a list of all possible combinations of the local and remote candidates - candidate pairs - each of which represents a possible network path between the two participants. The Interactive Connectivity Establishment (ICE) algorithm then works its magic to test each candidate pair until one or more valid combinations are found, but delving into that goes beyond the scope of this paper. For now, it's worth pointing out that there are three different types of network addresses:

1. Host: the IP address of your client device, usually in the private IP range. This address is often assigned by a local DHCP server on your network.
2. Reflexive: the IP address of the router or gateway that handles outbound Internet requests on your network. This address is often assigned by your ISP.
3. Relayed: the IP address of a TURN server that acts as a "dumb" relay for media in case a direct peer-to-peer connection isn't possible.

These three network address types represent the three different possible ways to talk to a given device. Clients on the same network should be able to talk directly between host addresses, whereas clients on different networks should be able to talk directly between reflexive addresses. In certain cases of symmetric, port-restricted, or otherwise more restrictive firewalls, it might be necessary for one side to use a relayed address.

As long as each peer can gather and include host, reflexive, and relayed candidates in their session description, a connection between the two peers should be possible, barring unavailable or blocked Internet access. Gathering candidates takes time, though, especially with respect to reflexive and relayed candidates which require asynchronous network requests to a STUN and TURN server, respectively. Precious seconds can be lost waiting for these requests to complete so they can be included in the session description.

This is where trickle ICE comes in. Instead of including network candidates in the session description, endpoints that support trickle ICE can send them as separate messages through the signaling layer. This allows the session descriptions to be formed and sent immediately without any delay, while candidates can be sent individually as they are gathered. The offer/answer exchange can be performed earlier, candidate pairs can be formed earlier, and the ICE algorithm can begin earlier (hence "trickle" ICE), all driving down the average time to connect.

Disconnecting

Signaling also often plays a critical role in handling client disconnects. The media path between two peers is often UDP-based, the preferable protocol for streaming media. While UDP is excellent for sending streams of real-time data, it is less useful when it comes to reliably disconnecting from a peer.

An RTCP “BYE” packet is typically sent when a peer leaves a connection, but there is no guarantee that the message will be delivered since it’s sent over an unreliable connection. Additionally, clients can disconnect abruptly and ungracefully for a variety of reasons including sudden loss of power or Internet access. Because of these cases, it is recommended to rely on the signaling path to indicate to a remote peer that the current client is going to disconnect.

Since signaling must be reliable, fault-tolerant and include some form of “dead client” detection, it is a natural choice for notifying remote peers of disconnections.

Wrap-Up

Hopefully you have found this informative in understanding what signaling is and why it’s role in WebRTC-based communications is so critical. As a final note and plug for our products, consider using LiveSwitch, IceLink and WebSync for your WebRTC-based applications.

Our client SDKs feature a consistent, powerful API and support for popular development frameworks/platforms like .NET, Java, macOS, iOS, Android, Windows Phone, Xamarin, Unity, and of course JavaScript. The powerful pipe-driven media engine in LiveSwitch and IceLink allows you to do just about anything imaginable with your audio and video data, and the flexible publish/subscribe architecture in LiveSwitch and WebSync lets you deliver real-time text and binary messages to your entire client base quickly and efficiently.