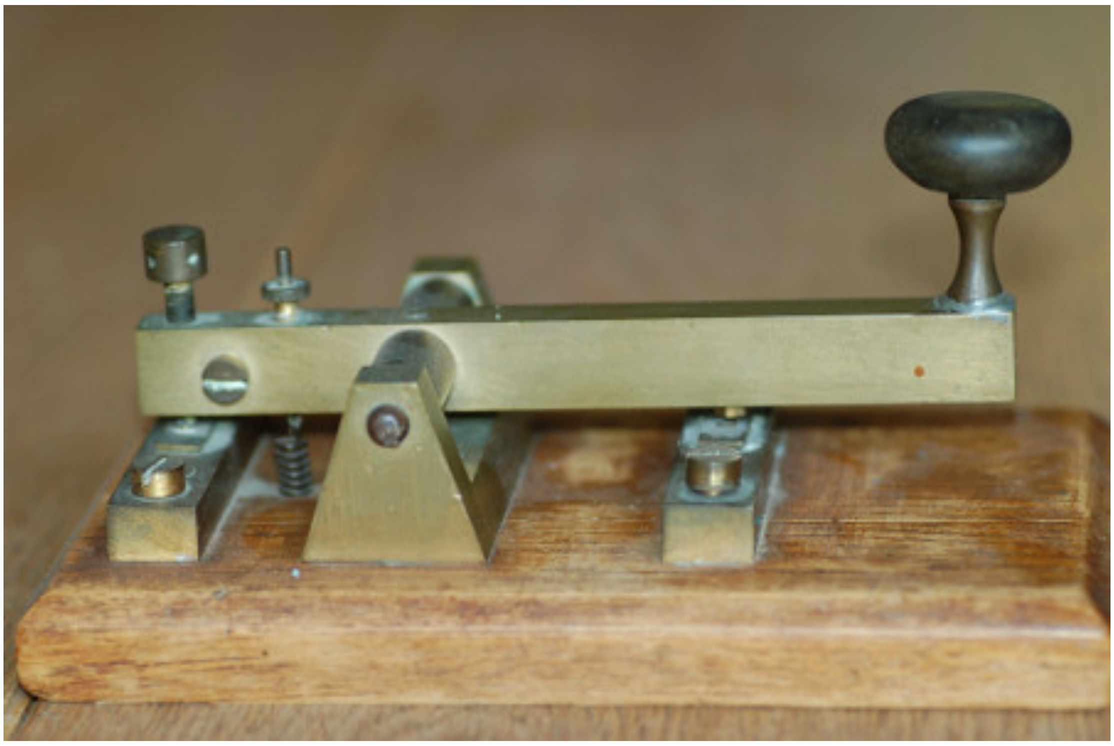
WebRTC Signaling Protocols and WebRTC Transport Protocols Demystified

A refresher on what I've written in 2014 (here and here).



WebRTC as a protocol comes without signaling. This means that you as a developer will need to take care of it.

The first step will be selecting the protocol for it. Or more accurately – two protocols: transport and signaling. In many cases, we don't see the distinction (or just don't care), but sometimes, they are important. A recent question in the comments section of one of the two posts mentioned here in the beginning, got me to write this explanation. Probably yet again.

WebRTC Transport Protocols and Browsers

This actually fits any browser transport protocol.

A transport protocol is necessary for us to send a message from one device to another. I don't care what is in that message or how the message is structured at this point – just that it can be sent – and then received.

HTTP/1.1

5 years ago browsers were simple when it came to transport protocols. We essentially had HTTP/1.1 and all the hacks on top of it, known as XHR, SSE, BOSH, Comet, etc. If you are interested in the exact mechanics of it, then leave a comment and I'll do my best to explain in a future post (though there's a lot of existing explanation around the internet already).

I call the group of solutions on top of HTTP/1.1 workarounds. They make use of HTTP/1.1 because there was no alternative at the time, but they do it in a way that makes no technical sense.

Oh – and you can even use REST to some extent, which is again a minor "detail" above HTTP/1.1.

Since then, three more technique materialized: WebSocket, WebRTC and recently HTTP/2.

WebSocket

The WebSocket was added to do what HTTP/1.1 can't. Provide a bidirectional mechanism where both the client and the web server can send each other messages. What these messages are, what they mean and what type of format they follow was left to the implementer of the web page to decide.

There's also socket.io or the less popular SockJS. Both offer client side implementations that simulate WebSocket in cases it cannot be used (browser or proxy doesn't support it). If you hear that the transport is socket.io – for the most part you can just think about it as WebSocket.

When your WebSocket work great, they are great. But sometimes it doesn't (more on that below, under the HTTP/2 part).

WebRTC’s Data Channel

To some extent, the Data Channel in WebRTC can be used for signaling.

Yes. You’ll need to negotiate IP addresses and use ICE first – and for that you’ll need an additional layer of signaling and transport (from the list in this post here), but once connected, you can use the data channel for it.

This can be done either directly between the two peers, or through intermediaries (for multiple reasons).

Where would you want to do that?

1. To reduce latency in your signaling – this is theoretically the fastest you can go
2. To reduce load on the server – now it won’t receive all messages just to route them around – you’ll be sending it things it really needs
3. To increase privacy – not sending messages through the server means the server can’t be privy to their content – or even the fact there was communication

For the most part, this is quite rare as transport for signaling in WebRTC.

HTTP/2

I’ve written about HTTP/2 before. Since then, HTTP/2 has grown in its popularity and spread.

HTTP/2 fixes a lot of the limitations in HTTP/1.1, which can make it a good long-term candidate for transport of signaling protocols.

A good read here would be Allan Denis' writeup on how HTTP/2 may affect the need for WebSocket.

WebRTC Signaling Protocols

Signaling is where you express yourself. Or rather your service does. You want one user to be able to reach out to another one. Or a group of people to join a virtual room of sorts. To that end, you decide on what types of messages you need, what they mean, how they look like, etc.

That's your signaling protocol.

As opposed to the transport protocol, you aren't really limited by what the browser allows, but rather by what you are trying to achieve.

Here are the 3 main signaling protocols out there in common use with WebRTC:

SIP

I hate SIP.

Never really cared for it.

It has its uses, especially when it comes to telephony and connecting to legacy voice and video services.

Other than that, I find it too bloated, complex and unnecessary. At least for most of the use cases people approach me with.

SIP comes from the telephony world. Its main transport was UDP. Then TCP and TLS were added as transport protocols for it. Later on SCTP. You don't care about any of these, as you can't really access them directly with a browser. So what was done was to add WebSocket as a SIP transport and just call it "SIP over WebSocket". Before WebRTC got standardized (it hasn't yet), SIP over WebSocket got standardized and already has an RFC of its own. Why is it important? Because the only use of SIP over WebSocket is to enable it to use WebRTC.

So there's SIP. And if you know it, like it or need it. You can use it for your WebRTC signaling protocol.

XMPP

I hate XMPP.

Not really sure why. Probably because any time I say something bad about it, a few hard core fans/followers/fanatics of XMPP come rushing in to its rescue in the comments section. It makes things fun.

XMPP has a worldview revolving around presence and instant messaging, and use cases that need it can really benefit from it – especially if the developer already knows XMPP and what he is doing.

If you like it enough – make sure to slam me in the comments – you'll find their section at the end of this post…

Proprietary

I hate NIH. And yet a proprietary signaling protocol has a lot of benefits in my view.

In many cases, you just want to get the two darn users into the "same page". Not much more. I know I am dumbing it down, but the alternative is to carry around you extra protocol messages you don't need or intend using.

In many other cases, you don't really want to add another web server to handle signaling. You want your web server to host the whole site. So you resolve into a proprietary signaling protocol. You might not even call it that, or think of it as a signaling protocol at all.

How to Choose?

Always start from the signaling protocol.

If there’s reason to use SIP due to existing infrastructure or external systems you need to connect to – then use it. If there’s no such need, then my suggestion would be to skip it.

If you like XMPP, or need its presence and instant messaging capabilities – then go use it.

If the service you are adding WebRTC to already has some logic of its own, it probably has signaling in there. So you just add the relevant messages you need to that proprietary signaling.

In any other case, my advice would be to use a proprietary signaling solution that fits your exact need. If you’re fine with it, I’d even go as far as picking a SaaS vendor for signaling.