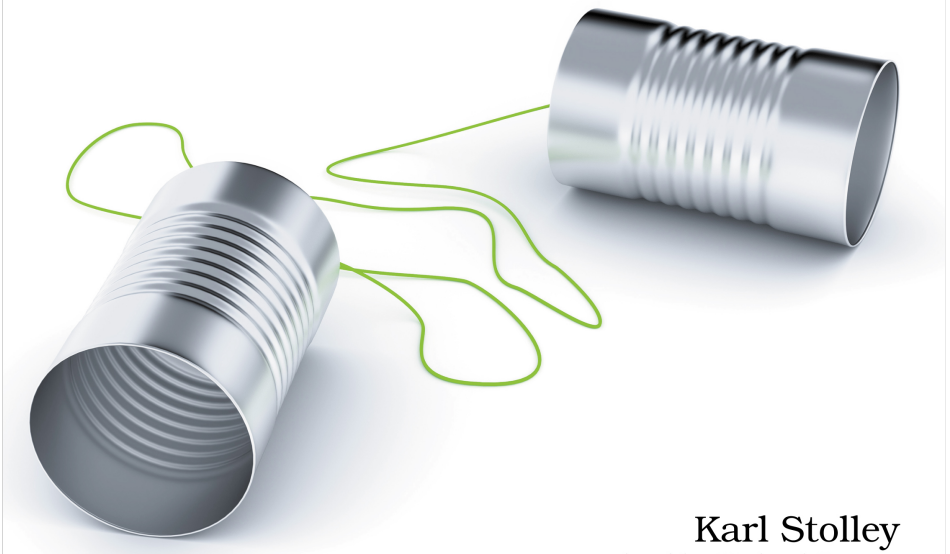
The Pragmatic Programmers

Programming WebRTC

Build Real-Time Streaming Applications for the Web



Karl Stolley
edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Setting Up the Peer Connection

All right. Your video-call app is coming together. You've prepared the UI and set up the signaling channel for users to join whenever they are ready, and now you've got streaming video set up on the self side of the connection. It's time to put all of those pieces together and get a peer connection set up and established using WebRTC.

Let's kick things off by returning to the top of main.js. Once there, you can modify the $self object to include an rtcConfig property set to null. Eventually, $self.rtcConfig will include some important WebRTC configuration values (see Configuring a WebRTC App for Public Deployment, on page ?). But for testing WebRTC on a local network, initially setting the configuration to null will suffice. Immediately below $self, declare a new $peer variable and assign it an object literal—just like $self. All $peer should contain for now is a connection property to hold onto an instance of the RTCPeerConnection object:

demos/basic-p2p/js/main.js

```
const $self = {
  rtcConfig: null,
  mediaConstraints: { audio: false, video: true },
};

const $peer = {
  connection: new RTCPeerConnection($self.rtcConfig),
};
```

And that's it! App finished.

Okay, not really. Not by a long shot. I'm just being mean. There's so much more to come. Those few lines of code *are* accomplishing a lot, though. That's especially true for the $peer.connection property and its reference to a new RTCPeerConnection instance, which is the key piece of everything left to build.

But before we leave the enchanted land of variable assignment and make a triumphant return to the kingdom of callback function definitions, let's set a few more properties on $self, just below the rtcConfig property we added:

demos/basic-p2p/js/main.js

```
const $self = {
  rtcConfig: null,
  isPolite: false,
  isMakingOffer: false,
  isIgnoringOffer: false,
  isSettingRemoteAnswerPending: false,
  mediaConstraints: { audio: false, video: true },
};
```

We'll return to each of those new properties in time. Their purpose will be to track various states on $self as a WebRTC connection is established. But the one that is most interesting for the time being is the first one: $self.isPolite, which is initially set to false. That means we've admitted to the world that each of us, as $self, is surly and rude. But why?

Achieving Asymmetric Function from Symmetric Code

You're about to discover many mind-bending qualities of WebRTC. The primary source of its many mind-benders is loading the same codebase—HTML, CSS, and JavaScript—in the browsers on both ends of a WebRTC call. That means the code is symmetric: each browser loads the exact same code. There's no need to goof around writing server-side logic that sends different code to different peers.

As we established in Designing Peer-to-Peer UI Patterns, on page ?, everyone on the call is just a joiner. That choice frees us up to maintain just a single symmetric WebRTC codebase. Like the symmetric joiner interface it supports, our codebase will provide both peers (and eventually, in Chapter 6, Managing Multipeer Connections, on page ?, *all* peers) with the ability to initiate and answer offers to connect over WebRTC.

"But wait," you might say. "Why are we suddenly talking about initiating and answering 'offers to connect'? Didn't we explicitly set up a joiner pattern, rather than a caller pattern?" Yes, yes we did. The only real piece of UI in our interface—the Join Call button—means we aren't going to worry at all about who's calling whom. The peers just join the call. Your button doesn't lie.

But just because *we're* not going to worry about caller and answerer roles doesn't mean that WebRTC won't. I hate to talk about WebRTC behind its back like this, but you should know that it worries quite a lot—about caller and answerer roles and many other things that we'll get to soon enough. The RTCPeerConnection instance that we're hanging onto in $peer.connection is basically a giant worrywart. And like most worrywarts, it doesn't quite know how to cope on its own with all the things that might go wrong. We almost have to be like good therapists and give RTCPeerConnection some solid coping strategies to help manage its many worries.

That brings us back to the $self.isPolite value: it represents the core principle behind establishing a worry-free WebRTC connection. For each set of two peers connecting over your app, one peer will be polite ($self.isPolite === true), and one will be impolite ($self.isPolite === false). So even though the same code appears in both peers' browsers, each peer will behave a little differently,

being either polite or impolite, depending on the circumstances of the call. And that's how we'll be able to pull off asymmetric behavior from symmetric code.

Establishing the Polite Peer

That all might sound reasonable. But if the code is still *exactly* the same in both browsers, including the opening lines of main.js that set $self.isPolite to false, what sorcery is required to establish that one of the peers is polite?

We can consider a few different approaches to establish politeness. We could write a function that plays a virtual game of rock-paper-scissors between the two peers over the signaling channel before they connect. Or we could make it weird and administer a short personality test to determine which of the two peers really *is* the rude one.

Or we could establish politeness based on the order peers join the call. The first person to click Join Call will be polite, being prompt and punctual (okay, actually, they'll be first to connect to the signaling channel), and the second person to join will be impolite. Probabilistically speaking, there is a very low chance that both peers might join the room at exactly the same fraction of a second. But it's so improbable that we don't have to worry about it.

Recall that the signaling channel broadcast emits a connected peer event. The broadcast emit is special, because it gets sent to everyone on the namespace *except* the person who triggered the event (otherwise, connecting peers would hear their own connected peer events). Every time a peer joins the namespaced signaling channel, the connected peer event fires. You've already written a callback function in the main.js file to handle the connected peer event. Let's open up its definition, and add a single line that sets $self.isPolite to true:

demos/basic-p2p/js/main.js

```
function handleScConnectedPeer() {
  $self.isPolite = true;
}
```

Here again, we appear to end up with another symmetric paradox: both peers join the call, meaning both peers will trigger the signaling server to broadcast the connected peer event to the other peer. That's true.

The trick is that when the first peer joins the call, the connected peer event winds up in Zen koan territory: *If a tree falls in the forest and no one is around to hear it, does it make a sound?* When the first peer triggers the connected peer broadcast-emit event, *the other peer isn't around to receive it*. The second peer hasn't connected yet, and likely will not connect in the nanosecond timeframe required to receive the event.

But—when the *second* peer connects, connected peer fires again—only this time, the first peer *is* around to receive it. And so the first peer becomes the polite one ($self.isPolite === true, as set in the handleScConnectedPeer() callback function), while the second peer stays impolite ($self.isPolite === false, as set near the top of the main.js file). Even though the second peer also has the handleScConnectedPeer function available, it will never execute, because the second peer will not hear a connected peer event.

Adding Peer-Connection Callbacks

Hold onto the fact that one peer on a call is polite, and the other is impolite, while you stack up a whole new collection of events tied to the RTCPeerConnection object. Find the "WebRTC Functions and Callbacks" area of the main.js file, and let's write these events inside their own wrapper function like we did with the signaling channel events on Preparing Placeholders for the Remaining Signaling Callbacks, on page ?. We'll execute this registerRtcCallbacks() function when a user connects to the signaling channel:

demos/basic-p2p/js/main.js

```
function registerRtcCallbacks(peer) {
  peer.connection.onnegotiationneeded = handleRtcConnectionNegotiation;
  peer.connection.onicecandidate = handleRtcIceCandidate;
  peer.connection.ontrack = handleRtcPeerTrack;
}
```

Those events use different syntax for registering callbacks from what we have seen so far. Because these three events—negotiationneeded, icecandidate, and track—are so commonly used, the RTCPeerConnection object gives us shorthand properties for each of them, prefixed with on. (You probably have seen this before with the DOM: you can use domObj.addEventListener('click', handleClick) or the shorthand domObj.onclick = handleClick. Both syntaxes achieve the same thing.)

Below the registerRtcCallbacks() definition, you can drop in the definition of handleRtcPeerTrack(). Because adding media tracks is necessary only for apps that include streaming media, we'll keep that function definition with the application-specific code:

```
function handleRtcPeerTrack() {
  // TODO: Handle peer media tracks
}
```

The onnegotiationneeded and onicecandidate events, however, are generic across any WebRTC apps you might build, so we'll isolate them in the "Reusable WebRTC Functions and Callbacks" area of the JavaScript file:

```
/**
 *  Reusable WebRTC Functions and Callbacks
 */
async function handleRtcConnectionNegotiation() {
  // TODO: Handle connection negotiation
}
function handleRtcIceCandidate() {
  // TODO: Handle ICE candidates
}
```

We'll build out those two reusable functions here, and return to the app-specific handleRtcPeerTrack() function later.

The handleRtcConnectionNegotiation() function will use await in its body, so we prefix it with async. It awaits a promise-based RTC peer connection method, setLocalDescription(), which prepares an offer according to the Session Description Protocol (SDP; see Making an Offer They Can't Refuse, on page ?). SDP offers include details on media capabilities on the self side of the call. While the body of the handleRtcConnectionNegotiation() function awaits the offer returned by setLocalDescription(), $self.isMakingOffer is set to true (recall we initially set the isMakingOffer property to false on $self). Once the offer—held in $peer.connection.localDescription—has been sent over the signaling channel, $self is no longer in the process of making an offer, so $self.isMakingOffer gets set back to its initial state, false:

demos/basic-p2p/js/main.js

```
async function handleRtcConnectionNegotiation() {
  $self.isMakingOffer = true;
  console.log('Attempting to make an offer...');
  await $peer.connection.setLocalDescription();
  sc.emit('signal', { description: $peer.connection.localDescription });
  $self.isMakingOffer = false;
}
```

Now don't be fooled by await, in that function or anywhere else: under most conditions, the wait is less than a blink of an eye. When we get to handling SDP offers (and answers) in Building Connection Logic to the "Perfect Negotiation" Pattern, on page ?, you'll see that blinks of an eye can still matter—especially to a worrywart WebRTC peer connection.

The handleRtcConnectionNegotiation() function generates and sends SDP offers, and the handleRtcIceCandiate() callback must do roughly the same for ICE candidates. I know, that name in all caps makes me think of awful things, too. But ICE in this case stands for Interactive Connectivity Establishment,[2] an IETF-defined protocol that enables browsers to describe how they can be reached

2. https://tools.ietf.org/html/rfc8445

Which Browsers Does This Code Support?

Elsewhere in the book, you've read that browser support for WebRTC is very good. But support isn't always uniform for all features—that's the case for any web API. Browsers must support two baselines for your code to work: RTCPeerConnection methods like setLocalDescription() must return a Promise (allowing them to work with await), and the standardized navigator.mediaDevices.getUserMedia() method must be available.

Those two features were implemented at different times in different browsers, but ultimately the code you're writing here works in at least Chrome 53, Edge 79, Firefox 66, and Safari 11. Those are all pretty ancient desktop browsers by almost any measure.

Safari lags all other browsers in regard to one feature: implicit rollback on setRemoteDescription, which didn't arrive until Safari 15.4 (elsewhere it showed up in Chrome 80, Edge 80, and Firefox 70). You can read about accommodating older Safaris in Appendix 1, Connection Negotiation in Legacy Browsers, on page ?. But apart from that, we'll write any other fallback code as needed throughout the book.

over the internet or a local network. While the offer in localDescription describes the media that will flow over the connection, ICE candidates describe possible peer-to-peer routes over the network. WebRTC needs both to establish a connection.

The handleRtcIceCandiate() callback is less complicated to write. Whenever a candidate becomes available, we just need to send it over the signaling channel, attached again to the signal event.

demos/basic-p2p/js/main.js

```
function handleRtcIceCandidate({ candidate }) {
  console.log('Attempting to handle an ICE candidate...');
  sc.emit('signal', { candidate: candidate });
}
```

Destructuring Assignment

Something might look odd to you in the handleRtcIceCandiate() callback function definition: the { candidate } in curly braces that gets passed into the function as an argument. Ordinarily, when we see curly braces surrounding a value, they look more like what gets sent over the signaling channel: { candidate: candidate }, with one value and another separated by a colon, :. Good, old-fashioned object literals, just like Grandmother used to make.

A newer feature available in JavaScript syntax is something called *destructuring assignment*. It's a shorthand way of pulling a value out of an object literal and assigning the value to its own variable. Here's a simple example:

```
const obj = { one: 1, two: 2 };
```

If we were interested in assigning the value of one to its own variable, we could write const one = obj.one; But destructuring assignment allows us to write this instead:

```
const {one} = obj;
console.log(one); //-> 1
```

In short, the value in the curly braces gets assigned as a standalone representation of whatever matching property exists in the object—assuming the property exists. If the property doesn't exist, the value is undefined:

```
const {three} = obj;
console.log(three); //-> undefined
```

So what function handleRtcIceCandidate({ candidate }) does is extract the value of candidate from the chunk of data returned by $peer.connection.onicecandidate and create a local candidate variable that we can use within the function itself.

Sending Media Tracks

We should test out those two RTC callbacks. The onnegotiationneeded event kicks off the establishment of a WebRTC call. But it's adding media to the peer connection that causes onnegotiationneeded to fire.

While you've already set your local stream to appear on the self <video> element, you need to also add the tracks from that stream to the peer connection so that the remote peer can ultimately receive them. We'll handle the receiving side in Receiving Media Tracks, on page ?. For now, let's write the track-sending logic as another little reusable function, below the other user-media functions:

demos/basic-p2p/js/main.js

```
function addStreamingMedia(stream, peer) {
  if (stream) {
    for (let track of stream.getTracks()) {
      peer.connection.addTrack(track, stream);
    }
  }
}
```

We'll pass the peer instance and the self media stream into that function. Assuming a media stream is set on self, a for loop runs through the tracks and adds them to the peer connection. Don't succumb to a false sense of *déjà vu* here: RTCPeerConnection has an addTracks() method, just like MediaStream does—as we saw in Requesting User-Media Permissions, on page ?. Here, though, we're adding media tracks to the call, not to a MediaStream.

While we could call addStreamingMedia() along with registerRtcCallbacks() directly inside of the handleScConnect() callback, let's take an opportunity to future-proof the code a bit. Streaming media is one possible feature of a WebRTC call. As you will see in Chapter 4, Handling Data Channels, on page ?, data channels represent other features. We can write a wrapper function called establishCallFeatures() which wraps registerRtcCallbacks() together with any app-specific call features, like adding streaming user-media. Add this function definition in the "Call Features & Reset Functions" area of the JavaScript file:

demos/basic-p2p/js/main.js

```
/**
 *  Call Features & Reset Functions
 */

function establishCallFeatures(peer) {
  registerRtcCallbacks(peer);
  addStreamingMedia($self.mediaStream, peer);
}
```

Note that it is essential to register the WebRTC callbacks before adding media or any other features to the connection. Adding the media to the RTCPeerConnection instance triggers the negotiationneeded event, which in turn sets the entire connection-negotiation process in motion. If media or other features are added before a callback is set to handle negotiationneeded, the event will fire without executing a callback, leaving users unable to initiate a WebRTC connection. So be sure to call registerRtcCallbacks() at the very top of the establishCallFeatures() function definition.

With the establishCallFeatures() function finished, call it from within the handleScConnect() callback, which is now complete, generic, and portable across any peer-to-peer WebRTC app you ever want to write:

demos/basic-p2p/js/main.js

```
function handleScConnect() {
  console.log('Successfully connected to the signaling server!');
➤ establishCallFeatures($peer);
}
```

Go ahead and refresh your browser and click the Join Call button. You'll now see some new messages logged to the console: one announcing "Attempting to make an offer…," followed by more than one announcing "Attempting to handle an ICE candidate…."

With the handleRtcConnectionNegotiation() and the handleRtcIceCandidate() callback functions defined and now tested out, you now have all of the sending logic in place for establishing a peer connection. The big piece remaining is the

receiving logic, which will handle descriptions and ICE candidates coming in over the signaling channel.