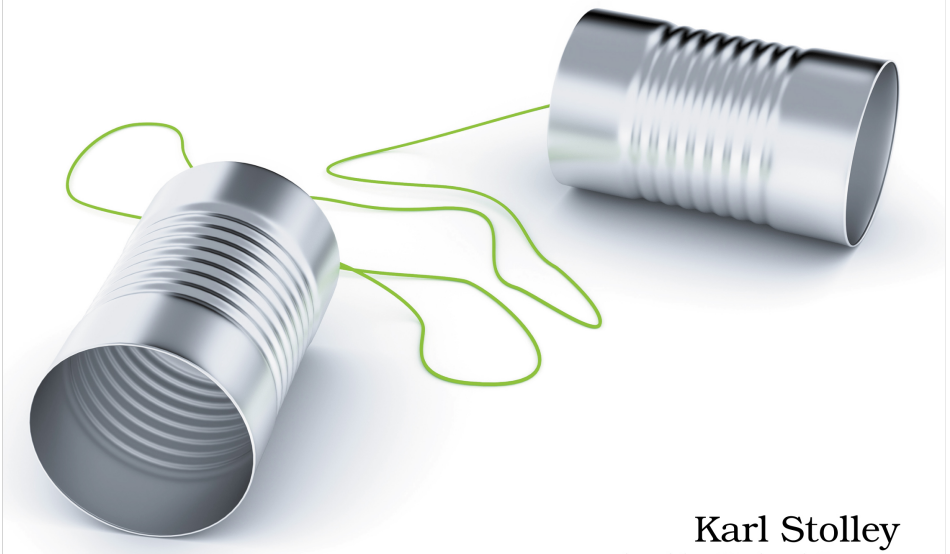
The Pragmatic Programmers

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.

Restructuring WebRTC Callbacks with Closures

So far, rewriting the peer-to-peer code to work on multipeer calls has been pretty straightforward. Well, fairly straightforward. Fine—*straightforwardish*, the way an alien abduction is straightforward. However you want to spin it, your work towards a multipeer WebRTC app has required shifting some logic around on the signaling callbacks and introducing a new `id` variable to a number of functions.

By contrast, the revisions you'll make to your WebRTC callbacks require rethinking your approach to callbacks entirely, especially how the callbacks themselves are registered.

Let's step back and think about peer IDs for a moment: each of your WebRTC callbacks needs to be associated with a specific peer ID's RTCPeerConnection instance. Accessing IDs wasn't a problem with signaling callbacks: peer IDs are included in all data returned by the signaling channel's events. But neither the RTCPeerConnection object nor the data returned by its events have any access to peer IDs. And that's a problem, because unique peer IDs are central to the way we're architecting multipeer, mesh-networked calls.

Providing the correct ID value to each WebRTC callback that needs it is a trickier proposition than it might sound. But it's not impossible. Let's start big-picture and open up the registerRtcCallbacks() function, modify it to take an id argument that will be passed into the function from within establishCallFeatures(), and prepare each of its event assignments so that we're passing in the ID for a specific peer. Remember that establishCallFeatures() is executed for each peer on a call, thanks to the logic you wrote a moment ago in Fleshing out the Skeletal Signaling Callbacks, on page ?:

demos/multipeer/js/main.js

```
/**
 *  WebRTC Functions and Callbacks
 */

function registerRtcCallbacks(id) {
  const peer = $peers.get(id);
  peer.connection
    .onconnectionstatechange = handleRtcConnectionStateChange(id);
  peer.connection
    .onnegotiationneeded = handleRtcConnectionNegotiation(id);
  peer.connection
    .onicecandidate = handleRtcIceCandidate(id);
  peer.connection
    .ontrack = handleRtcPeerTrack(id);
}
```

Just like you did with addStreamingMedia(), you're setting up a function-scoped peer variable inside registerRtcCallbacks(). But something probably looks off about those rewritten callback assignments. In Writing Named Functions as Callbacks, on page ?, you read about how it's a very big deal to make sure that callback functions are passed in or assigned by reference, without parentheses. But here, not only are you calling functions with parentheses, but you're even passing each of them an id value. That will blow up all of the peer-to-peer WebRTC callbacks you wrote previously, which destructured data returned by the WebRTC events that triggered them.

That's a big problem: the rules have not changed on callback assignments. The callbacks assigned to WebRTC events must still reference functions to be executed when each associated event fires—now with the added complexity of firing on a specific peer.connection instance, too.

What we need to do, then, is restructure all the peer-to-peer WebRTC callback functions to create closures.[10] In other words, each of the existing WebRTC callbacks must be rewritten so as to return a function when executed. When a WebRTC event fires, it will execute the returned function, which must be capable of receiving and acting on the WebRTC event's data.

Let's try expressing those abstract ideas as real code. Right below the registerRtcCallbacks() function definition, find the handleRtcPeerTrack() callback. Think back to how you wrote that function to display the media streaming in from a single remote peer:

```
// main.js, peer-to-peer logic
function handleRtcPeerTrack({ track }) {
  console.log(`Handle incoming ${track.kind} track...`)
  $peer.mediaTracks[track.kind] = track;
  $peer.mediaStream.addTrack(track);
  displayStream($peer.mediaStream, '#peer');
}
```

We must do two things to rewrite that callback to create a closure: first, we'll grab the ID argument that's being passed into handleRtcPeerTrack() from within registerRtcCallbacks(). And then we'll change the body of handleRtcPeerTrack() to return a function—an anonymous but otherwise identical version of the original callback that you wrote—which will be called when the track event actually fires the peer connection for that ID value:

demos/multipeer/js/main.js

```
function handleRtcPeerTrack(id) {
  return function({ track }) {
    const peer = $peers.get(id);
    console.log(`Handle incoming ${track.kind} track from peer ID: ${id}`);
    peer.mediaTracks[track.kind] = track;
    peer.mediaStream.addTrack(track);
    displayStream(peer.mediaStream, id);
  };
}
```

10. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

If you've not worked with closures before, the idea behind them is that the returned function is capable of holding onto the values of any variables, like id, from the scope of the outer function: in this case, handleRtcPeerTrack(). The anonymous function returned by handleRtcPeerTrack() is what's actually assigned to the peer.connection.ontrack event property. Note that the anonymous function is still neatly destructuring the track from the event, too.

In slightly more concrete terms: the new function returned by handleRtcPeerTrack() must fire on the track event for a specific remote peer. Preserving the reference to that peer's ID is made possible by the closure. The ID value makes it possible to continue to preserve the tracks and stream on the peer's record in the $peers map. The peer's ID itself must also be passed into the displayStream() function you wrote back on page ?.

In slightly more nerdy terms: the inner anonymous function has access to the outer function's lexical scope, including the id value, at the moment the anonymous function is defined. That's the magic of closures in JavaScript: we can preserve custom values for use inside a callback function without messing with the set structure of arguments passed into the callback at runtime.

Enough nerding out. Let's keep going. If you're still feeling uncertain about closures, that's okay. Their behavior should become clearer as you write a few more. Find the "Reusable WebRTC Functions and Callbacks" area of your main.js file and rewrite those three callbacks to use closures, too. You can go in order, and prove to yourself what a champ you are by starting with handleRtcConnectionNegotiation(), which also involves the signaling channel:

demos/multipeer/js/main.js

```
/**
 *  Reusable WebRTC Functions and Callbacks
 */
function handleRtcConnectionNegotiation(id) {
  return async function() {
    const peer = $peers.get(id);
    const self_state = peer.selfStates;
    self_state.isMakingOffer = true;
    await peer.connection.setLocalDescription();
    sc.emit('signal',
      { recipient: id, sender: $self.id,
        signal: { description: peer.connection.localDescription } });
    self_state.isMakingOffer = false;
  };
}
```

Here you're making use of the self states associated with a specific peer ID, which for the sake of brevity gets its own variable assignment (self_state) for use in the connection-negotiation callback's logic.

Beyond that adjustment in how you reference self states, the most significant change to handleRtcConnectionNegotiation() is on its call to the sc.emit() method. In the peer-to-peer code, that call looked like this:

```
// main.js
// snip, snip...
  sc.emit('signal',
    { description: $peer.connection.localDescription });
// snip, snip...
```

But in a multipeer setup, it's necessary to include the recipient and sender values for properly routing signals over the signaling channel whenever you call sc.emit()—just like you did for the handleScSignal() callback in Correcting Variable References, on page ?. The routing values are made accessible from the returned anonymous callback function, care of the closure around id.

See? You are indeed a closure champ. You can make quick work of rewriting the other two reusable WebRTC callbacks as closures, too. They're less complicated by comparison.

The ICE-candidate callback needs to reference an id for routing each candidate to the correct peer:

demos/multipeer/js/main.js

```
function handleRtcIceCandidate(id) {
  return function({ candidate }) {
    sc.emit('signal', { recipient: id, sender: $self.id,
      signal: { candidate } });
  };
}
```

We can also write a diagnostic closure on the connection state-change callback. It doesn't make sense to add a class to the body in a multipeer setting (all of the changes from multiple peers' connection states would clobber each other), but what we can do instead is reference an element and add the connection state there, assuming the element exists. Because this is meant to be reusable code, let's opt for a generic peer_element, so that it reads sensibly even if you build WebRTC applications that do not include video elements as part of the interface. The connectionstatechange event will first fire immediately, likely before the peer element has been added to the self side of the call. So we check for the element's existence before doing anything with it:

demos/multipeer/js/main.js

```
function handleRtcConnectionStateChange(id) {
  return function() {
    const peer = $peers.get(id);
    const connection_state = peer.connection.connectionState;
    // Assume *some* element will take a unique peer ID
    const peer_element = document.querySelector(`#peer-${id}`);
    if (peer_element) {
      peer_element.dataset.connectionState = connection_state;
    }
    console.log(`Connection state '${connection_state}' for Peer ID: ${id}`);
  };
}
```

When the element exists, the function preserves the connection state in a data-connection-state attribute, care of the dataset property.[11] The camelCased dataset.connectionState property will automatically convert to a dash-styled data-connection-state attribute in the DOM. The latest state will always replace any older state on the data-connection-state attribute.

And with that, all of your WebRTC callbacks are properly using closures to return id-backed anonymous functions.

11. https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset