Learning WebRTC: The Ultimate Getting Started Guide

A Step-by-Step Guide that Gets You Started with WebRTC

Written By: Joey Lott

Table of Contents

1 Let Me Make Learning WebRTC Easier

I wanted to learn about WebRTC, and boy was it hard!

I don't think the technology is inherently difficult. It *is* a bit convoluted. But once you get your head around it, it's not difficult.

The trouble, of course, was getting my head around it.

The documentation and examples I could lay my hands on made for a steep learning curve. It took a while, but bit by bit, I started to suss out how to do some simple stuff, like build chat apps.

Given how difficult it was for me to piece it all together – and given that I'm no dummy – I figured there must be plenty of other people in the same situation: wanting to get started with WebRTC, but having difficulty getting a solid start.

My goal with this guide is to give you a clear and practical introduction to the basics of WebRTC.

I am most definitely not attempting to create a comprehensive guide on the subject. I am sure that other people are better suited to do that.

I am not the world's greatest programmer. I'm not even in the top 10 percent. But I don't need to be to teach you something useful to you.

So please, don't get distracted by my poor JavaScript or PHP. I'm a JavaScript and PHP hack. I admit that up front. (My professional programming background is in *ActionScript*, which is based on the same specification as JavaScript, so I can figure my way around JavaScript, but if you're a pro, I'm sure you'll shake your head at my code. But don't get distracted. Stay focused on what I can help you with – WebRTC.)

But more than that, I also want to keep things as simple as possible. I want to keep the focus on the basic building blocks, going step by step.

I don't want to distract you from what is most important by trying to wow you with my coding skills and my mastery of design patterns and efficiency.

Heck, I'm not even going to do the most basic of error handling in this guide.

Not because that stuff isn't important for a quality app.

Rather, because I want to keep the focus on what you don't yet know and something very specific – how to create WebRTC applications.

I will leave everything else to others. If you want to learn how to write the greatest JavaScript, there are better resources. If you want to learn how to design apps to handle errors gracefully, there are better resources.

The *only* goal of this book is to provide you with a clear understanding of and practical experience with stuff that is specific to WebRTC apps.

And if you read this and follow along with that in mind, I'm sure you'll agree that this is the best darn guide on the subject.

Let's get started.

2 The 10,000 Foot View of WebRTC: Breaking It Down into Understandable English

You likely already have an understanding of what WebRTC is. At least sort of. But there can be some murky bits in your understanding that can make it way more complicated for you than it needs to be.

So let's start with the basics of what WebRTC is.

There are all kinds of solutions that allow people to communicate in real time over the internet. But none of them meet both of the following two criteria:

1. They are peer-to-peer
2. They are based on standards (so that no plug-ins or third-party apps are necessary)

But WebRTC meets both those criteria.

It's not perfect. (What is?) But it is a step in the right direction. When I write that it's not perfect, I am referring to two main things:

1. WebRTC allows peer-to-peer communication most, but not all of the time. That is a limitation of networks, not WebRTC, per se. But it is a real-world challenge.
2. Not all browsers and devices implement the WebRTC standards, and not all of those that do, do so to the same degree. But, hey, this has always been the challenge with things like browsers, hasn't it?

Most people think of WebRTC as being for video chat apps and customer service widgets on websites. Which it is. But the truth is, WebRTC is an appropriate solution for any real-time communication need in apps. That includes, but is not limited to things such as peer-to-peer file sharing and multi-player games. Really, the sky's the limit. Or, I guess, imagination is the limit.

All of this you may already know. But, if you're like most of us, where it starts to get confusing is as soon as you even begin to think about implementation.

So let's look at the 10,000 foot view in terms of implementation. Because despite the fact that – as I have already written – the implementation can be a little convoluted, once you get your head around what is happening *fundamentally*, you'll find it MUCH less confusing.

There are two categories of functionality that you need to consider when you are working with WebRTC.

1. First, there's the obvious: the actual communication between peers. In other words, how do we use WebRTC to make my web browser communicate directly with your phone so that we can, for example, share photos?
2. Second, there's the *not* so obvious: how do peers find one another to connect in the first place? In other words, how does my web browser *find* your phone?

We need to think about this in the reverse order because we always have to find peers before we can connect them.

The second category – how to find peers – is what, in the world of WebRTC, is called *signaling*

If you search Google for answers about WebRTC signaling, you'll leave more confused than you started. At least that's what happened to me.

But if only somebody will explain it in the right way, honestly, it's not that complicated.

Here's the thing: WebRTC does *not* define how you do signaling. You can do it any which way you want. However you achieve the result, the goal of signaling is to allow peers to find one another and share some information so that they can connect directly.

Let's think about this in terms of some real-world scenarios.

Scenario 1: Imagine that you are a freelance programmer (this may not be hard to imagine if you *are* a freelance programmer). You offer programmer services. But there's a problem: you don't have any clients.

What to do?

Well, you need to find some clients.

How might you do that? Lots of options are available to you. You might…

- Search online job boards
- Call up an existing or past client to see if they need more of your services
- Go to an industry conference or networking event
- Meet up with programmer friends to see if they have any leads

There are lots of different ways you can find clients. But once you find a client, you hopefully have a process that you follow for successfully completing projects. In the WebRTC world, signaling is like searching for clients. Then, WebRTC provides the means to do the communication – which is like having a process for completing projects.

In this scenario, you wouldn't want to be restricted to only one avenue. It would limit your likelihood of achieving success.

In the same way, signaling for WebRTC is wide open so that programmers and companies can use whatever tech is appropriate for them and their context.

The premise of signaling is that you want to provide a forum by which peers can find one another.

[You can use any type of signaling you wish and still make use of the Xirsys offerings such as Xirsys TURN servers (more on TURN later). However, FYI, Xirsys provides a signaling service that uses web sockets. So you can use Xirsys as your full-service WebRTC solution provider as well.]

In the preceding scenario, programmers and clients are looking for one another, but perhaps only in a general way. Meaning, Programmer A is looking for a client. Client X is looking for a programmer. But Programmer A and Client X may not be looking for one another *specifically*.

That is one way signaling may play out. Consider the example of an open chat on a particular theme…say lolcats. (Yes, WebRTC can be used for good as well as evil, and goodness knows the world needs more lolcats chats.)

In an open lolcats chat, people may congregate around a theme – lolcats – and not because they are specifically looking for other users by name.

On the other hand, in the real world we find others not only in a generalized way (such as networking events), but also in specific ways. Which leads us to…

Scenario 2: Imagine that you have made an appointment with a lawyer. Let's say you've invented something amazing, and you want to patent it (you greedy person, you). So you've arranged for an appointment with a patent lawyer who comes highly recommended.

You know that your meeting is at 2 PM on Thursday. But in addition to a time, it also has a *place*. That place in this scenario is the lawyer's office.

On Thursday at 2 PM you walk to her office. Since you'd scheduled the appointment, she is there waiting for you.

You find one another in this scenario first through some kind of referral. Then, specifically, by showing up in a specific place at a specific time.

Signaling for WebRTC could look a lot like that as well…albeit virtually. For example, a tele-medicine WebRTC app could need to connect patients with doctors based on appointments. In other words, specific patients need to connect with specific doctors at specific times.

In that case, the signaling might look like a virtual version of your visit to the lawyer's office. That is, the app would need a server that would host virtual rooms for each doctor. Patients could connect to that room in order to find the doctor and allow the patient and doctor to connect directly. Once the signaling has done as much, the bulk of the communication could happen peer-to-peer rather than needing to use the server. You could think of it like this: once you meet with somebody in their office, you could easily go for a walk or move your meeting to a coffee shop. There's no need to be in the office.

As I've said repeatedly, you can accomplish signaling in lots of different ways. The way that I am most familiar with is to use a socket server. Socket servers allow clients to connect to them and maintain open, bidirectional connections. Because it is the solution I am most familiar with, it will be the one that I'll use in the examples. But at the risk of sounding like a broken record, please remember that you can implement signaling in whatever way makes sense for you.

Although WebRTC does not define how the signaling must be done, it *does* require that certain information is shared during the signaling process.

You can think of it like this: whether you tell your spouse that you love them in person, by phone, or in writing, you do need to communicate that. The means by which you do the communication (the signaling) is open ended. But certain things do need to be communicated for things to work out successfully.

In the case of WebRTC, you can share the information by way of sockets or some other protocol. You can use any data exchange format you wish. But during the process, you need to share some specific

information between the peers interested in communicating directly. I call this *handshaking*, and we'll look at it in more detail later on.

Once peers have successfully shaken hands, they are connected. At that point, they can communicate directly with one another (there is an important caveat here, but we'll get to that later – along with the solution). That communication can be in the form of data, media, or both. In other words, peers could share files, they could share camera and audio streams, they could share real-time information for multi-player gaming, etc.

Hopefully I've given you a clear 10,000 foot view. Next we'll start looking at some actual code so you can see real examples.

3 Signaling

Most signaling examples I've found use a Node.js socket server that runs on localhost. I understand why that is. After all, it is way easier to demonstrate without so many dependencies.

But here's the problem: that's not a real-world example. Not that Node.js cannot be used to create a real-world signaling server. Rather, the example used in those cases is not a real-world example.

And that is no bueno. Because we want to build real-world apps. That's the whole point. We're not interested in building sample apps that we can only ever show off to friends. I mean, you're not going to have many friends for long if every time you have a party you are like, "Hey, gather round my laptop. I'm going to open two browser windows and show you my text chat app that only runs on my machine."

Because that's not real-world.

I'm not saying that dumbed-down example apps aren't useful and don't have their place. They are and they do. I'll show you some dumbed-down examples in this guide too.

But when it comes to signaling, we need to see it working in a real-world example – something that we could actually imagine using in an app we'd build for a client.

To that end, in this guide I'm going to show you examples that use the Xirsys signaling system. It (the Xirsys signaling system) is free to use. So if you haven't already signed up for a free Xirsys account, do so now at http://xirsys.com. That way you can follow along.

Even if you are going to ultimately use your own signaling solution, following along with these examples will still help you see one way to handle signaling in the real-world.

Sidebar: Creating a Channel for your Xirsys Account

You'll need to create what Xirsys calls a channel in order to make use of Xirsys. A channel is simply a unique namespace associated with your Xirsys account. That allows you to create as many different namespaces – or *rooms*, to extend our previous metaphor – for your apps.

As an example, if you create a tele-medicine app that allows doctors and patients to chat with one another, you would want to create a channel for the app – something like awesomeTelemedApp. And you can even create sub-channels within a channel. For example, you could create sub-channels (akin to rooms) for each doctor. So you could have a sub-channel called drJeckle within the awesomeTelemedApp channel.

You can create channels and sub-channels programmatically using the Xirsys RESTful APi. But for getting started, the easiest thing to do is create your first channel through the Xirsys dashboard.

Simply sign in to your Xirsys account, and click the "Services" menu option from the menu on the left. If you have not yet created a channel, you'll be presented with a screen prompting you to enter a channel name and click the button to create the channel.

Create a channel called *sampleAppChannel* and click the create button.

Once you've created the channel, you'll see a screen with information about that channel, including some information to the right that you'll need in order to make RESTful calls. Particularly, you'll need the *ident* and *secret*. You'll come back and copy these later on.

3.1 Opening a Connection to the Signaling Server

Whatever signaling server you use, you're going to have to connect to it. And if it is a socket server – as is the case with the Xirsys signaling system - you'll need to open a persistent connection to it.

Opening a connection to a Xirsys signaling server requires three steps:

1. Getting a temporary token
2. Requesting the best host server
3. Making the connection

Let's look at each of these steps in the following sections.

3.1.1 Getting a Temporary Token

Xirsys signaling requires that you pass a temporary token to the server when opening a connection. And tokens can only be gotten through the Xirsys RESTful API by making a call to the _token end point along with your channel credentials.

To make a request for a token, you will call to the _token endpoint, including your channel name as part of the URL and passing it a variable called *k*. The *k* variable is the username of the user connecting through the app. Each instance of the app will connect as a different user. For example, if you have a multi-person text chat app, each instance of the app will be a different user joining the chat. Or if you have a customer service chat widget on a website, you will have two users – the customer and the customer service representative.

The username can be defined programmatically, of course. Let's say a user signs in to a tele-medicine app to meet with their doctor at the scheduled time. Their account information includes the name, so the app can use that as the username (encoded for use in a URL, that is).

Or the username could be defined by the user at the time of making the request. For example, an open group lolcat chat might simply prompt the user for their name when they try to join.

Point being, however the username is generated, it should be a unique identifier for the user connecting the app instance.

The RESTful call to get the token should be made using PUT and should send the channel *ident* and *secret* using HTTP basic access authentication. In other words, the request looks something like the following (the italicized parts are variables you'll replace with real values):

https://*ident*:*secret*@ global.xirsys.net/_token/sampleAppChannel?k=*username*

This request will respond with a JSON string containing the token. That string looks something like the following:

```
{"v":"iMqWrjlPNs2zIa8S7EnuV1fQEwH_hJDH960UgbueOlkrfYTohR_KM3WkW-
HE4xoWwIQMSPpKZu7Rl8MyVJGxy9hV3NfqriCH9Ims_7S79aEeg24o08Xdz0VgV31fwW
ZayGMHe1c-
wvyOPBdurdE930nuFdmW3ahmjC7kipqUpengrpYKpaV59vD_xRDQjg","s":"ok"}
```

Of course, to actually use this in an app, you'll want to make this request from a server-side bit of code. Why? Because you need to keep your *ident* and *secret*…well, secret! If you put that information in, say, and HTML file, you'd be exposing it so that unscrupulous coders could use your Xirsys account, running up your bandwidth use, and cause problems for you. So keep that information in server-side code only, okay?

For the purposes of this guide, I'll demonstrate how to achieve that using simple PHP pages, starting with this one, which we'll call gettoken.php.

```
<?php
$curl = curl_init();
curl_setopt_array( $curl, array (
      CURLOPT_URL =>
"https://global.xirsys.net/_token/sampleAppChannel?k=".$_POST["usern
ame"],
      CURLOPT_USERPWD => "ident:secret",
      CURLOPT_HTTPAUTH => CURLAUTH_BASIC,
      CURLOPT_CUSTOMREQUEST => "PUT",
      CURLOPT_RETURNTRANSFER => 1
));
$resp = curl_exec($curl);
print $resp;
curl_close($curl);
?>
```

Just add your *ident* and *secret* values, and you can use this code. A call to this page, POSTing a username will return the token JSON string.

Okay. So now you need to call that from your client-side script. Here's the JavaScript code (using jQuery) that does that and writes the response to the console.

```
$.post("gettoken.php", {username: "example-user"}, r =>
console.log(r));
```

We'll put this all together in a complete HTML page in a moment. But first, let's look at some more pieces of the puzzle. Namely…

3.1.2 Requesting the Best Host Server

Xirsys has lots of servers. Lots of servers mean you can achieve the lowest latency possible by finding the server closest to your user that has the lowest load.

That's really useful. But it also means you have to make another RESTful call. This time to an endpoint called _host.

When you call _host, you'll include your channel as part of the URL and you'll pass two variables using GET. The variables are *k* (the same username you passed when calling _token) and *type*. In this case, *type* should always be set to *signal*.

As with all the RESTful calls, you'll pass your *ident* and *secret* using HTTP basic access authentication.

To make that call for our example, we'll use a PHP page called gethost.php, which looks like this:

```
<?php
$curl = curl_init();
curl_setopt_array( $curl, array (
      CURLOPT_URL =>
"https://global.xirsys.net/_host/sampleAppChannel?type=signal&k=".$_
POST["username"],
      CURLOPT_USERPWD => "ident:secret",
      CURLOPT_HTTPAUTH => CURLAUTH_BASIC,
      CURLOPT_CUSTOMREQUEST => "GET",
      CURLOPT_RETURNTRANSFER => 1
));
$resp = curl_exec($curl);
print $resp;
curl_close($curl);
?>
```

This returns a JSON string that provides the host. It looks something like this:

```
{"v":"wss://u2.xirsys.com:4005/ws","s":"ok"}
```

Now, you need to call that from the client-side code using JavaScript code (jQuery) such as this:

```
$.post("gethost.php", {username: "example-user"}, r =>
console.log(r));
```

We're getting close to the goal of starting our signaling. Now, we'll be…

3.1.3 Making the Connection

JavaScript has a WebSocket type that allows us to easily connect to a socket server. All we need to do is provide the URL as a parameter to the constructor.

```
var ws = new WebSocket(serverUrl);
```

The correct URL for the socket server is:

```
host/v2/token
```

That is, *host* is the value returned by the call to _host and *token* is the value returned by _token.

Once you have a WebSocket object, you need to listen for some events using the *addEventListener* method. You can (and probably should) listen for the *open*, *close*, and *message* events. As per the WebSocket specification, WebSocket objects pass MessageEvent objects to the event callbacks.

When dealing with MessageEvent objects in this context, the only property we are interested in is the *data* property, which the socket server sent. The Xirsys signaling socket server sends messages in JSON string format containing a meta object (*m*) and a payload (*p*).

The meta object tells about the message by providing a message type (*o*), the intended recipient of the message (*t*) in the case of a private message, and the sender of the message (*f*).

Here's an example of a JSON string returned by the Xirsys socket server:

```
{"t":"u","p":"user4906c1b8","m":{"t":null,"o":"peer_removed","f":"/s
ampleAppChannel/user4906c1b8"}}
```

You can see that the meta object tells us that the message type is *peer_removed*, which means a user disconnected from the socket server on the same channel. In this case, the payload is simply the username of the user that disconnected.

The Xirsys socket server also sends out messages of type *peers.* And we'll have it send messages of type *message* in order to send arbitrary data to connected peers. We use the *message* type specifically for the handshaking, and we'll talk about that more in a little while. First, for the sake of completeness, let's look at the *peers* message type.

The *peers* message type is a message that contains an array of other users already connected to the socket on the same channel. That is the first message that the socket server sends to a client when it connects. Here's an example of what that can look like:

```
{"t":"u","p":{"users":["user3a092eeb","user4906c1b8"]},"m":{"t":"
user3a092eeb ","o":"peers","f":"/sampleAppChannel/__sys__"}}
```

As you can see, in this case the payload is an object with a *users* property that contains an array of the users connected to the socket, including the user that just connected from the peer receiving the message. Also note that the sending user in this case is *__sys__*, which is the system user, i.e. the Xirsys socket server.

Okay. So let's now look at a complete, working example that requests the token and host, then connects to the Xirsys signaling server and listens for messages. This code is all in one HTML page and does not do any error handling just to keep things simple. Clearly (hopefully this is clear, at least) in a real application, you'd want to handle errors at every step. For example, you'd want to handle errors returned by the RESTful API. But we're just wanting to see the basic steps to connect to signaling at this point.

```
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
  </head><body>
  <body>

    <script>
      var token;
```

```
      var socket;

      window.onload = () => {
        $.post("http://localhost/gettoken.php", {username: "example-
user"}, r => getHost(r));
      }

      function getHost(r) {
        token = JSON.parse(r).v;
        $.post("http://localhost/gethost.php", {username: "example-
user"}, r => openSocket(r));
      }

      function openSocket(r) {
        var host = JSON.parse(r).v;
        socket = new WebSocket(host + "/v2/" + token);
        socket.addEventListener("message", onSocketMessage);
      }

      function onSocketMessage(evt) {
        console.log(evt);
      }

    </script>

  </body>

</html>
```

If you run this page and open the console in the browser, you'll see messages returned from the socket server. Or, at least, you'll see one – a *peers* message listing (probably) just one user (*example-user*).

That's interesting. But now let's do a little bit more with this. Let's allow multiple users to connect at the same time. To accomplish that, all we need is to prompt the user for a username before connecting. Then, we can open the page in two browser windows and each can connect with a different username.

To provide a visual indicator that the users have connected, let's also add a div element to the page and display the users connected to the signaling server in it.

```
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
  </head><body>
  <body>

    <div>
      username: <input type="text" id="username" />
      <button id="connectButton"
onclick="connect()">connect</button>
```

```
    </div>

    <div id="messages">
    </div>


    <script>
      var token;
      var socket;

      function connect() {
        $.post("http://localhost/gettoken.php", {username:
$('#username')[0].value}, r => getHost(r));
      }

      function getHost(r) {
        token = JSON.parse(r).v;
        $.post("http://localhost/gethost.php", {username: "example-
user"}, r => openSocket(r));
      }

      function openSocket(r) {
        var host = JSON.parse(r).v;
        socket = new WebSocket(host + "/v2/" + token);
        socket.addEventListener("message", onSocketMessage);
      }

      function onSocketMessage(evt) {
        var data = JSON.parse(evt.data);
        var option;
        var messagesElement = $("#messages")[0];
        switch (data.m.o) {
          case "peers":
            var users = data.p.users;
            for(i = 0; i < users.length; i++) {
              messagesElement.innerHTML += users[i] + " is in the
chat<br>";
            }
            break;
          case "peer_connected":
            var f = data.m.f.split("/");
            var joining = f[f.length-1];
            messagesElement.innerHTML += joining + " has joined the
chat<br>";
        }
      }

    </script>

  </body>
```

```
</html>
```

This code is very simple. It doesn't verify the username is valid before connecting. So if you don't want to break things, just enter usernames that contain no spaces or other characters that will cause problems. And don't connect more than once without reloading the page.

But if you do this right – open the page in two or more browser windows, enter a unique username (no spaces), and click the connect button in each – you'll see that each instance updates to display all the connected users.

Now that we've seen how to do that much, we're ready for the convoluted bit…

3.2 Handshaking

What I call the handshaking process is, in my opinion, one of the most challenging pieces for most people to wrap their heads around.

However…my hope is that right here in this guide we can change that. It is convoluted, but it's not actually that hard to understand. I'm going to break it down, step by step, and by the time you've gone through the subsections of this handshaking part of the guide, you should be thinking, "Why would anybody have trouble with that? It's so easy!"

The handshaking process does have lots of bits and pieces. Normally it is best to try to give the big picture overview before diving into the details of the bits and pieces. But I think that in this case the overview is what confuses things. It actually is much easier to look at this one piece at a time without trying to understand the big picture. Once we've done that, you'll naturally understand the big picture. Or, at the very least, it will be pretty easy to stitch it all together.

So let's get started with…

3.2.1 ICE

Oh, I know what you're thinking. "Stop, collaborate, and listen…" But we're not talking about that kind of ICE.

It's ICE, which stands for Interactive Connectivity Establishment. And it basically, what ICE does is provides a collection of ways by which peers can attempt to connect to one another.

Think of it like this: you're at a networking event. You meet somebody you'd like to remain in contact with. What do you do? Well, you probably exchange contact information. And often, that contact information (in the form of, say, a business card) provides multiple ways to get in contact such as email, office phone, cell phone, mailing address, twitter handle, etc.

Maybe for you, the ideal form of communication is in person. But if that isn't possible, perhaps you fall back on phone, which is your second preferred method of communication. If that's not possible, you'll use email. And if email doesn't work, you'll reach out through Twitter. Etc., etc.

ICE does something similar.

In an ideal scenario, when two peers connect through signaling, they'd be able to exchange their IP addresses and connect directly. However, most devices don't know their IP address. At least not one

that that another peer can use to connect to them. That's because most devices are behind NATs, which allow multiple devices to connect using a single public IP address.

Therefore, what is needed is yet another server that can figure out the public IP addresses and routing to the two peers and send that information to the two peers so they can connect directly.

That server is called a STUN server. As in "set phasers to stun?" No, as in Session Traversal Utilities for NAT.

I'm sure it's way more complicated, but as far as you and I are concerned, STUN servers basically just tell the two peers how to connect directly, and after that, the STUN server is not needed. So that's good for bandwidth concerns because your app doesn't have to use much server bandwidth.

In many WebRTC scenarios, STUN is the preferred method for connecting peers because of the server bandwidth concern.

However, in reality, that won't always work. Why? Because sometimes a peer or peers are behind firewalls or some kind of obstruction that prevents them from connecting directly with any other peers.

So then what? Well, the peers can each – on their own – connect with a server out there in the great internet ethers. So that server can act as a relay between the two peers. In other words, rather than peer A connecting directly to peer B, A connects to server C and B connects to server C, and C relays communication between A and B. That server is called a TURN server.

TURN? As in "turn and face the strange, ch-ch-changes?" No. As in Traversal Using Relay NAT.

TURN servers mean that communication is much more bandwidth-expensive. That's because rather than A sending data to B, A has to send data to C (the TURN server) and C relays it to B. So each communication costs twice as much. And somebody's gonna pay for that server bandwidth, believe you me.

Of course, TURN has some added benefits, as we'll see later. But, I digress…

As far as ICE is concerned, TURN is a lower priority compared to STUN. Meaning, the app will first try STUN servers but fall back to TURN servers if needed.

There's more. Because in addition to STUN and TURN, there's also various protocols such as TCP/IP and UDP to try. And there may be multiple servers to try in each case because, hey, sometimes servers are down or maybe some peers are blocked from some servers but not others, etc.

So ICE is just a big bundle of different *candidates* to try.

Okay, great. ICE sounds pretty helpful and all, but…uh…it sounds *complicated*. Where do we start?

First, we want to get all the ICE candidates for an app instance as simply as possible. Typically, in a real-world scenario, that means making a call to a server for that information. And in the case of Xirsys, we make a call to the RESTful API endpoint called _turn.

As with the other API calls, we'll be including a channel name in the URL and we'll be providing the *ident* and *secret* as HTTP basic access authentication. To request the ICE candidates, call this end point using PUT.

Here's the PHP code I use. For the examples in this guide, the code goes in a page called getice.php.

```
<?php
$curl = curl_init();
curl_setopt_array( $curl, array (
      CURLOPT_URL =>
"https://global.xirsys.net/_turn/sampleAppChannel",
      CURLOPT_USERPWD => "ident:secret",
      CURLOPT_HTTPAUTH => CURLAUTH_BASIC,
      CURLOPT_CUSTOMREQUEST => "PUT",
      CURLOPT_RETURNTRANSFER => 1
));
$resp = curl_exec($curl);
print $resp;
curl_close($curl);
?>
```

In order to test this out, let's create a simple HTML page that makes a request to the PHP page and outputs the result to the console.

```
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
  </head><body>
  <body>

    <script>

      window.onload = () => {
        $.post("http://localhost/getice.php", null, r =>
console.log(r));
      }

    </script>

  </body>

</html>
```

When I view this in the browser, here's what I see in the console:

```
{"v":{"iceServers":[{"url":"stun:u1.xirsys.com"},{"username":"dec22e
00-a1ff-11e7-9bfc-
ce8483d9a683","url":"turn:u1.xirsys.com:80?transport=udp","credentia
l":"dec22e78-a1ff-11e7-98bd-5f53926dfc20"},{"username":"dec22e00-
a1ff-11e7-9bfc-
ce8483d9a683","url":"turn:u1.xirsys.com:3478?transport=udp","credent
ial":"dec22e78-a1ff-11e7-98bd-5f53926dfc20"},{"username":"dec22e00-
a1ff-11e7-9bfc-
ce8483d9a683","url":"turn:u1.xirsys.com:80?transport=tcp","credentia
```

```
l":"dec22e78-a1ff-11e7-98bd-5f53926dfc20"},{"username":"dec22e00-
a1ff-11e7-9bfc-
ce8483d9a683","url":"turn:u1.xirsys.com:3478?transport=tcp","credent
ial":"dec22e78-a1ff-11e7-98bd-5f53926dfc20"},{"username":"dec22e00-
a1ff-11e7-9bfc-
ce8483d9a683","url":"turns:u1.xirsys.com:443?transport=tcp","credent
ial":"dec22e78-a1ff-11e7-98bd-5f53926dfc20"},{"username":"dec22e00-
a1ff-11e7-9bfc-
ce8483d9a683","url":"turns:u1.xirsys.com:5349?transport=tcp","creden
tial":"dec22e78-a1ff-11e7-98bd-5f53926dfc20"}]},"s":"ok"}
```


As you can see, _turn returns a JSON string with a v property that contains the ICE information that I want to use in my app.

Let's look at that information in a little more detail.

Basically, it is a collection of servers that the app should try to use to connect peers. It says that first, the app should try to use a stun server at u1.xirsys.com. If that doesn't work, it states that next the app should try using a TURN server at u1.xirsys.com on port 80 using UDP. It also provides credentials for connecting to the server.

Should that fail, it provides further fall back options. You can see that it prefers to use STUN. If that doesn't work, it prefers to use TURN using UDP. And if that doesn't work, it falls back on TURN using TCP at various ports.

Okay, so now we know (kind of) what ICE is (hint: as far as we are concerned, it's just that hunk of data that we got back from the call to _turn). And we know how to get ICE candidates from the Xirsys API…

But now what on earth do we do with that?

The good news is that (for now) the only thing we have to do with it is parse it using JSON.parse() and pass the object with the ICE candidates to the constructor of a new RTCPeerConnection. The message returned by the Xirsys API call parses into an object with a *v* property that is what we want. So, assuming that *result* is the return string value from the Xirsys API call, the code would look something like this:

```
var iceCandidates = JSON.parse(result).v;
var pc = new RTCPeerConnection(iceCandidates);
```

That's it. Now we have a RTCPeerConnection object that knows its ICE candidates. Which means it is ready to start attempting to communicate with other peers. The first thing it needs to do is *make an offer*.

3.2.2 Making an Offer for Communication

Next, we need to be able to make an offer to peers that an app instance has found through signaling. One way to think of this is like dialing the phone. It is similar in that the caller must dial the phone and the recipient must answer before communication by phone can happen.

But that metaphor falls apart a bit in this case. While there are some similarities, what RTCPeerConnection communication is more like is this: two people meet at a networking event. Let's call them Phil and Bill. Phil says to Bill, "Hey, Bill, it's been really nice getting to meet you here at the lolcats convention. I'd like to stay in touch. Can we do that?"

And Bill, since he also enjoyed meeting Phil, says, “Phil, that would be great.”

Phil made an offer to Bill. He said, that he’d like to stay in touch.

Bill gave an answer. He responded affirmatively.

Once Phil and Bill make this small exchange, next they need to exchange information so that they can make that connection. Phil gives Bill his business card. Bill gives Phil his business card. Both business cards have multiple means of reaching one another – phone, fax, email, Twitter, Facebook, website, mailing address, etc.

When it comes to WebRTC, RTCPeerConnection objects do a similar dance. Their initial communication is facilitated by a third party – the signaling channel. One makes an offer for direct communication. The other accepts (or doesn’t). Then they exchange contact information so that they can begin to communicate directly rather than through the signaling channel.

Now, when Bill and Phil are at the lolcats convention (man, that sounds like *fun*), they are able to identify one another by sight. Bill recognizes Phil. Phil recognizes Bill. But in the world of WebRTC, the RTCPeerConnection objects don’t have such luxuries. So they need a way to identify themselves.

They do that by way of creating a session description of themselves and the communication session they intend to start. The initiator needs to generate a description of itself, which it does by calling the createOffer() method.

The createOffer() method generates a description in SDP format and inserts it into an RTCSessionDescription object with a type of offer. SDP stands for session description protocol. And for our purposes, you really don’t have to give a hoot about the workings of SDP. All you have to know and care about is that when you call createOffer() on a RTCPeerConnection object, it generates a description that happens to be in SDP format and inserts that into an *RTCSessionDescription* object with the type set to offer.

However, it does that asynchronously. So rather than returning the SDP value immediately, createOffer() returns a *Promise*. If you’re not yet familiar with the *Promise* interface in ES6, I recommend that you familiarize yourself with it. But basically, in this case all we need to do is call then() on the returned *Promise* object and pass it the callback function.

When that callback function is called, it will be passed that SDP description. Here’s an example that calls createOffer() and outputs the description to the console when it gets returned.

```
pc.createOffer().then(d => console.log(d));
```

If you test that out (I encourage you to do so), you’ll see that it outputs an *RTCSessionDescription* object that looks something like the following:

{type: "offer", sdp: "v=0
↵o=- 992697269155962202 2 IN IP4 127.0.0.1
↵s=…id:data
↵a=sctpmap:5000 webrtc-datachannel 1024
↵"}

Like I've already said, in most cases, you simply don't have to care at all about the details of SDP. But one thing you do want to notice about this value is that it has a type property set to offer. You'll be making use of that later to differentiate between messages that come across the signaling channel.

Of course, in the real world, we'll want to do more than output the value to the console. Specifically, there are two things we'll do in nearly every case.

1. First, we'll assign the description to the RTCPeerConnection object as the local description using *setLocalDescription()*. That way, future communications from the object to a peer will contain that description to identify the sender.
2. Second, we'll want to send a message using the signaling channel. The message should specify that an offer is being made, and it should contain the SDP description so that the recipient will be able to correctly make an answer.

Let's first look at setting the local description. Here's a very simple example that does just that.

```
pc.createOffer().then(d => onCreateOffer(d));

function onCreateOffer(d) {
  pc.setLocalDescription(d);
}
```

Next, let's look at an example of sending that description to a recipient using the signaling channel. Of course, exactly how you will achieve that depends on the signaling mechanism you are using. Here, I'll demonstrate how to do that using the Xirsys signaling server.

The Xirsys signaling server expects JSON strings that contain *t*, *m*, and *p* properties. In this case, *t* should always be set to the string *u*, which means the type is user message service. The *m* property is the meta object that tells about the message – the type, the recipient (or leave undefined if sent to all who are connected to the signaling server), and the sender of the message. And the *p* property is the payload – the message to be sent. In this case, the payload should be an object with a *msg* property, and the session description should be assigned to that.

Okay, so here's how we could construct that object, where *d* is the description returned by *createOffer()* and *username* is the local app instance username.

```
var pkt = {
    t: "u",
    m: {
          f: "sampleAppChannel/" + username,
          o: "message"
    },
    p: {msg:d}
};
```

Then, using the WebSocket object connected to the signaling server, simply call *send()* and pass it a stringified version of the object.

```
ws.send(JSON.stringify(pkt));
```

That sends the message to the server, which is then relayed to all the connected peers. If a connected peer is listening for messages from the socket server (which it should be), what it needs to then do is inspect the message and take action.

Of course, a peer could reject an offer. But let's assume for now that the peer will always accept offers. How do we code the acceptance of an offer? Let's take a look.

3.2.3 Answering the Offer

The peer doing the accepting of the offer needs to do four things:

1. It needs to (if it hasn't already), create an RTCPeerConnection object.
2. It needs to set the remote description of that object to the description included in the offer it just received
3. It needs to create an answer
4. It needs to send that answer

Most of this is going to be familiar to you now that you've seen how to create an offer. And what is new won't be surprising. Let's look at the code.

You'll recall from earlier that we set up a WebSocket object and listened for messages. The handler function for messages used a switch statement to handle different types of messages. Namely, peers and peer_connected.

Now, we'll need to add to that function and have it also handle the case in which the message type is *message*. And in that case, we'll want to handle the offer if the incoming message is an offer. We'll know it is because the payload will contain a msg property that itself is an object with a type property set to offer (remember, I told you that would be important later).

So we could add another case to the switch statement from our earlier function to handle messages of type message, and then a nested switch statement that looks at the cases for the value of the payload's msg.type property. And in the case of an offer message, we can cast the payload's msg object to an *RTCSessionDescription* object.

```
case "message":
  switch(data.p.msg.type) {
    case "offer":
      var desc = new RTCSessionDescription(data.p.msg);
      break;
  }
```

Once we've got that *remote* description (the description of the calling peer), we'll want to call *setRemoteDescription()* on the receiving RTCPeerConnection object and pass it the description.

```
case "message":
  switch(data.p.msg.type) {
    case "offer":
      var desc = new RTCSessionDescription(data.p.msg);
              pc.setRemoteDescription(desc);
      break;
  }
```

That way the RTCPeerConnection object will know from here on out what peer it is communicating with.

And next, it needs to create an answer, which generates a description of itself to be returned (via the signaling channel) to the caller. To do that, call *createAnswer()* on the *RTCPeerConnection* object. And as with *createOffer()*, this method generates an RTCSessionDescription object, but it does so ansychronously. So instead of returning the description immediately, it returns a *Promise*. This code demonstrates how you could generate the answer and output the description to the console.

```
pc.createAnswer().then(d => console.log(d));
```

If you test this (and I suggest you do), you'll see that the *RTCSessionDescription* object in this case has a type of *answer*. Again, that is going to be important.

Of course, in a real-world example, you'll want to do something more than output the description to the console. You'll want to return that answer to the caller by way of the signaling channel.

As always, how you do that depends on how you are handling signaling. I'll show you an example here using the Xirsys signaling server and an existing WebSocket object called *socket*.

I'd simply add the createAnswer() call to the switch statement from earlier:

```
case "message":
  switch(data.p.msg.type) {
    case "offer":
      var desc = new RTCSessionDescription(data.p.msg);
      pc.setRemoteDescrition(desc);
      pc.createAnswer().then(d => onCreateAnswer(d));
      break;
  }
```

And I'd define onCreateAnswer() thusly:

```
function onCreateAnswer(d) {
  pc.setLocalDescription(d);
  var pkt = {t: "u", m: {f: "sampleAppChannel/" + localUsername, o:
"message", t: null}, p: {msg:d}};
  socket.send(JSON.stringify(pkt));
}
```

3.2.4 Handling the Answer

Handling the answer is pretty darn simple compared to everything else. All you have to do is handle the case in which the message payload's message type is *answer*. Then, as in the previous step, cast the payload's msg object to *RTCSessionDescription*. Then, as when handling the offer, set the remote description to the description just received.

```
case "message":
  switch(data.p.msg.type) {
    case "offer":
      var desc = new RTCSessionDescription(data.p.msg);
      pc.setRemoteDescrition(desc);
      pc.createAnswer().then(d => onCreateAnswer(d));
      break;
    case "answer":
```

```
            var desc = new RTCSessionDescription(data.p.msg);
            pc.setRemoteDescrition(desc);
            break;
    }
```

And that is all there is to do in handling the answer. The only purpose here is to set the remote description.

To recap, up to this point in the handshaking process what we've done is this:

1. Make an offer, which consists of:
    a. creating a session description using createOffer()
    b. setting that as the local description
    c. sending an offer message including the description over signaling.
2. Receive the offer, which consists of:
    a. setting the incoming description as the remote description
    b. creating a session description using createAnswer()
    c. setting the local description to the value returned by createAnswer()
    d. sending the answer message including the description over signaling
3. Receive the answer, which consists of:
    a. Setting the incoming description as the remote description

This whole business can get confusing *if* you don't keep a few important things in mind. But if you keep these few things in mind, you'll find that none of this is confusing. It's actually quite simple.

First, keep in mind that peer connections are always between two peers. As we'll see later, if you want to create apps that allow more than two peers to communicate, you can do that, but it requires peer connections between all communicating pairs. In other words, if you have users A, B, and C, A and B will need a peer connection, A and C will need a peer connection, and B and C will need a peer connection.

Second, keep in mind that peer connections always involve a caller/offerer and answerer. In other words, in the case of A and B, A can be the caller and B can be the answerer or B can be the caller and A can be the answerer, but there are no other possibilities. This is exactly like is the case with telephony in that if you and I want to talk by phone, one of us has to call and one has to answer. It doesn't work if we both try to initiate the call.

Alright, we've done a lot of the handshaking, but…*but*…I can almost hear you wanting to know, "Is that really all there is to it? What about all those ICE candidates? How do those get exchanged? Isn't that something we have to deal with?"

And you'd be right. So let's take a look.

3.2.5 Handling ICE Candidates

This far, we've retrieved ICE candidates, assigned them to the RTCPeerConnection objects, and had the peers do a little dance of offer and answer, exchanging some session descriptions.

But what has *not* happened yet is the peers have not exchanged ICE candidates. Meaning, they are not yet able to communicate directly without relying on the signaling channel.

How do we do that? Well, once the conditions are right, a *RTCPeerConnection* object will start practically shouting out its ICE candidates, one at a time. What are the conditions?

First, obviously, it must have some ICE candidates. But we've taken care of that by passing the ICE candidates to the constructor. So we can check that off as done.

Next, it must have both a local and a remote description. Again, we can check that off as done.

But there's one more thing that must be true for an RTCPeerConnection object to start "shouting" out its ICE candidates. It must have some kind of data or media sharing mechanism defined.

Now, as of this writing, WebRTC defines two means of sharing: data channels and streams/tracks. In other words, peers can share data with one another or they can share audio/video.

The simplest of these for us to set up is a data channel. So for now, solely so that we can complete the handshaking process, we're going to define a data channel. Later, we'll focus on working with data channels. But for now, we're only concerned with completing the handshaking process, which requires some form of sharing set up. And for our purposes now, we'll use a data channel to achieve that.

So…to do that, simply call the createDataChannel() method on the *RTCPeerConnection* object, and pass it the name (and arbitrary name that you get to choose) of the data channel. In this example we'll use *data* as the name of the channel.

```
var dataChannel = pc.createDataChannel("data");
```

Just add that bit of code immediately after you've constructed the peer connection, et voila! The peer connection is now shouting out its ICE candidates! In other words:

```
var pc = new RTCPeerConnection(iceServers);
var dataChannel = pc.createDataChannel("data");
```

(Note: We'll cover this in more depth in just a moment, but for the sake of clarity, know that you only need to create the data channel in one peer. The other will receive it once the peers are connected. We'll see how to do that in section 4. But for now, we'll just create the data channel in both the peers since our only goal at this point is to get the two communicating.)

Oh, wait. What's that? You don't hear anything?

Right. That's because you're not listening. Let's add an onicecandidate handler to the peer connection.

```
pc.onicecandidate = candidate => console.log(candidate);
```

Assuming you've got your code set up correctly to handle the handshaking up to this point, what you'll see now is ICE candidates being written to the console.

If only that was all that was necessary. But, alas, we now have to send those candidates over signaling to the other peer. And we've got to handle them in the peer.

To send the candidates, all we have to do is send a message containing the candidate data over the signaling channel. As usual, you have a lot of leeway as to exactly how to do that. Here's one way to do it using the Xirsys signaling server. In this code, assume that *candidate* is an *RTCIceCandidate* object. You don't have to know much about that type most of the time. But for the purposes of passing the correct

information across the signaling channel, you need to know that it has sdpMLineIndex, sdpMid, and candidate properties. In this case, we're simply creating an object that copies those values along with a type property set to candidate. We'll send that in the payload so that we can handle it in the peer. (The following code is just a snippet. To see it in context, look at the complete example that follows in the next section.)

```
var cPkt = {type: "candidate",
          sdpMLineIndex: candidate.sdpMLineIndex,
          sdpMid: candidate.sdpMid,
          candidate: candidate.candidate
          };
var pkt = {t: "u",
      m: {f: "SampleAppChannel/" + localUsername,
           o: 'message'},
      p: {msg:cPkt}
      };
socket.send(JSON.stringify(pkt));
```

Now, in the peer, we'll add to the switch statement from earlier. This time, handling messages of type candidate. In that case, we'll cast the payload's msg value to RTCIceCandidate and pass that to the *addIceCandidate()* method of the peer connection.

```
case "candidate":
   var candidate = new RTCIceCandidate(data.p.msg);
  pc.addIceCandidate(candidate);
```

The result here is that the peers send their candidates, one at a time, across the signaling channel. The receiving peer adds the candidates to its *RTCPeerConnection* object. And then, through the inner workings of the browser or OS or whatever implements WebRTC, the peers know how to deal with these candidates and attempt to contact one another directly.

Okay. Let's see how all of that works in the real world.

3.2.6 Handshaking by Example

In this simple example, we'll add to the previous example by handling all the handshaking and outputting some progress to the user.

First, let's just look at the whole shebang: I'll bold the new bits.

```
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
  </head><body>
  <body>

    <div>
      username: <input type="text" id="username" />
      <button id="connectButton"
onclick="connect()">connect</button>
```

```
    </div>

    <div id="messages">
    </div>

    <script>
      var token;
      var socket;
      var ice;
      var pc;
      var localUsername;
      var dataChannel;

      function displayMessage(message) {
        $('#messages')[0].innerHTML += message + "<br>";
      }

      window.onload = () => {
        $.post("http://localhost/getice.php", null, r => onIce(r));
      }

      function onIce(r) {
        ice = JSON.parse(r).v;
        pc = new RTCPeerConnection(ice);
        displayMessage("peer connection state:" +
pc.connectionState);
        pc.onicecandidate = evt => onIceCandidate(evt);
        pc.onconnectionstatechange = evt => displayMessage("peer
connection state:" + pc.connectionState);
        pc.oniceconnectionstatechange = evt =>
displayMessage("trying to connect to peer:" +
pc.iceConnectionState);
      }

      function connect() {
        localUsername = $('#username')[0].value;
        $.post("http://localhost/gettoken.php", {username:
localUsername}, r => getHost(r));
      }

      function getHost(r) {
        token = JSON.parse(r).v;
        $.post("http://localhost/gethost.php", {username:
localUsername}, r => openSocket(r));
      }

      function openSocket(r) {
        var host = JSON.parse(r).v;
        socket = new WebSocket(host + "/v2/" + token);
        socket.addEventListener("message", onSocketMessage);
      }
```

```
        function onSocketMessage(evt) {
          var data = JSON.parse(evt.data);
          var option;
          var selectElement = $("#usersSelect")[0];
          switch (data.m.o) {
            case "peers":
              var users = data.p.users;
              for(i = 0; i < users.length; i++) {
                displayMessage("user in chat:" + users[i]);
              }
              break;
            case "peer_connected":
              var f = data.m.f.split("/");
              var joining = f[f.length-1];
              displayMessage("new user joined:" + joining)
              callPeer(joining);
              break;
            case "message":
              switch(data.p.msg.type) {
                case "offer":
                  var desc = new RTCSessionDescription(data.p.msg);
                  var f = data.m.f.split("/");
                  var sender = f[f.length-1];
                  pc.setRemoteDescription(desc);
                  pc.createAnswer().then(d => onCreateAnswer(d,
sender));
                  break;
                case "answer":
                  var desc = new RTCSessionDescription(data.p.msg);
                  var f = data.m.f.split("/");
                  var sender = f[f.length-1];
                  pc.setRemoteDescription(desc);
                  break;
                case "candidate":
                  displayMessage("you have received a candidate");
                  var candidate = new RTCIceCandidate(data.p.msg);
                  pc.addIceCandidate(candidate);
              }
          }
        }

        function callPeer(peer) {
          displayMessage("you are the caller...calling " + peer);
          dataChannel = pc.createDataChannel("data");
          pc.createOffer().then(d => onCreateOffer(d, peer));
        }

        function onCreateOffer(d, peer) {
          pc.setLocalDescription(d);
```

```
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }

      function onCreateAnswer(d, peer) {
        displayMessage("you are the answerer...you are answering " +
peer);
        pc.setLocalDescription(d);
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }

      function onIceCandidate(evt) {
        displayMessage("you are sending a candidate");
        var candidate = evt.candidate;
        if (evt.candidate != null) {
                    var cPkt = {type: "candidate",
                          sdpMLineIndex: candidate.sdpMLineIndex,
                          sdpMid: candidate.sdpMid,
                          candidate: candidate.candidate
                      };
                    var pkt = {
                      t: "u",
                      m: {
                          f: "SampleAppChannel/" + localUsername,
                          o: 'message'
                      },
                      p: {msg:cPkt}
                  }
                    socket.send(JSON.stringify(pkt));
        }
      }

    </script>
  </body>
</html>
```

Okay. That's quite a lot. So let's look at some of the parts that might require a little bit of explanation.

First, the onIce() function, which gets called as soon as the $.post() success event gets fired. (Note that on window load I've got the app calling the getice.php page.)

This function parses the JSON string returned by PHP. The v property of that object contains the ICE candidates in the format that we can pass to the RTCPeerConnection constructor, which we do immediately. Then, we add some event handlers to the peer connection.

```
      function onIce(r) {
        ice = JSON.parse(r).v;
        pc = new RTCPeerConnection(ice);
```

```
        displayMessage("peer connection state:" +
pc.connectionState);
        pc.onicecandidate = evt => onIceCandidate(evt);
        pc.onconnectionstatechange = evt => displayMessage("peer
connection state:" + pc.connectionState);
        pc.oniceconnectionstatechange = evt =>
displayMessage("trying to connect to peer:" +
pc.iceConnectionState);
      }
```

In the onSocketMessage() function, we attempt to calls peers as they join (in the "peer_connected" case).

```
            case "peer_connected":
              var f = data.m.f.split("/");
              var joining = f[f.length-1];
              displayMessage("new user joined:" + joining)
              callPeer(joining);
              break;
```

And we've added a new case to handle "message" types. These types can have three subtypes, which we also handle with a nested switch statement.

```
            case "message":
              switch(data.p.msg.type) {
First, there's the case of an offer. When that occurs, we cast the
description to a RTCSessionDescription object and set the remote
description on the peer connection. Then we create answer.
                case "offer":
                  var desc = new RTCSessionDescription(data.p.msg);
                  var f = data.m.f.split("/");
                  var sender = f[f.length-1];
                  pc.setRemoteDescription(desc);
                  pc.createAnswer().then(d => onCreateAnswer(d,
sender));
                  break;
```

If the message is an answer, we do much the same as when it is an offer. The difference here is just that one peer will handle an offer and another will handle the answer – but neither will handle both.

```
                case "answer":
                  var desc = new RTCSessionDescription(data.p.msg);
                  var f = data.m.f.split("/");
                  var sender = f[f.length-1];
                  pc.setRemoteDescription(desc);
                  break;
```

Then there's the case of a candidate. When that happens we cast the message to a RTCIceCandidate object and add that to the peer connection.

```
                case "candidate":
                  displayMessage("you have received a candidate");
                  var candidate = new RTCIceCandidate(data.p.msg);
                  pc.addIceCandidate(candidate);
```

```
        }
```

The callPeer() function creates the data channel. Only the caller will create a data channel. The answerer will receive it after negotiation. After creating the data channel, the function creates the offer.

```
function callPeer(peer) {
  displayMessage("you are the caller...calling " + peer);
  dataChannel = pc.createDataChannel("data");
  pc.createOffer().then(d => onCreateOffer(d, peer));
}
```

Once the offer is created, we handle it with onCreateOffer(). This function sets the local description and sends a message with the description to the peer over the signaling channel.

```
function onCreateOffer(d, peer) {
  pc.setLocalDescription(d);
  var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
  socket.send(JSON.stringify(pkt));
}
```

The onCreateAnswer() function is basically identical to onCreateOffer(). In fact, the same function could be used in either case. But for simplicity and clarity while learning, we've defined two functions.

```
function onCreateAnswer(d, peer) {
  displayMessage("you are the answerer...you are answering " +
peer);
  pc.setLocalDescription(d);
  var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
  socket.send(JSON.stringify(pkt));
}
```

Lastly, we handle ICE candidates with onIceCandidate(). This function just creates the message with the candidate data in it and sends it over signaling.

```
function onIceCandidate(evt) {
  displayMessage("you are sending a candidate");
  var candidate = evt.candidate;
  if (evt.candidate != null) {
              var cPkt = {type: "candidate",
                    sdpMLineIndex: candidate.sdpMLineIndex,
                    sdpMid: candidate.sdpMid,
                    candidate: candidate.candidate
                };
              var pkt = {
                t: "u",
                m: {
                    f: "SampleAppChannel/" + localUsername,
                    o: 'message'
                },
                p: {msg:cPkt}
            }
              socket.send(JSON.stringify(pkt));
```

```
        }
    }
```

Now that we've gotten all the heavy lifting out of the way, what follows is going to be a breeze.

4 Data Communication

All that has come before in this guide may seem like a whole lot for very little. After all, we haven't accomplished much that seems very useful in and of itself.

But…*BUT*…now that we've set the stage, we've enabled two peers to communicate directly. (And a little later we'll see how to use this to allow more than two peers to communicate directly…albeit always in pairs.) And that's where things get interesting. Or, as a friend of mine used to say, "Now we're cooking with gas!"

As I've stated before, WebRTC provides two ways for peers to share information directly with one another: media (video and audio) and generic data.

In terms of implementation, sharing data is the easier of the two. And, I'll argue, the more exciting because it opens up a wide world of possibilities. After all, with data sharing, we can do so much more than a simple text chat (which is what we'll be looking at in the examples). We can do screen sharing, multiplayer games, file sharing, and…well, pretty much anything.

Okay. So as we saw in the previous sections, you need to have some data sharing set up on the RTCPeerConnection object before it will broadcast its ICE candidates and complete the handshaking process. Therefore, I showed you how to create a data channel, which we do through the *createDataChannel()* method of the RTCPeerConnection object.

But we didn't get into much more detail or see how to use that data channel. Let's do that now.

4.1 What Is a Data Channel?

A peer connection can have up to 65,535 data channels, which are, as the name suggests, channels between which two connected *RTCPeerConnection* objects can share data.

The *createDataChannel()* method returns a data channel object of type *RTCDataChannel*. The *RTCDataChannel* interface is pretty simple. It doesn't need to do much.

You can send data using the *send()* method. And you can listen for data channel events such as events for when the channel opens (onopen) or closes (onclose) or when a message comes across the channel (onmessage).

That's the essence of what it does.

There are a few adjustments you can make to a data channel when you create it. Depending on the needs of your data channel, you can, for example, force messages to arrive in the same order they are sent or allow them to arrive in whatever order they do. But for our purposes, we're not going to concern ourselves with any of that. We'll be using the defaults. If and when you have apps that have needs not met by the defaults, take a look at the API documentation for createDataChannel(), and you'll find what you need there.

So for this guide, all we need to do to create a new data channel is call createDataChannel() on one of the *RTCPeerConnection* objects. At that point, here's what happens:

1. If the peer connection hasn't yet finalized the handshaking by exchanging ICE candidates (a process known as negotiation), it will do that.
2. The object for which the data channel gets created will then send that data channel to the peer it is connected to.
3. The peer's *RTCPeerConnection* object will dispatch an event handled by the function assigned to ondatachannel. It will pass that function an event that contains a reference to the data channel.
4. The peer should then maintain a reference to that data channel and listen for events dispatched by it such as the open, close, and message events.

In other words, only one of the two peers in a connection should call createDataChannel() to create a single channel to communicate with the other peer. It will automatically be sent to the connected peer. That peer simply needs to handle the event that occurs when the channel is passed to it.

Let's look at the specifics of how to implement all this.

4.2 Handling the Data Channel Event

When the peer that creates the channel sends the data channel to the connected peer (which happens automatically if and when the peers are connected by a peer connection), the receiving peer's *RTCPeerConnection* object dispatches an event.

You can (and should) handle that event by assigning a function to the ondatachannel property of the *RTCPeerConnection* object on the receiving peer.

The handler function gets passed a parameter of type *RTCDataChannelEvent*. The main thing you need to know about that type is that it has a *channel* property that will have a reference to the data channel that the sending peer sent.

At that point you should hold on to a reference to that data channel (so that you can send data to the peer) and you should assign some event handler functions to some of the properties of the object. In particular, you should handle message and open events.

That all may still be a bit confusing. Let's look at some simple examples.

In the peer that creates the channel, it will call createDataChannel().

```
var dataChannel = pc.dataChannel("data");
```

The peer that creates the channel should also handle events such as message events on the data channel.

```
dataChannel.onmessage = evt => console.log(evt);
```

The other peer will *not* create the channel. Instead, on its *RTCPeerConnection* object, it will need to listen for the data channel event by assigning a function to the ondatachannel property.

```
pc.ondatachannel = evt => onDataChannel(evt);
```

And when it handles that event, it should hold on to a reference to the channel (by assigning it to a variable that doesn't fall out of scope, for example) and handle events dispatched by the channel.

```
function onDataChannel(evt) {
  dataChannel = evt.channel;
  dataChannel.onmessage = evt => console.log(evt);
}
```

Of course, the same code base will often be used to define either the creator of the data channel or the recipient of it. The only difference between the two in practice is which originates the peer connection handshaking process by sending the offer.

Given that, what you'll typically do is create the data channel at the same time as making the offer. That way you only create the channel on the peer creating the offer.

Then, you'll assign a function to ondatachannel when receiving the offer. That way the only peer that handles ondatachannel is the peer that receives the offer. (Technically you can handle this event in both peers since only one will ever receive the event. But I find it useful for the sake of clarity only to do so in the peer that will need to handle it.)

To make this a bit clearer, in the next section we'll modify the previous handshaking demo by adding in of the code I just mentioned for dealing with a data channel. We'll use that data channel to allow peer-to-peer text chat.

4.3 A Data Channel Text Chat Example

Without too many changes, we can add text chat to our previous handshaking example. First, let's look at the whole thing with the new bits bolded.

```
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
  </head><body>
  <body>

    <div>
      username: <input type="text" id="username" />
      <button id="connectButton"
onclick="connect()">connect</button>
    </div>

    <div>
      your message: <input type="text" id="newMessage" disabled />
      <button id="sendButton" onclick="send()" disabled
>send</button>
    </div>

    <div id="messages">
    </div>

    <script>
      var token;
```

```
        var socket;
        var ice;
        var pc;
        var localUsername;
        var dataChannel;

        function displayMessage(message) {
          $('#messages')[0].innerHTML += message + "<br>";
        }

        window.onload = () => {
          $.post("http://localhost/getice.php", null, r => onIce(r));
        }

        function onIce(r) {
          ice = JSON.parse(r).v;
          pc = new RTCPeerConnection(ice);
          displayMessage("peer connection state:" +
pc.connectionstate);
          pc.onicecandidate = evt => onIceCandidate(evt);
          pc.onconnectionstatechange = evt => displayMessage("peer
connection state:" + pc.connectionState);
          pc.oniceconnectionstatechange = evt => displayMessage("tring
to connect to peer:" + pc.iceConnectionState);
        }

        function connect() {
          localUsername = $('#username')[0].value;
          $.post("http://localhost/gettoken.php", {username:
localUsername}, r => getHost(r));
        }

        function getHost(r) {
          token = JSON.parse(r).v;
          $.post("http://localhost/gethost.php", {username:
localUsername}, r => openSocket(r));
        }

        function openSocket(r) {
          var host = JSON.parse(r).v;
          socket = new WebSocket(host + "/v2/" + token);
          socket.addEventListener("message", onSocketMessage);
        }

        function onSocketMessage(evt) {
          var data = JSON.parse(evt.data);
          var option;
          var selectElement = $("#usersSelect")[0];
          switch (data.m.o) {
            case "peers":
              var users = data.p.users;
```

```
            for(i = 0; i < users.length; i++) {
              displayMessage("user in chat:" + users[i]);
            }
            break;
          case "peer_connected":
            var f = data.m.f.split("/");
            var joining = f[f.length-1];
            displayMessage("new user joined:" + joining)
            callPeer(joining);
            break;
          case "message":
            switch(data.p.msg.type) {
              case "offer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pc.setRemoteDescription(desc);
                pc.ondatachannel = evt => onDataChannel(evt);
                pc.createAnswer().then(d => onCreateAnswer(d,
sender));
                break;
              case "answer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pc.setRemoteDescription(desc);
                break;
              case "candidate":
                displayMessage("you have received a candidate");
                var candidate = new RTCIceCandidate(data.p.msg);
                pc.addIceCandidate(candidate);
            }
        }
      }

      function onCreateOffer(d, peer) {
        pc.setLocalDescription(d);
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }

      function onCreateAnswer(d, peer) {
        displayMessage("you are the answerer...you are answering " +
peer);
        pc.setLocalDescription(d);
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }
```

```
function onIceCandidate(evt) {
  displayMessage("you are sending a candidate");
  var candidate = evt.candidate;
  if (evt.candidate != null) {
              var cPkt = {type: "candidate",
                    sdpMLineIndex: candidate.sdpMLineIndex,
                    sdpMid: candidate.sdpMid,
                    candidate: candidate.candidate
                };
              var pkt = {
                t: "u",
                m: {
                    f: "SampleAppChannel/" + localUsername,
                    o: 'message'
                },
                p: {msg:cPkt}
            }
              socket.send(JSON.stringify(pkt));
  }
}

function callPeer(peer) {
  displayMessage("you are the caller...calling " + peer);
  dataChannel = pc.createDataChannel("data");
  setDataChannelHandlers(dataChannel);
  pc.createOffer().then(d => onCreateOffer(d, peer));
}

function setDataChannelHandlers(dc) {
  dc.onmessage = evt => onDataMessage(evt);
  dc.onopen = evt => onDataChannelOpen(evt);
}

function onDataChannelOpen(evt) {
  $("#newMessage")[0].disabled = false;
  $("#sendButton")[0].disabled = false;
}

function onDataChannel(evt) {
  dataChannel = evt.channel;
  setDataChannelHandlers(dataChannel);
}

function send() {
  var messageElement = $('#newMessage')[0];
  displayMessage("you said: " + messageElement.value);
  var message = {f: localUsername, msg: messageElement.value};
  messageElement.value = "";
  dataChannel.send(JSON.stringify(message));
}
```

```
        function onDataMessage(evt) {
          var messageObj = JSON.parse(evt.data);
          displayMessage(messageObj.f + " said: " + messageObj.msg);
        }

      </script>
    </body>
  </html>
```

Alright. Now let's look at some of the new bits in a little more detail where appropriate.

First, we add a few new page elements, allowing for the user to enter a message and send it. They are disabled to start.

```
      <div>
        your message: <input type="text" id="newMessage" disabled />
        <button id="sendButton" onclick="send()" disabled
>send</button>
      </div>
```

The first notable addition is that we add an ondatachannel handler to the peer connection. But we only need to do this for the answered – not the caller. So we add it to the code that handles the offer.

```
            case "message":
              switch(data.p.msg.type) {
                case "offer":
                  var desc = new RTCSessionDescription(data.p.msg);
                  var f = data.m.f.split("/");
                  var sender = f[f.length-1];
                  pc.setRemoteDescription(desc);
                  pc.ondatachannel = evt => onDataChannel(evt);
                  pc.createAnswer().then(d => onCreateAnswer(d,
sender));
                  break;
```

Next, we want to create the data channel in the caller. We add that code to the callPeer() function since that will only get executed by the calling peer, not the answerer.

```
        function callPeer(peer) {
          displayMessage("you are the caller...calling " + peer);
          dataChannel = pc.createDataChannel("data");
          setDataChannelHandlers(dataChannel);
          pc.createOffer().then(d => onCreateOffer(d, peer));
        }
```

The setDataChannelHandlers() function just adds handlers for message and open events.

```
        function setDataChannelHandlers(dc) {
          dc.onmessage = evt => onDataMessage(evt);
          dc.onopen = evt => onDataChannelOpen(evt);
        }
```

When the data channel opens, we enable the new page elements. That way users can't attempt to send messages until the channel is ready.

```
function onDataChannelOpen(evt) {
  $("#newMessage")[0].disabled = false;
  $("#sendButton")[0].disabled = false;
}
```

When the answerer receives the data channel from the calling peer, onDataChannel handles it. It simply assigns the data channel to a variable so that it doesn't fall out of scope. And it sets the handlers for that channel.

```
function onDataChannel(evt) {
  dataChannel = evt.channel;
  setDataChannelHandlers(dataChannel);
}
```

We configured the send button element to call the send() function on click. This function retrieves the value from the message element, writes it to the display, puts it in a message to send across the data channel, and sends it.

```
function send() {
  var messageElement = $('#newMessage')[0];
  displayMessage("you said: " + messageElement.value);
  var message = {f: localUsername, msg: messageElement.value};
  messageElement.value = "";
  dataChannel.send(JSON.stringify(message));
}
```

And lastly, when handling data channel messages, all we do is parse the message and display it.

```
function onDataMessage(evt) {
  var messageObj = JSON.parse(evt.data);
  displayMessage(messageObj.f + " said: " + messageObj.msg);
}
```

This app works great…for a pair of peers. But it does not allow more than two people to chat. It will break in some weird ways if you attempt it.

So what if we want to allow more than two people to chat at once? Let's see…

4.4 Multi-Peer Data Communication

Now that we've seen how to create a simple working peer-to-peer data communication app, what about enabling multiple pairs of peers to connect and do a group chat?

It's pretty easy to do. What we need to do is to create peer connections between *all* the users in the chat. So each time a new user joins via the signaling channel, each existing peer needs to handshake with the joining user and form a peer connection with it.

In other words, each peer will have n-1 peer connections where n is the number of connected users. For example, A will have peer connections with B and another with C. If D joins, then A will also create a peer connection with D. Meanwhile, B will have peer connections with A, C, and D. C will have peer connections with A, B, and D. And D will have peer connections with A, B, and C.

In addition, each peer connection will need to have its own data channel. Meaning that each peer will also have n-1 data channels – one for each peer connection. A will have separate data channels with B, C, and D, for example.

In a group chat, each peer will broadcast his messages to all of his data channels. But, of course, we could easily add a future feature that allows for private messages since each peer will have a unique data channel with every other peer.

Let's take a look at the code for this multi-user modification to our previous two-person chat. Here it is all in one go with the changes bolded.

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
  </head><body>
  <body>

    <div>
      username: <input type="text" id="username" />
      <button id="connectButton"
onclick="connect()">connect</button>
    </div>

    <div>
      your message: <input type="text" id="newMessage" disabled />
      <button id="sendButton" onclick="send()" disabled
>send</button>
    </div>

    <div id="messages">
    </div>

    <script>
      var token;
      var socket;
      var ice;
      var pcs = {};
      var localUsername;

      function displayMessage(message) {
        $('#messages')[0].innerHTML += message + "<br>";
      }

      window.onload = () => {
        $.post("http://localhost/getice.php", null, r => onIce(r));
      }
```

```
      function onIce(r) {
        ice = JSON.parse(r).v;
        // We deleted code in this section.
      }

      function connect() {
        localUsername = $('#username')[0].value;
        $.post("http://localhost/gettoken.php", {username:
localUsername}, r => getHost(r));
      }

      function getHost(r) {
        token = JSON.parse(r).v;
        $.post("http://localhost/gethost.php", {username:
localUsername}, r => openSocket(r));
      }

      function openSocket(r) {
        var host = JSON.parse(r).v;
        socket = new WebSocket(host + "/v2/" + token);
        socket.addEventListener("message", onSocketMessage);
      }

      function onSocketMessage(evt) {
        var data = JSON.parse(evt.data);
        var option;
        var pc;
        var selectElement = $("#usersSelect")[0];
        switch (data.m.o) {
          case "peers":
            var users = data.p.users;
            for(i = 0; i < users.length; i++) {
              displayMessage("user in chat:" + users[i]);
            }
            break;
          case "peer_connected":
            var f = data.m.f.split("/");
            var joining = f[f.length-1];
            displayMessage("new user joined: " + joining)
            callPeer(joining);
            break;
          case "message":
            switch(data.p.msg.type) {
              case "offer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pc = createNewPeerConnection(sender);
                pc.setRemoteDescription(desc);
                pc.createAnswer().then(d => onCreateAnswer(d,
sender));
```

```
                break;
              case "answer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pcs[sender].pc.setRemoteDescription(desc);
                break;
              case "candidate":
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                console.log("candidate", sender, data, pcs);
                var candidate = new RTCIceCandidate(data.p.msg);
                pcs[sender].pc.addIceCandidate(candidate);
            }
          }
        }

        function callPeer(peer) {
          var pc = createNewPeerConnection(peer);
          var dataChannel = pc.createDataChannel("data");
          pcs[peer].dc = dataChannel;
          setDataChannelHandlers(dataChannel);
          pc.createOffer().then(d => onCreateOffer(d, peer));
        }

        function onCreateOffer(d, peer) {
          pcs[peer].pc.setLocalDescription(d);
          var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
          socket.send(JSON.stringify(pkt));
        }

        function onCreateAnswer(d, peer) {
          pcs[peer].pc.setLocalDescription(d);
          var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
          socket.send(JSON.stringify(pkt));
        }

        function onIceCandidate(evt) {
          var remoteUsername =
getUsernameByRemoteDescription(evt.target.remoteDescription.sdp);
          var candidate = evt.candidate;
          if (candidate != null && remoteUsername != null) {
                      var cPkt = {type: "candidate",
                            sdpMLineIndex: candidate.sdpMLineIndex,
                            sdpMid: candidate.sdpMid,
                            candidate: candidate.candidate
                          };
                      var pkt = {
                        t: "u",
```

```
                        m: {
                             f: "SampleAppChannel/" + localUsername,
                             o: "message",
                             t: remoteUsername
                        },
                        p: {msg:cPkt}
                    }
                      socket.send(JSON.stringify(pkt));
      }
    }

    function setDataChannelHandlers(dc) {
      dc.onmessage = evt => onDataMessage(evt);
      dc.onopen = evt => onDataChannelOpen(evt);
    }

    function onDataChannelOpen(evt) {
      $("#newMessage")[0].disabled = false;
      $("#sendButton")[0].disabled = false;
    }

    function onDataChannel(evt) {
      var dataChannel = evt.channel;
      var keys = Object.keys(pcs);
      var comp;
      var localDescription;
      var remoteDescription;
      for(var i = 0; i < keys.length; i++) {
        comp = pcs[keys[i]];
        if(evt.currentTarget.localDescription.sdp ==
comp.pc.localDescription.sdp) {
          comp.dc = dataChannel;
        }
      }
      setDataChannelHandlers(dataChannel);
    }

    function send() {
      var messageElement = $('#newMessage')[0];
      displayMessage("you said: " + messageElement.value);
      var message = {f: localUsername, msg: messageElement.value};
      messageElement.value = "";
      var dataChannel;
      var keys = Object.keys(pcs);
      var comp;
      for(var i = 0; i < keys.length; i++) {
        comp = pcs[keys[i]];
        dataChannel = comp.dc;
        dataChannel.send(JSON.stringify(message));
      }
    }
```

```
        function onDataMessage(evt) {
          var messageObj = JSON.parse(evt.data);
          displayMessage(messageObj.f + " said: " + messageObj.msg);
        }

        function createNewPeerConnection(username){
          var pc = new RTCPeerConnection(ice);
          pc.ondatachannel = evt => onDataChannel(evt);
          pc.onicecandidate = candidate => onIceCandidate(candidate);
          pc.oniceconnectionstatechange = evt => console.log("ice
  connection state ", evt);
          pcs[username] = {pc: pc, dc: null};
          //console.log("setting new peer connection: ", pcs);
          return pc;
        }


        function getUsernameByRemoteDescription(sdp) {
          var keys = Object.keys(pcs);
          var pc;
          for(var i = 0; i < keys.length; i++) {
            pc = pcs[keys[i]].pc;
            if(pc.remoteDescription.sdp == sdp) {
              return keys[i];
            }
          }
          return null;
        }

      </script>
    </body>
  </html>
```

Let's look at some of that code in a bit more detail.

The most significant thing going on here is that rather than having just one *RTCPeerConnection* object, we have multiple – one for each pair of peers. So we define an associative array called pcs that we'll use to store references to all those connections.

```
var pcs = {};
```

We create those connections lazily as peers come online. To do so, we use a new function called createNewPeerConnection(). We define it a bit further down in the code, but it's important enough that we'll look at it now.

The function creates the new peer connection object and sets up the event handlers. Then, it adds that peer connection to the associative array, using the remote peer's username as the key. The value is an object with properties for the *RTCPeerConnection* object as well as the data channel that we'll assign to it later once we create it.

```
        function createNewPeerConnection(username){
```

```
        var pc = new RTCPeerConnection(ice);
        pc.ondatachannel = evt => onDataChannel(evt);
        pc.onicecandidate = candidate => onIceCandidate(candidate);
        pc.oniceconnectionstatechange = evt => console.log("ice
connection state ", evt);
        pcs[username] = {pc: pc, dc: null};
        return pc;
      }
```

Now, when we want to get a reference to a peer connection, we can do so as long as we know the remote peer's username. The first time we see that in the code is in the onSocketMessage() function in the answer and candidate cases.

```
            case "answer":
              var desc = new RTCSessionDescription(data.p.msg);
              var f = data.m.f.split("/");
              var sender = f[f.length-1];
              pcs[sender].pc.setRemoteDescription(desc);
              break;
            case "candidate":
              var f = data.m.f.split("/");
              var sender = f[f.length-1];
              console.log("candidate", sender, data, pcs);
              var candidate = new RTCIceCandidate(data.p.msg);
              pcs[sender].pc.addIceCandidate(candidate);
```

In the callPeer() function, we create a data channel specific to that peer connection. And we assign a reference to that to the correct element of the associative array where we also keep references to the peer connections.

```
      function callPeer(peer) {
        var pc = createNewPeerConnection(peer);
        var dataChannel = pc.createDataChannel("data");
        pcs[peer].dc = dataChannel;
        setDataChannelHandlers(dataChannel);
        pc.createOffer().then(d => onCreateOffer(d, peer));
      }
```

When handling the event for a receiving a data channel from a peer, things change slightly because now we have to look up the element in the peers associative array and assign the data channel to that element. To do that, we iterate through the elements and compare the remote description of the peer that sent the channel with the remote description of the peer connection in the item from the associative array. If they match, we store a reference to the dataChannel alongside it.

```
      function onDataChannel(evt) {
        var dataChannel = evt.channel;
        var keys = Object.keys(pcs);
        var comp;
        var localDescription;
        var remoteDescription;
        for(var i = 0; i < keys.length; i++) {
```

```
            comp = pcs[keys[i]];
            if(evt.currentTarget.remoteDescription.sdp ==
comp.pc.remoteDescription.sdp) {
              comp.dc = dataChannel;
            }
          }
          setDataChannelHandlers(dataChannel);
        }
```

And when sending messages now we have to send to all the data channels, not just one.

```
        function send() {
          var messageElement = $('#newMessage')[0];
          displayMessage("you said: " + messageElement.value);
          var message = {f: localUsername, msg: messageElement.value};
          messageElement.value = "";
          var dataChannel;
          var keys = Object.keys(pcs);
          var comp;
          for(var i = 0; i < keys.length; i++) {
            comp = pcs[keys[i]];
            dataChannel = comp.dc;
            dataChannel.send(JSON.stringify(message));
          }
        }
```

Otherwise, it's pretty much the same app as before. Just beefed up with more peer connections and data channels.

5 Media Communication

Now that you've done all the heavy lifting of signaling and you've understood data sharing using a data channel, what follows is going to be pretty easy.

There's all kinds of stuff you can do to work with the minutiae of media – controlling the bitrate, video dimensions, etc. But for our purposes here, all we're interested in is the basics: sharing media streams with peers.

Once you understand that much, you can figure out anything else you need pretty easily by reading the API documentation.

So for this guide, all we want to do is get local media (i.e. get access to the user's camera and microphone), send that media to a remote peer, and play the media.

5.1 Getting Access to Local Media

Access to local media (user's camera and microphone) is by permission only. The user has to provide that permission – either by granting permission at the point of request or by having saved a setting previously that authorizes permission.

That means, getting the local media must be an asynchronous operation.

In supporting web browsers you can request that access by calling navigator.mediaDevices.getUserMedia(). The method requires that you provide some constraints and it returns a *Promise*.

The constraints consist of an object with two properties: *audio* and *video*. The simplest version of the constraints is to specify Boolean values for these properties; true if you require the media, false if not. For example, the following constraint object requires both audio and video.

```
{audio: true, video: true}
```

While this object requires audio but not video.

```
{audio: true, video: false}
```

You can also use non-Boolean object values for the *video* property to provide further specifications such as requiring minimum video resolutions or giving preference to one camera over another (such as the front camera versus rear camera on mobile devices). But in our examples, we'll not use any such constraints.

When the promise calls the (success) handler function, it passes it a parameter containing a reference to the *MediaStream* object obtained. The *MediaStream* object holds all the information needed to, for example, play the stream in a video element on an HTML page. (If the user denies access, the failure handler function will be called. But we're not going to bother with that in our examples, despite the fact that in a real app you certainly ought to.)

Here's an example that requests the media stream, and when it receives the stream, it plays it in a video element.

```
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
  </head><body>
  <body>

    <video autoplay="true" id="localVideo"></video>

    <script>

      window.onload = () => {
        var constraints = {audio: true, video: true};
        var p = navigator.mediaDevices.getUserMedia(constraints)
        p.then(stream => onGetMedia(stream));
      }

      function onGetMedia(stream) {
        var vid = $("#localVideo")[0];
        vid.srcObject = stream;
      }
```

```
    </script>
  </body>
</html>
```

5.2 Attaching the Stream to the Peer Connection

Once you've gotten a media stream, you'll next want to attach it to the *RTCPeerConnection* object you are using to communicate with a peer. In previous sections, we've relied on the existence of a data channel in order to prompt a peer connection to complete the negotiation portion of handshaking. Otherwise, peer connections don't broadcast their ICE candidates.

However, a media stream will achieve the same result as a data channel in terms of triggering negotiation. Therefore, keep in mind that if your app does not require a data channel, you can omit it and attach a stream instead.

You attach a stream to a peer connection using the addStream() method. Technically, this method is deprecated, and you *should* use the addTrack() method instead. However, not all browsers have the addTrack() method yet. So for now, unless one wants to use a shim that adds the method, just use addStream() and know that code will break in future browsers once addStream() is no longer supported.

```
function onGetMedia(stream) {
  var vid = $("#localVideo")[0];
  vid.srcObject = stream;
  pc.addStream(stream);
}
```

That's it. The stream is now attached to the peer connection, and the remote peer will be able to make use of it.

5.3 Get the Remote Stream

Once a stream is added to a peer connection, the remote peer will likely want to make use of it. For example, in a video chat app, the remote peer will want to play the stream in a video element on the page.

Getting the stream is pretty darn easy. What you need to do is add an event handler function by assigning it to the onaddstream property of the peer connection object. The peer connection will pass a *MediaStreamEvent* object to the function when it calls the function. That type has among its properties, a *stream* property that references the remote stream.

```
pc.onaddstream = evt => onAddStream(evt);
function onAddStream(evt) {
  console.log(evt.stream);
}
```

If you want to play that stream in a video element on the page, just assign it to the element's srcObject property.

```
pc.onaddstream = evt => onAddStream(evt);
function onAddStream(evt) {
  $("#remoteVideo")[0].srcObject = evt.stream;
}
```

Bada bing, bada boom. It's that easy.

5.4 A Two Person Video Chat Example

Now, let's look at how to use this to add a video to our earlier, two-person text-only chat app.

The additional code needed is so minor that we might as well just look at it. Here's the entire code for the app, additions/changes bolded. (Again, this is a modification to the two-person chat app..not the multiuser version.) Note that I've made a few changes to the HTML elements just to make better use of the screen real estate. I won't bold those. But I did add two new video elements, which I will bold.

```
<html>
  <head>
    <title>Xirsys Demo</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
    <style>
      .col {
        display: flex;
        flex-direction: column;
      }
      .row {
        display: flex;
      }
      #messages {
        overflow-y:scroll;
        height: 100%;
      }
      #localVideo {
        width: 150px;
        height: 150px;
        position: absolute;
        top: 15px;
        right: 15px;
      }
    </style>
  </head><body>
  <body>

    <div class="col">
      <div class="row">

        <div class="col">
          <div>
            your message: <input type="text" id="newMessage"
disabled />
            <button id="sendButton" onclick="send()" disabled
>send</button>
          </div>

          <div id="messages">
          </div>
        </div>
```

```
        <div>
          <video autoplay="true" id="localVideo"></video>
          <video autoplay="true" id="remoteVideo"></video>
        </div>

      </div>
      <div>
        username: <input type="text" id="username" />
        <button id="connectButton"
onclick="connect()">connect</button>
      </div>
    </div>

    <script>
      var token;
      var socket;
      var ice;
      var pc;
      var localUsername;
      var dataChannel;
      var localStream;
      var remoteStream;

      function displayMessage(message) {
        $('#messages')[0].innerHTML += message + "<br>";
      }

      window.onload = () => {
        $.post("http://localhost/getice.php", null, r => onIce(r));
      }

      function onIce(r) {
        ice = JSON.parse(r).v;
        pc = new RTCPeerConnection(ice);
        displayMessage("peer connection state:" +
pc.connectionstate);
        pc.onicecandidate = evt => onIceCandidate(evt);
        pc.onconnectionstatechange = evt => displayMessage("peer
connection state:" + pc.connectionState);
        pc.oniceconnectionstatechange = evt => displayMessage("tring
to connect to peer:" + pc.iceConnectionState);
        pc.onaddstream = evt => onAddStream(evt);
        navigator.mediaDevices.getUserMedia({audio: true, video:
true}).then(stream => onGetMedia(stream));
      }

      function connect() {
        localUsername = $('#username')[0].value;
        $.post("http://localhost/gettoken.php", {username:
localUsername}, r => getHost(r));
```



```
      }

      function getHost(r) {
        token = JSON.parse(r).v;
        $.post("http://localhost/gethost.php", {username:
localUsername}, r => openSocket(r));
      }

      function openSocket(r) {
        var host = JSON.parse(r).v;
        socket = new WebSocket(host + "/v2/" + token);
        socket.addEventListener("message", onSocketMessage);
      }

      function onSocketMessage(evt) {
        var data = JSON.parse(evt.data);
        var option;
        var selectElement = $("#usersSelect")[0];
        switch (data.m.o) {
          case "peers":
            var users = data.p.users;
            for(i = 0; i < users.length; i++) {
              displayMessage("user in chat:" + users[i]);
            }
            break;
          case "peer_connected":
            var f = data.m.f.split("/");
            var joining = f[f.length-1];
            displayMessage("new user joined:" + joining)
            callPeer(joining);
            break;
          case "message":
            switch(data.p.msg.type) {
              case "offer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pc.setRemoteDescription(desc);
                pc.ondatachannel = evt => onDataChannel(evt);
                pc.createAnswer().then(d => onCreateAnswer(d,
sender));
                break;
              case "answer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pc.setRemoteDescription(desc);
                break;
              case "candidate":
                displayMessage("you have received a candidate");
                var candidate = new RTCIceCandidate(data.p.msg);
```

```
                pc.addIceCandidate(candidate);
            }
        }
      }

      function callPeer(peer) {
        displayMessage("you are the caller...calling " + peer);
        dataChannel = pc.createDataChannel("data");
        setDataChannelHandlers(dataChannel);
        pc.createOffer().then(d => onCreateOffer(d, peer));
      }

      function onCreateOffer(d, peer) {
        pc.setLocalDescription(d);
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }

      function onCreateAnswer(d, peer) {
        displayMessage("you are the answerer...you are answering " +
peer);
        pc.setLocalDescription(d);
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }

      function onIceCandidate(evt) {
        displayMessage("you are sending a candidate");
        var candidate = evt.candidate;
        if (evt.candidate != null) {
                    var cPkt = {type: "candidate",
                          sdpMLineIndex: candidate.sdpMLineIndex,
                          sdpMid: candidate.sdpMid,
                          candidate: candidate.candidate
                      };
                    var pkt = {
                      t: "u",
                      m: {
                          f: "SampleAppChannel/" + localUsername,
                          o: 'message'
                      },
                      p: {msg:cPkt}
                  }
                    socket.send(JSON.stringify(pkt));
        }
      }

      function setDataChannelHandlers(dc) {
        dc.onmessage = evt => onDataMessage(evt);
```

```
        dc.onopen = evt => onDataChannelOpen(evt);
      }

      function onDataChannelOpen(evt) {
        $("#newMessage")[0].disabled = false;
        $("#sendButton")[0].disabled = false;
      }

      function onDataChannel(evt) {
        //console.log(evt);
        dataChannel = evt.channel;
        setDataChannelHandlers(dataChannel);
      }

      function send() {
        var messageElement = $('#newMessage')[0];
        displayMessage("you said: " + messageElement.value);
        var message = {f: localUsername, msg: messageElement.value};
        messageElement.value = "";
        dataChannel.send(JSON.stringify(message));
      }

      function onDataMessage(evt) {
        var messageObj = JSON.parse(evt.data);
        displayMessage(messageObj.f + " said: " + messageObj.msg);
      }

      function onAddStream(evt) {
        remoteStream = evt.stream;
        var vid = $("#remoteVideo")[0];
        vid.srcObject = remoteStream;
      }

      function onGetMedia(stream) {
        var vid = $("#localVideo")[0];
        vid.srcObject = stream;
        pc.addStream(stream);
        localStream = stream;
      }

    </script>
  </body>
</html>
```

As you can see, the changes amount to only around a dozen simple lines of code. Let's look at those changes/additions in a little more detail.

First, I added some video elements to the page. These are for playing the video streams. One is to play the local stream. The other is to play the remote stream.

```
<video autoplay="true" id="localVideo"></video>
<video autoplay="true" id="remoteVideo"></video>
```

Next, I added two new variables. In truth, I'm not even using these for anything needed right now. But I like to keep some references to the streams that won't fall out of scope.

```
  var localStream;
  var remoteStream;
```

Then, I added two lines of code in onICE(). One adds the handler for onaddstream. The other requests the user media.

```
pc.onaddstream = evt => onAddStream(evt);
navigator.mediaDevices.getUserMedia({audio: true, video:
true}).then(stream => onGetMedia(stream));
```

I next define onAddStream(). Remember that this handler gets passed a MediaStreamEvent. That has a stream property with the remote stream. So I assign that to the remoteStream variable. Then I assign it to the srcObject property of the remoteVideo page element so that it will play on the page.

```
function onAddStream(evt) {
  remoteStream = evt.stream;
  var vid = $("#remoteVideo")[0];
  vid.srcObject = remoteStream;
}
```

Lastly, I define onGetMedia(). This is the success handler for getUserMedia(). As such, it gets passed a MediaStream object. Here, I assign that stream to the localVideo element's srcObject so it plays on the page. Then I add the stream to the peer connection, which will (once the connection is made) cause the onaddstream event handler to run on the remote peer. Then I assign the stream to the localStream variable so it doesn't fall out of scope and I can use it later if I want to.

```
function onGetMedia(stream) {
  var vid = $("#localVideo")[0];
  vid.srcObject = stream;
  pc.addStream(stream);
  localStream = stream;
}
```

And that's that. A working, albeit simple, video chat app.

5.5 A Multi-Person Video Chat App

We can allow for more than two users to join the video chat by using a similar approach to the way in which we increased the text chat capacity from two to many. In fact, we'll add video to the existing multiuser text chat by changing that code.

Here's the whole thing in one go with the important changes in bold. Note also that I've made a few changes to the page element configuration and the styles applied to them to support dynamically adding video elements. But since our focus is on WebRTC, we won't discuss those changes.

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Xirsys Demo</title>
```

```
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.j
s"></script>
    <style>
      .col {
        display: flex;
        flex-direction: column;
      }
      .row {
        display: flex;
      }
      #messages {
        overflow-y:scroll;
        height: 100%;
      }
      video {
        background: #cccccc;
        width: 200px;
      }

    </style>
  </head><body>
  <body>

    <div class="col">
      <div class="row">

        <div class="col">
          <div>
            your message: <input type="text" id="newMessage"
disabled />
            <button id="sendButton" onclick="send()" disabled
>send</button>
          </div>

          <div id="messages">
          </div>
        </div>

        <div id="videos">
          <video autoplay="true" id="localVideo"></video>
        </div>

      </div>
      <div>
        username: <input type="text" id="username" />
        <button id="connectButton"
onclick="connect()">connect</button>
      </div>
    </div>
```



```
    <script>
      var token;
      var socket;
      var ice;
      var pcs = {};
      var localUsername;

      function displayMessage(message) {
        $('#messages')[0].innerHTML += message + "<br>";
      }

      window.onload = () => {
        $.post("http://localhost/getice.php", null, r => onIce(r));
      }

      function onIce(r) {
        ice = JSON.parse(r).v;
        navigator.mediaDevices.getUserMedia({audio: true, video:
true}).then(stream => onGetMedia(stream));
      }

      function connect() {
        localUsername = $('#username')[0].value;
        $.post("http://localhost/gettoken.php", {username:
localUsername}, r => getHost(r));
      }

      function getHost(r) {
        token = JSON.parse(r).v;
        console.log("token", r);
        $.post("http://localhost/gethost.php", {username:
localUsername}, r => openSocket(r));
      }

      function openSocket(r) {
        var host = JSON.parse(r).v;
        socket = new WebSocket(host + "/v2/" + token);
        socket.addEventListener("message", onSocketMessage);
      }

      function onSocketMessage(evt) {
        var data = JSON.parse(evt.data);
        var option;
        var pc;
        var selectElement = $("#usersSelect")[0];
        switch (data.m.o) {
          case "peers":
            var users = data.p.users;
            for(i = 0; i < users.length; i++) {
              displayMessage("user in chat:" + users[i]);
            }
```



```
            break;
          case "peer_connected":
            var f = data.m.f.split("/");
            var joining = f[f.length-1];
            displayMessage("new user joined: " + joining)
            callPeer(joining);
            break;
          case "message":
            switch(data.p.msg.type) {
              case "offer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pc = createNewPeerConnection(sender);
                pc.setRemoteDescription(desc);
                pc.createAnswer().then(d => onCreateAnswer(d,
sender));
                break;
              case "answer":
                var desc = new RTCSessionDescription(data.p.msg);
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                pcs[sender].pc.setRemoteDescription(desc);
                break;
              case "candidate":
                var f = data.m.f.split("/");
                var sender = f[f.length-1];
                var candidate = new RTCIceCandidate(data.p.msg);
                pcs[sender].pc.addIceCandidate(candidate);
            }
        }
      }

      function callPeer(peer) {
        var pc = createNewPeerConnection(peer);
        var dataChannel = pc.createDataChannel("data");
        pcs[peer].dc = dataChannel;
        setDataChannelHandlers(dataChannel);
        pc.createOffer().then(d => onCreateOffer(d, peer));
      }

      function onCreateOffer(d, peer) {
        pcs[peer].pc.setLocalDescription(d);
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }

      function onCreateAnswer(d, peer) {
        pcs[peer].pc.setLocalDescription(d);
```

```
        var pkt = {t: "u", m: {f: "SampleAppChannel/" +
localUsername, o: "message", t: peer}, p: {msg:d}};
        socket.send(JSON.stringify(pkt));
      }

      function onIceCandidate(evt) {
        var remoteUsername =
getUsernameByRemoteDescription(evt.target.remoteDescription.sdp);
        var candidate = evt.candidate;
        if (candidate != null && remoteUsername != null) {
                    var cPkt = {type: "candidate",
                          sdpMLineIndex: candidate.sdpMLineIndex,
                          sdpMid: candidate.sdpMid,
                          candidate: candidate.candidate
                      };
                    var pkt = {
                      t: "u",
                      m: {
                          f: "SampleAppChannel/" + localUsername,
                          o: "message",
                          t: remoteUsername
                      },
                      p: {msg:cPkt}
                  }
                    socket.send(JSON.stringify(pkt));
        }
      }

      function setDataChannelHandlers(dc) {
        dc.onmessage = evt => onDataMessage(evt);
        dc.onopen = evt => onDataChannelOpen(evt);
      }

      function onDataChannelOpen(evt) {
        $("#newMessage")[0].disabled = false;
        $("#sendButton")[0].disabled = false;
      }

      function onDataChannel(evt) {
        var dataChannel = evt.channel;
        var keys = Object.keys(pcs);
        var comp;
        var localDescription;
        var remoteDescription;
        for(var i = 0; i < keys.length; i++) {
          comp = pcs[keys[i]];
          if(evt.currentTarget.localDescription.sdp ==
comp.pc.localDescription.sdp) {
            comp.dc = dataChannel;
          }
        }
```

```
  setDataChannelHandlers(dataChannel);
}

function send() {
  var messageElement = $('#newMessage')[0];
  displayMessage("you said: " + messageElement.value);
  var message = {f: localUsername, msg: messageElement.value};
  messageElement.value = "";
  var dataChannel;
  var keys = Object.keys(pcs);
  var comp;
  for(var i = 0; i < keys.length; i++) {
    comp = pcs[keys[i]];
    dataChannel = comp.dc;
    dataChannel.send(JSON.stringify(message));
  }
}

function onDataMessage(evt) {
  var messageObj = JSON.parse(evt.data);
  displayMessage(messageObj.f + " said: " + messageObj.msg);
}

function createNewPeerConnection(username){
  var pc = new RTCPeerConnection(ice);
  pc.addStream(localStream);
  pc.onaddstream = evt => onAddStream(evt);
  pc.ondatachannel = evt => onDataChannel(evt);
  pc.onicecandidate = candidate => onIceCandidate(candidate);
  pcs[username] = {pc: pc, dc: null, s: null, v: null};
  return pc;
}

function getUsernameByRemoteDescription(sdp) {
  var keys = Object.keys(pcs);
  var pc;
  for(var i = 0; i < keys.length; i++) {
    pc = pcs[keys[i]].pc;
    if(pc.remoteDescription.sdp == sdp) {
      return keys[i];
    }
  }
  return null;
}

function onGetMedia(stream) {
  var vid = $("#localVideo")[0];
  vid.srcObject = stream;
  localStream = stream;
}
```



```
        function onAddStream(evt) {
          var pc = evt.target;
          var stream = evt.stream;
          var peer =
  getUsernameByRemoteDescription(pc.remoteDescription.sdp);
          pcs[peer].s = stream;
          var v = addNewVideo(stream);
          pcs[peer].v = v;
        }

        function addNewVideo(stream) {
          var vid = document.createElement("video");
          $("#videos")[0].appendChild(vid);
          vid.srcObject = stream;
          return vid;
        }

    </script>
  </body>
</html>
```

These few changes should mostly be familiar to you since they are variations on what we already did in making the two-person video chat app. So we don't even need to discuss them for the most part.

The main thing to note is that we added v and s properties to the items in the associative array. Those hold references to the video element and the stream associated with that remote peer.

Otherwise, the only notable change is the addNewVideo() function, which creates the new video element and appends it to the div on the screen, assigns the stream to the video element, and returns that video element so it can be added to the associative array.

The thing that you will almost certainly find with this multiuser video chat app, however, is that it slows down dramatically with each additional user that joins. On my laptop, which is pretty slow, I cannot even get a reasonable performance with just three users! And four causes it to grind pretty close to a halt.

This is a limitation of the processing power of the peer devices. And until which time the processing power of devices on the whole rises to meet the demands of playing multiple video streams simultaneously, we'll need to solve this problem by adding a server in the middle of the communication that combines the streams into a single video that gets broadcast to the peers.

6 What's to Come

This is the end of this first edition guide.

My sincere hope is that this guide has answered your core questions regarding WebRTC and set you on a path of competence and confidence with this technology.

As I stated from the outset, my intention with this book has been to reduce your learning curve by providing you with a clear picture of the basics of WebRTC along with some simple, easy-to-follow examples.

WebRTC will continue to evolve. And solutions providers will continue to provide new, creative solutions to the problems that app developers face. As such, you should stay up to date.

That can be a daunting task. But one simple way to do that is to rely on Xirsys as one of your key sources of WebRTC news. Keep your finger on the pulse of WebRTC without having to do the sleuthing yourself. Simply sign up for the free Xirsys newsletter for relevant updates that will help you stay up to date.

Furthermore, when you sign up for the free Xirsys newsletter, you'll be guaranteed to receive future editions of this book when we publish them.

Here's some of what we're planning for future editions:

- Recording video and audio stream for later playback
- Real-time processing of video and audio (for example, adding watermarks)
- One-to-many video/audio broadcasts
- Screensharing

Sign up now and we'll notify you when you can get updated editions for free.