WebRTC basics

Once you understand the WebRTC architecture, you can read this article, which takes you through the creation of a cross-browser RTC App. By the end of it you should have working peer-to-peer Datachannels and Media.

Examples on this page outdated! Don't try to repeat them.

Note

Due to recent changes in the API there are many old examples that require fixing:

- louisstow
- mozilla
- webrtc-experiment

The currently working example is:

- apprtc (source)

Implementation may be inferred from the specification.

This remainder of this page contains outdated information as noted on bugzilla.

Shims

As you can imagine, with such an early API, you must use the browser prefixes and shim it to a common variable.

```
1 var RTCPeerConnection = window.mozRTCPeerConnection || window.webkitRTCPeerConnection;
2 var IceCandidate = window.mozRTCIceCandidate || window.RTCIceCandidate;
3 var SessionDescription = window.mozRTCSessionDescription || window.RTCSessionDescription
4 navigator.getUserMedia = navigator.getUserMedia || navigator.mozGetUserMedia || navigato
```

RTCPeerConnection

This is the starting point to creating a connection with a peer. It accepts configuration options about ICE servers to use to establish a connection.

```
var pc = new RTCPeerConnection(configuration);
```

`RTCConfiguration`

The `RTCConfiguration` object contains information about which TURN and/or STUN servers to use for ICE. This is required to ensure most users can actually create a connection by avoiding restrictions in NAT and firewalls.

```
var configuration = {
    iceServers: [
        {urls: "stun:23.21.150.121"},
        {urls: "stun:stun.l.google.com:19302"},
        {urls: "turn:numb.viagenie.ca", credential: "webrtcdemo", username: "louis%40moz
    ]
}
```

Google runs a public STUN server that we can use. I also created an account at http://numb.viagenie.ca/ for a free TURN server to access. You may want to do the same and replace with your own credentials.

ICECandidate

After creating the PeerConnection and passing in the available STUN and TURN servers, an event will be fired once the ICE framework has found some “candidates” that will allow you to connect with a peer. This is known as an ICE Candidate and will execute a callback function on PeerConnection#onicecandidate.

```
pc.onicecandidate = function (e) {
    // candidate exists in e.candidate
    if (!e.candidate) return;
    send("icecandidate", JSON.stringify(e.candidate));
};
```

When the callback is executed, we must use the signal channel to send the Candidate to the peer. On Chrome, multiple ICE candidates are usually found, we only need one so I typically send the first one then remove the handler. Firefox includes the Candidate in the Offer SDP.

Signal Channel

Now that we have an ICE candidate, we need to send that to our peer so they know how to connect with us. However this leaves us with a chicken and egg situation; we want PeerConnection to send data to a peer but before that we need to send them metadata...

This is where the signal channel comes in. It's any method of data transport that allows two peers to exchange information. In this article, we're going to use FireBase because it's incredibly easy to setup and doesn't require any hosting or server-code.

For now just imagine two methods exist: `send()` will take a key and assign data to it and `recv()` will call a handler when a key has a value.

The structure of the database will look like this:

```
{
    "": {
        "candidate:": …
        "offer": …
        "answer": …
    }
}
```

Connections are divided by a `roomId` and will store 4 pieces of information, the ICE candidate from the offerer, the ICE candidate from the answerer, the offer SDP and the answer SDP.

Offer

An Offer SDP (Session Description Protocol) is metadata that describes to the other peer the format to expect (video, formats, codecs, encryption, resolution, size, etc etc).

An exchange requires an offer from a peer, then the other peer must receive the offer and provide back an answer.

```
pc.createOffer(function (offer) {
    pc.setLocalDescription(offer, function() {
        send("offer", JSON.stringify(pc.localDescription);
    }, errorHandler);
}, errorHandler, options);
```

errorHandler

If there was an issue generating an offer, this method will be executed with error details as the first argument.

```
var errorHandler = function (err) {
    console.error(err);
};
```

options

Options for the offer SDP.

```
var options = {
    offerToReceiveAudio: true,
    offerToReceiveVideo: true
};
```

`offerToReceiveAudio/Video` tells the other peer that you would like to receive video or audio from them. This is not needed for DataChannels.

Once the offer has been generated we must set the local SDP to the new offer and send it through the signal channel to the other peer and await their Answer SDP.

Answer

An Answer SDP is just like an offer but a response; sort of like answering the phone. We can only generate an answer once we have received an offer.

```
recv("offer", function (offer) {
    offer = new SessionDescription(JSON.parse(offer))
    pc.setRemoteDescription(offer);

    pc.createAnswer(function (answer) {
        pc.setLocalDescription(answer, function() {
            send("answer", JSON.stringify(pc.localDescription));
        }, errorHandler);
    }, errorHandler);
});
```

DataChannel

I will first explain how to use PeerConnection for the DataChannels API and transferring arbitrary data between peers.

Note: At the time of this article, interoperability between Chrome and Firefox is not possible with DataChannels. Chrome supports a similar but private protocol and will be supporting the standard protocol soon.

```
1 var channel = pc.createDataChannel(channelName, channelOptions);
```

The offerer should be the peer who creates the channel. The answerer will receive the channel in the callback `ondatachannel` on PeerConnection. You must call `createDataChannel()` once before creating the offer.

`channelName`

This is a string that acts as a label for your channel name. *Warning: Make sure your channel name has no spaces or Chrome will fail on `createAnswer()`.*

`channelOptions`

```
1 var channelOptions = {};
```

Currently these options are not well supported on Chrome so you can leave this empty for now. Check the RFC for more information about the options.

Channel Events and Methods

onopen

Executed when the connection is established.

`onerror`

Executed if there is an error creating the connection. First argument is an error object.

```
1 channel.onerror = function (err) {
2     console.error("Channel Error:", err);
3 };
```

`onmessage`

```
channel.onmessage = function (e) {
    console.log("Got message:", e.data);
}
```

The heart of the connection. When you receive a message, this method will execute. The first argument is an event object which contains the data, time received and other information.

`onclose`

Executed if the other peer closes the connection.

Binding the Events

If you were the creator of the channel (meaning the offerer), you can bind events directly to the DataChannel you created with `createChannel`. If you are the answerer, you must use the `ondatachannel` callback on PeerConnection to access the same channel.

```
pc.ondatachannel = function (e) {
    e.channel.onmessage = function () { … };
};
```

The channel is available in the event object passed into the handler as `e.channel`.

`send()`

```
channel.send("Hi Peer!");
```

This method allows you to send data directly to the peer! Amazing. You must send either String, Blob, ArrayBuffer or ArrayBufferView, so be sure to stringify objects.

`close()`

Close the channel once the connection should end. It is recommended to do this on page unload.

Media

Now we will cover transmitting media such as audio and video. To display the video and audio you must include a `<video>` tag on the document with the attribute `autoplay`.

Get User Media

```
<video id="preview" autoplay></video>

```

```
var video = document.getElementById("preview");
navigator.getUserMedia(constraints, function (stream) {
    video.src = URL.createObjectURL(stream);
}, errorHandler);
```

`constraints`

Constraints on what media types you want to return from the user.

```
var constraints = {
    video: true,
    audio: true
};
```

If you just want an audio chat, remove the `video` member.

`errorHandler`

Executed if there is an error returning the requested media.

Media Events and Methods

`addStream`

Add the stream from `getUserMedia` to the PeerConnection.

```
pc.addStream(stream);
```

`onaddstream`

```
<video id="otherPeer" autoplay></video>

var otherPeer = document.getElementById("otherPeer");
pc.onaddstream = function (e) {
    otherPeer.src = URL.createObjectURL(e.stream);
};
```

Executed when the connection has been setup and the other peer has added the stream to the peer connection with `addStream`. You need another `<video>` tag to display the other peer's media.

The first argument is an event object with the other peer's media stream.