Getting Started with WebRTC

By Sam Dutton

Published: July 23rd, 2012
Updated: February 21st, 2014
Comments: 0

WebRTC is a new front in the long war for an open and unencumbered web.

- Brendan Eich, inventor of JavaScript

Real-time communication without plugins

Imagine a world where your phone, TV and computer could all communicate on a common platform. Imagine it was easy to add video chat and peer-to-peer data sharing to your web application. That's the vision of WebRTC.

Want to try it out? WebRTC is available now in Google Chrome, Safari, Firefox and Opera, on desktop and mobile. A good place to start is the simple video chat application at appr.tc:

1. Open appr.tc in your browser.
2. Click the Join button to join a chat room and let the app use your webcam.
3. Open the URL displayed at the bottom of the page in a new tab or, better still, on a different computer.

Quick start

Haven't got time to read this article, or just want code?

1. Get an overview of WebRTC from the Google I/O presentation (the slides are here):

cancellation techniques. Google open sourced the technologies developed by GIPS and engaged with relevant standards bodies at the IETF and W3C to ensure industry consensus. In May 2011, Ericsson built the first implementation of WebRTC.

WebRTC implemented open standards for real-time, plugin-free video, audio and data communication. The need was real:

- Many web services used RTC, but needed downloads, native apps or plugins. Those included Skype, Facebook and Google Hangouts.
- Downloading, installing and updating plugins is complex, error prone and annoying.
- Plugins are difficult to deploy, debug, troubleshoot, test and maintain—and may require licensing and integration with complex, expensive technology. It's often difficult to persuade people to install plugins in the first place!

The guiding principles of the WebRTC project are that its APIs should be open source, free, standardized, built into web browsers and more efficient than existing technologies.

Where are we now?

WebRTC is used in various apps like WhatsApp, Facebook Messenger, appear.in and platforms such as TokBox. WebRTC has also been integrated with WebKitGTK+ and Qt native apps.

WebRTC implements three APIs:

- `MediaStream` (aka `getUserMedia`)
- `RTCPeerConnection`
- `RTCDataChannel`

The APIs are defined in two specs:

- WebRTC
- getUserMedia

All three APIs are supported on mobile and desktop by Chrome, Safari, Firefox, Edge and Opera.

`getUserMedia`: View the demos and code at webrtc.github.io/samples or try out Chris Wilson's amazing examples that use `getUserMedia` as input for Web Audio.

`RTCPeerConnection`: There's an ultra-simple demo at webrtc.github.io/samples and a fully functional video chat application at appr.tc. This app uses adapter.js, a JavaScript shim, maintained Google with help from the WebRTC community, to abstracts away browser differences and spec changes.

2. If you haven't used getUserMedia, take a look at the HTML5 Rocks article and view the source for the simple example at simpl.info/gum.
3. Get to grips with the RTCPeerConnection API by reading through the example below and the demo at simpl.info/pc, which implements WebRTC on a single web page.
4. Learn more about how WebRTC uses servers for signaling, and firewall and NAT traversal, by reading through the code and console logs from appr.tc.
5. Can't wait and just want to try out WebRTC right now? Try out some of the 20+ demos that exercise the WebRTC JavaScript APIs.
6. Having trouble with your machine and WebRTC? Try out our troubleshooting page test.webrtc.org.

Alternatively, jump straight into our WebRTC codelab: a step-by-step guide that explains how to build a complete video chat app, including a simple signaling server.

A very short history of WebRTC

One of the last major challenges for the web is to enable human communication via voice and video: Real Time Communication, RTC for short. RTC should be as natural in a web application as entering text in a text input. Without it, we're limited in our ability to innovate and develop new ways for people to interact.

Historically, RTC has been corporate and complex, requiring expensive audio and video technologies to be licensed or developed in house. Integrating RTC technology with existing content, data and services has been difficult and time consuming, particularly on the web.

Gmail video chat became popular in 2008, and in 2011 Google introduced Hangouts, which use the Google Talk service (as did Gmail). Google bought GIPS, a company which had developed many components required for RTC, such as codecs and echo

`RTCDataChannel`: Check out one of the data channel demos at webrtc.github.io/samples to see this in action.

Our WebRTC codelab shows how to use all three APIs to build a simple application for video chat and file sharing.

My first WebRTC

WebRTC applications need to do several things:

- Get streaming audio, video or other data.
- Get network information such as IP addresses and ports, and exchange this with other WebRTC clients (known as *peers*) to enable connection, even through NATs and firewalls.
- Coordinate signaling communication to report errors and initiate or close sessions.
- Exchange information about media and client capability, such as resolution and codecs.
- Communicate streaming audio, video or data.

To acquire and communicate streaming data, WebRTC implements the following APIs:

- MediaStream: get access to data streams, such as from the user's camera and microphone.
- RTCPeerConnection: audio or video calling, with facilities for encryption and bandwidth management.
- RTCDataChannel: peer-to-peer communication of generic data.

(There is detailed discussion of the network and signaling aspects of WebRTC below.)

MediaStream (aka getUserMedia)

The MediaStream API represents synchronized streams of media. For example, a stream taken from camera and microphone input has synchronized video and audio tracks. (Don't confuse MediaStreamTrack with the <track> element, which is something entirely different.)

Probably the easiest way to understand MediaStream is to look at it in the wild:

1. In your browser, open the demo at webrtc.github.io/samples/src/content/getusermedia/gum.
2. Open the console.
3. Inspect the `stream` variable, which is in global scope.

Each MediaStream has an input, which might be a MediaStream generated by `getUserMedia()`, and an output, which might be passed to a video element or an

RTCPeerConnection.

The `getUserMedia()` method takes a MediaStreamConstraints object parameter, and returns a Promise that resolves to a MediaStream object.

Each MediaStream has a `label`, such as'Xk7EuLhsuHKbnjLWkW4yYGNJJ8ONsgwHBvLQ'. An array of MediaStreamTracks is returned by the `getAudioTracks()` and `getVideoTracks()` methods.

For the webrtc.github.io/samples/src/content/getusermedia/gum example, `stream.getAudioTracks()` returns an empty array (because there's no audio) and, assuming a working webcam is connected, `stream.getVideoTracks()` returns an array of one MediaStreamTrack representing the stream from the webcam. Each MediaStreamTrack has a kind ('video' or 'audio'), and a label (something like 'FaceTime HD Camera (Built-in)'), and represents one or more channels of either audio or video. In this case, there is only one video track and no audio, but it is easy to imagine use cases where there are more: for example, a chat application that gets streams from the front camera, rear camera, microphone, and a 'screenshared' application.

A MediaStream can be attached to a video element by setting the `srcObject` attribute.. Previously this was done by setting the `src` attribute to an object URL created with `URL.createObjectURL()`, but this has been deprecated.

Tip!

The MediaStreamTrack is actively using the camera, which takes resources and keeps the camera open (and camera light on). When you are no longer using a track make sure to call `track.stop()` so that the camera can be closed.

`getUserMedia` can also be used as an input node for the Web Audio API:

```
// cope with browser differences
let audioContext;
if (typeof AudioContext === 'function') {
  audioContext = new AudioContext();
} else if (typeof webkitAudioContext === 'function') {
  audioContext = new webkitAudioContext(); // eslint-disable-line
new-cap
} else {
  console.log('Sorry! Web Audio not supported.');
}

// create a filter node
var filterNode = audioContext.createBiquadFilter();
// see https://dvcs.w3.org/hg/audio/raw-
file/tip/webaudio/specification.html#BiquadFilterNode-section
filterNode.type = 'highpass';
// cutoff frequency: for highpass, audio is attenuated below this
frequency
```

```
filterNode.frequency.value = 10000;

// create a gain node (to change audio volume)
var gainNode = audioContext.createGain();
// default is 1 (no change); less than 1 means audio is attenuated
// and vice versa
gainNode.gain.value = 0.5;

navigator.mediaDevices.getUserMedia({audio: true}, (stream) => {
  // Create an AudioNode from the stream
  const mediaStreamSource =
    audioContext.createMediaStreamSource(stream);
  mediaStreamSource.connect(filterNode);
  filterNode.connect(gainNode);
  // connect the gain node to the destination (i.e. play the
sound)
  gainNode.connect(audioContext.destination);
});
```

Chromium-based apps and extensions can also incorporate `getUserMedia`. Adding `audioCapture` and/or `videoCapture` permissions to the manifest enables permission to be requested and granted only once, on installation. Thereafter the user is not asked for permission for camera or microphone access.

Permission only has to be granted once for `getUserMedia()`. First time around, an Allow button is displayed in the browser's infobar. HTTP access for `getUserMedia()` was deprecated by Chrome at the end of 2015 due to it being classified as a Powerful feature.

The intention is potentially to enable a MediaStream for any streaming data source, not just a camera or microphone. This would enable streaming from disc, or from arbitrary data sources such as sensors or other inputs.

`getUserMedia()` really comes to life in combination with other JavaScript APIs and libraries:

- Webcam Toy is a photobooth app that uses WebGL to add weird and wonderful effects to photos which can be shared or saved locally.
- FaceKat is a 'face tracking' game built with headtrackr.js.
- ASCII Camera uses the Canvas API to generate ASCII images.

gUM ASCII art!

Constraints

Constraints can be used to set values for video resolution for `getUserMedia()`. This also allows support for other constraints such as aspect ratio, facing mode (front or back camera), frame rate, height and width, along with an `applyConstraints()` method.

There's an example at webrtc.github.io/samples/src/content/getusermedia/resolution.

One gotcha: getUserMedia constraints may affect the available configurations of a shared resource. For example if a camera was opened in 640x480 mode by one tab, another tab will not be able to use constraints to open it in a higher-resolution mode since it can only be opened in one mode. Note that this is an implementation detail: it would be possible to let the second tab reopen the camera in a higher resolution mode and use video processing to downscale the video track to 640x480 for the first tab, but this has not been implemented.

Setting a disallowed constraint value gives a DOMException, or an OverconstrainedError if (for example) a resolution requested is not available. To see this

in action, try the demo at webrtc.github.io/samples/src/content/getusermedia/resolution.

Screen and tab capture

Chrome apps also make it possible to share a live 'video' of a single browser tab or the entire desktop via chrome.tabCapture and chrome.desktopCapture APIs. (There's a demo and more information in the HTML5 Rocks Update article Screensharing with WebRTC. A few years old, but still interesting.)

It's also possible to use screen capture as a MediaStream source in Chrome using the experimental chromeMediaSource constraint, as in this demo. Note that screen capture requires HTTPS and should only be used for development due to it being enabled via a command line flag as explaind in this discuss-webrtc post.

Signaling: session control, network and media information

WebRTC uses RTCPeerConnection to communicate streaming data between browsers (aka peers), but also needs a mechanism to coordinate communication and to send control messages, a process known as signaling. Signaling methods and protocols are *not* specified by WebRTC: signaling is not part of the RTCPeerConnection API.

Instead, WebRTC app developers can choose whatever messaging protocol they prefer, such as SIP or XMPP, and any appropriate duplex (two-way) communication channel. The appr.tc example uses XHR and the Channel API as the signaling mechanism. The codelab we built uses Socket.io running on a Node server.

Signaling is used to exchange three types of information:

- Session control messages: to initialize or close communication and report errors.
- Network configuration: to the outside world, what's my computer's IP address and port?
- Media capabilities: what codecs and resolutions can be handled by my browser and the browser it wants to communicate with?

The exchange of information via signaling must have completed successfully before peer-to-peer streaming can begin.

For example, imagine Alice wants to communicate with Bob. Here's a code sample from the W3C WebRTC spec, which shows the signaling process in action. The code assumes the existence of some signaling mechanism, created in the `createSignalingChannel()` method. Also note that on Chrome and Opera, RTCPeerConnection is currently prefixed.

```
// handles JSON.stringify/parse
const signaling = new SignalingChannel();
const constraints = {audio: true, video: true};
const configuration = {iceServers: [{urls:
'stuns:stun.example.org'}]};
const pc = new RTCPeerConnection(configuration);

// send any ice candidates to the other peer
pc.onicecandidate = ({candidate}) => signaling.send({candidate});

// let the "negotiationneeded" event trigger offer generation
pc.onnegotiationneeded = async () => {
  try {
    await pc.setLocalDescription(await pc.createOffer());
    // send the offer to the other peer
    signaling.send({desc: pc.localDescription});
  } catch (err) {
    console.error(err);
  }
};

// once remote track media arrives, show it in remote video
element
pc.ontrack = (event) => {
  // don't set srcObject again if it is already set.
  if (remoteView.srcObject) return;
  remoteView.srcObject = event.streams[0];
};

// call start() to initiate
async function start() {
  try {
    // get local stream, show it in self-view and add it to be
sent
    const stream =
      await navigator.mediaDevices.getUserMedia(constraints);
    stream.getTracks().forEach((track) =>
      pc.addTrack(track, stream));
    selfView.srcObject = stream;
  } catch (err) {
    console.error(err);
  }
}

signaling.onmessage = async ({desc, candidate}) => {
  try {
    if (desc) {
      // if we get an offer, we need to reply with an answer
      if (desc.type === 'offer') {
        await pc.setRemoteDescription(desc);
        const stream =
          await navigator.mediaDevices.getUserMedia(constraints);
        stream.getTracks().forEach((track) =>
          pc.addTrack(track, stream));
```

```
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send({desc: pc.localDescription});
      } else if (desc.type === 'answer') {
        await pc.setRemoteDescription(desc);
      } else {
        console.log('Unsupported SDP type.');
      }
    } else if (candidate) {
      await pc.addIceCandidate(candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

First up, Alice and Bob exchange network information. (The expression 'finding candidates' refers to the process of finding network interfaces and ports using the ICE framework.)

1. Alice creates an RTCPeerConnection object with an `onicecandidate` handler.
2. The handler is run when network candidates become available.
3. Alice sends serialized candidate data to Bob, via whatever signaling channel they are using: WebSocket or some other mechanism.
4. When Bob gets a candidate message from Alice, he calls `addIceCandidate`, to add the candidate to the remote peer description.

WebRTC clients (known as **peers**, aka Alice and Bob) also need to ascertain and exchange local and remote audio and video media information, such as resolution and codec capabilities. Signaling to exchange media configuration information proceeds by exchanging an *offer* and an *answer* using the Session Description Protocol (SDP):

1. Alice runs the RTCPeerConnection `createOffer()` method. The return from this of this is passed an RTCSessionDescription: Alice's local session description.
2. In the callback, Alice sets the local description using `setLocalDescription()` and then sends this session description to Bob via their signaling channel. Note that RTCPeerConnection won't start gathering candidates until `setLocalDescription()` is called: this is codified in JSEP IETF draft.
3. Bob sets the description Alice sent him as the remote description using `setRemoteDescription()`.
4. Bob runs the RTCPeerConnection `createAnswer()` method, passing it the remote description he got from Alice, so a local session can be generated that is compatible with hers. The `createAnswer()` callback is passed an RTCSessionDescription: Bob sets that as the local description and sends it to Alice.
5. When Alice gets Bob's session description, she sets that as the remote description with `setRemoteDescription`.

6. Ping!

Tip!

> *Make sure to allow the RTCPeerConnection to be garbage collected by calling `close()` when it's no longer needed. Otherwise threads and connections are kept alive. It's possible to leak heavy resources in WebRTC!*

RTCSessionDescription objects are blobs that conform to the Session Description Protocol, SDP. Serialized, an SDP object looks like this:

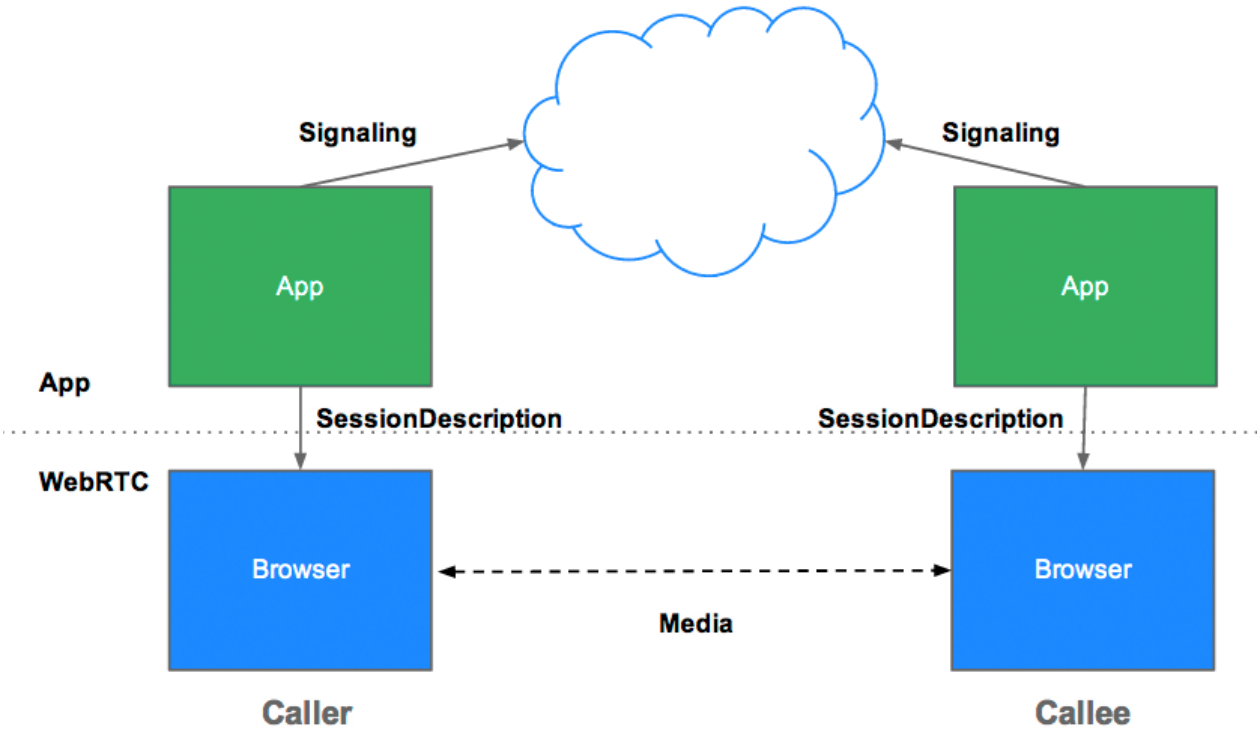
```
v=0
o=- 3883943731 1 IN IP4 127.0.0.1
s=
t=0 0
a=group:BUNDLE audio video
m=audio 1 RTP/SAVPF 103 104 0 8 106 105 13 126

// ...

a=ssrc:2223794119 label:H4fjnMzxy3dPIgQ7HxuCTLb4wLLLeRHnFxh810
```

The acquisition and exchange of network and media information can be done simultaneously, but both processes must have completed before audio and video streaming between peers can begin.

The offer/answer architecture described above is called JSEP, JavaScript Session Establishment Protocol. (There's an excellent animation explaining the process of signaling and streaming in Ericsson's demo video for its first WebRTC implementation.)



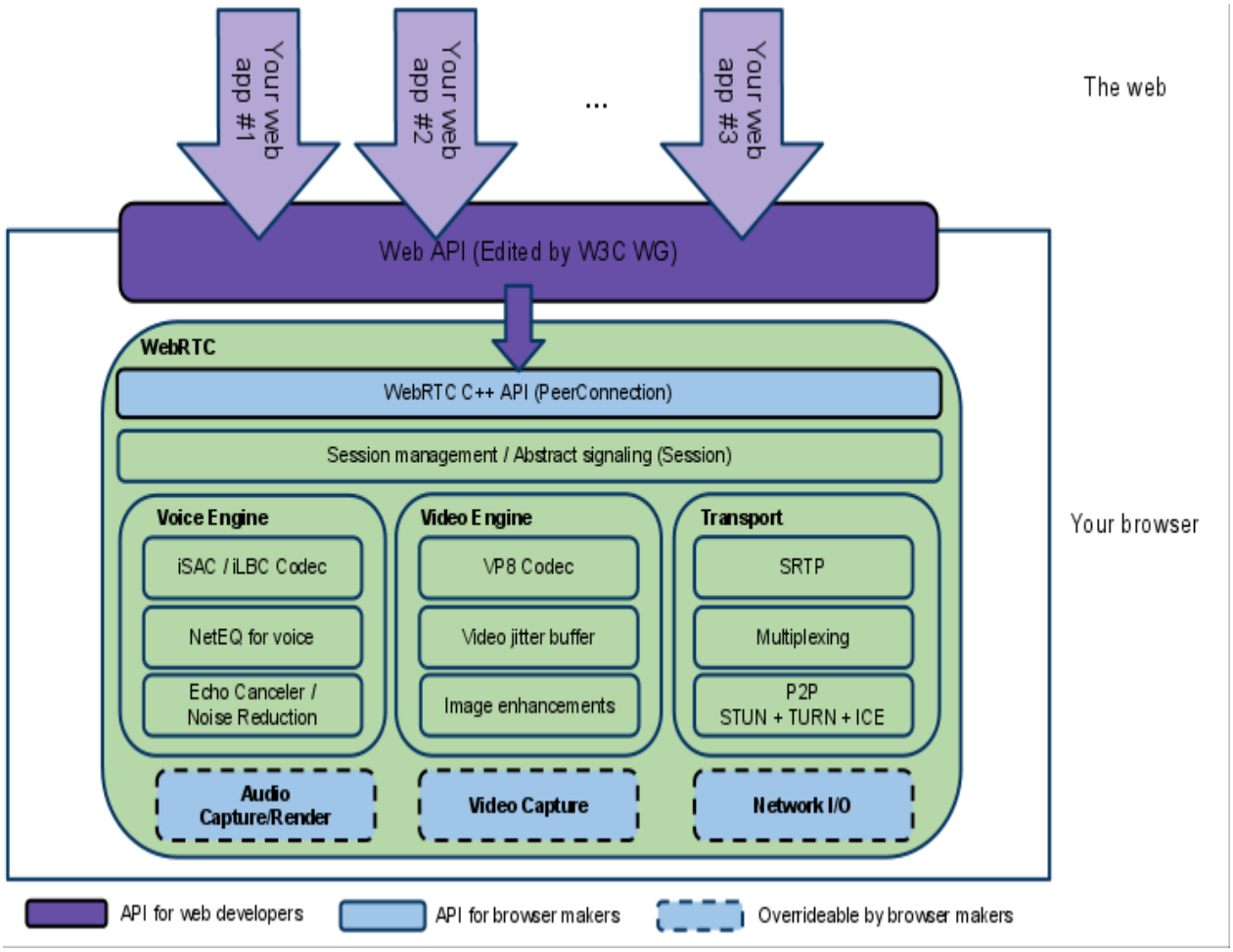
JSEP architecture

Once the signaling process has completed successfully, data can be streamed directly peer to peer, between the caller and callee—or if that fails, via an intermediary relay server (more about that below). Streaming is the job of RTCPeerConnection.

RTCPeerConnection

RTCPeerConnection is the WebRTC component that handles stable and efficient communication of streaming data between peers.

Below is a WebRTC architecture diagram showing the role of RTCPeerConnection. As you will notice, the green parts are complex!



WebRTC architecture (from webrtc.org)

From a JavaScript perspective, the main thing to understand from this diagram is that RTCPeerConnection shields web developers from the myriad complexities that lurk beneath. The codecs and protocols used by WebRTC do a huge amount of work to make real-time communication possible, even over unreliable networks:

- packet loss concealment
- echo cancellation
- bandwidth adaptivity
- dynamic jitter buffering
- automatic gain control
- noise reduction and suppression
- image 'cleaning'.

The W3C code above shows a simplified example of WebRTC from a signaling perspective. Below are walkthroughs of two working WebRTC applications: the first is a simple example to demonstrate RTCPeerConnection; the second is a fully operational video chat client.

RTCPeerConnection without servers

The code below is taken from the 'single page' WebRTC demo at webrtc.github.io/samples/src/content/peerconnection/pc1, which has local *and* remote RTCPeerConnection (and local and remote video) on one web page. This doesn't constitute anything very useful—caller and callee are on the same page—but it does make the workings of the RTCPeerConnection API a little clearer, since the RTCPeerConnection objects on the page can exchange data and messages directly without having to use intermediary signaling mechanisms.

In this example, `pc1` represents the local peer (caller) and `pc2` represents the remote peer (callee).

Caller

1. Create a new RTCPeerConnection and add the stream from `getUserMedia()`:

```
// servers is an optional config file (see TURN and STUN
discussion below)
pc1 = new RTCPeerConnection(servers);
// ...
localStream.getTracks().forEach((track) => {
  pc1.addTrack(track, localStream);
});
```

2. Create an offer and set it as the local description for `pc1` and as the remote description for `pc2`. This can be done directly in the code without using signaling, because both caller and callee are on the same page:

```
pc1.setLocalDescription(desc).then(() => {
      onSetLocalSuccess(pc1);
    },
    onSetSessionDescriptionError
  );
  trace('pc2 setRemoteDescription start');
  pc2.setRemoteDescription(desc).then(() => {
      onSetRemoteSuccess(pc2);
    },
    onSetSessionDescriptionError
  );
```

Callee

1. Create `pc2` and, when the stream from `pc1` is added, display it in a video element:

```
pc2 = new RTCPeerConnection(servers);
pc2.ontrack = gotRemoteStream;
//...
```

```
function gotRemoteStream(e){
  vid2.srcObject = e.stream;
}
```

RTCPeerConnection plus servers

In the real world, WebRTC needs servers, however simple, so the following can happen:

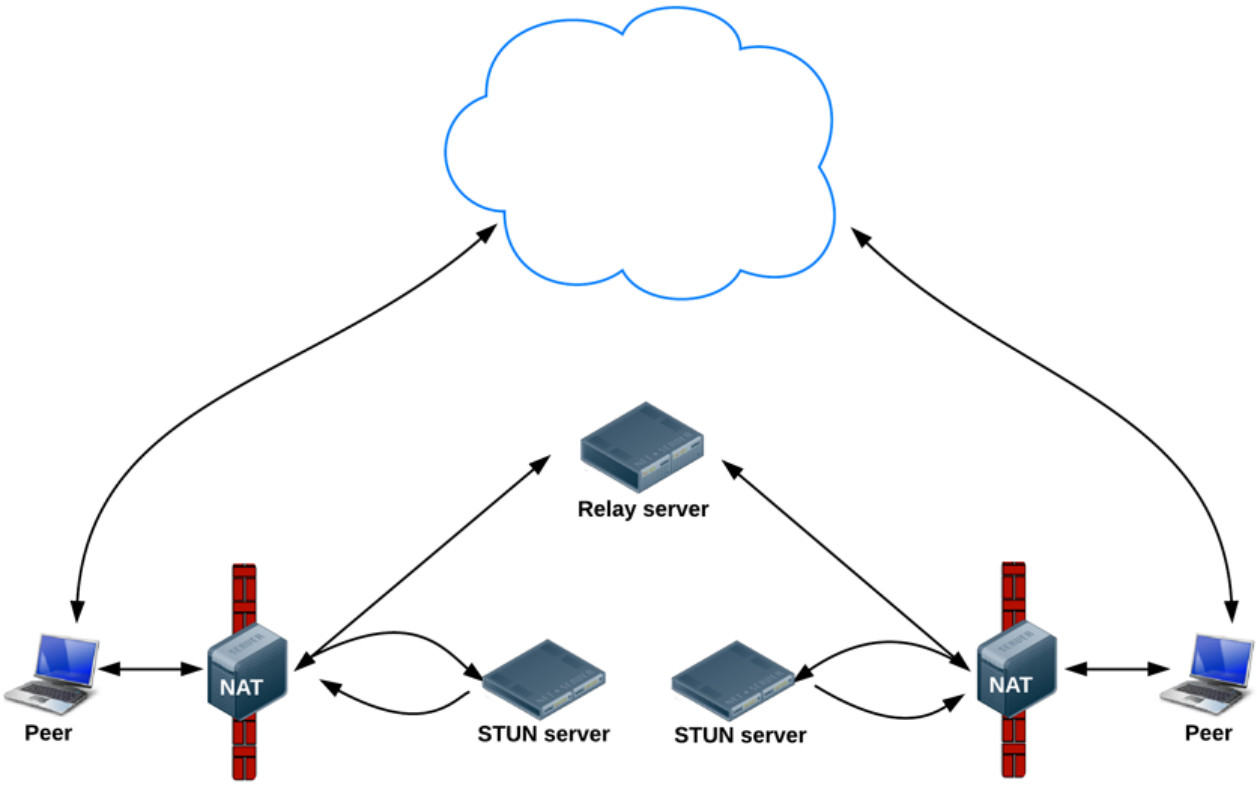
- Users discover each other and exchange 'real world' details such as names.
- WebRTC client applications (peers) exchange network information.
- Peers exchange data about media such as video format and resolution.
- WebRTC client applications traverse NAT gateways and firewalls.

In other words, WebRTC needs four types of server-side functionality:

- User discovery and communication.
- Signaling.
- NAT/firewall traversal.
- Relay servers in case peer-to-peer communication fails.

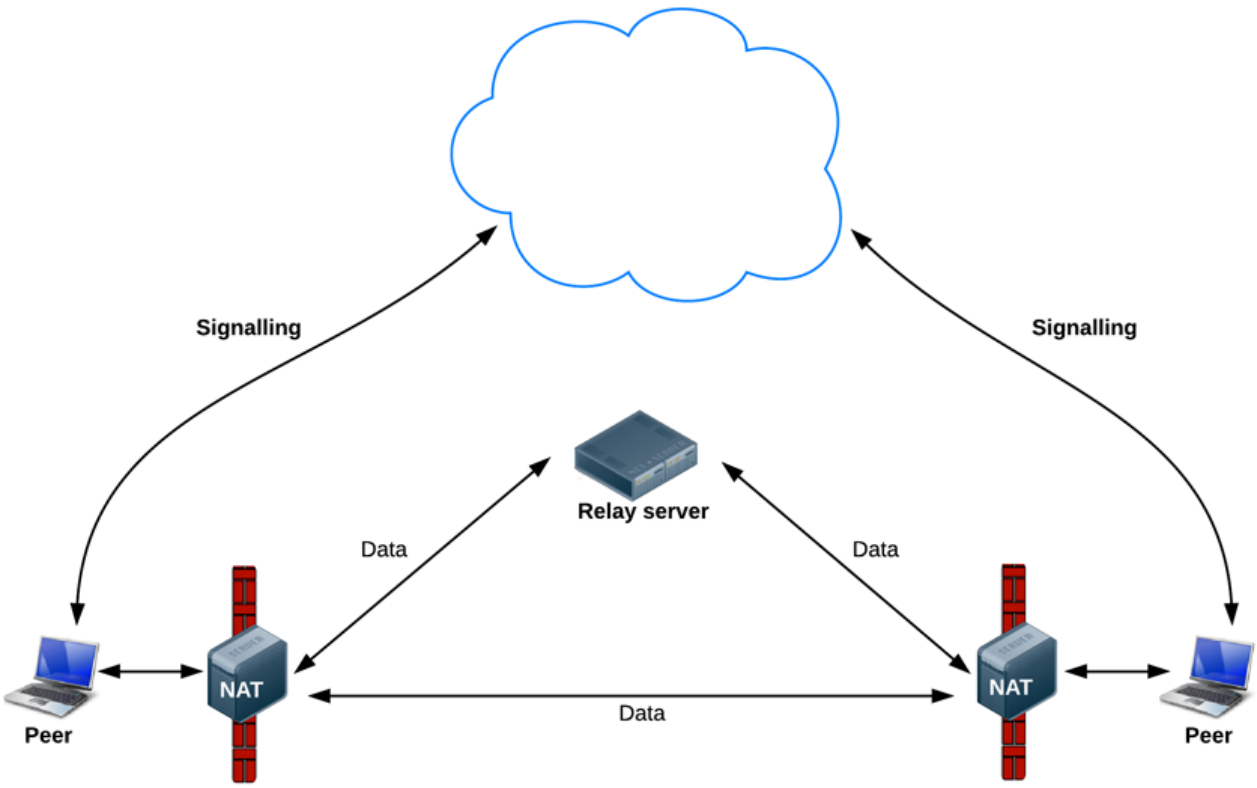
NAT traversal, peer-to-peer networking, and the requirements for building a server app for user discovery and signaling, are beyond the scope of this article. Suffice to say that the STUN protocol and its extension TURN are used by the ICE framework to enable RTCPeerConnection to cope with NAT traversal and other network vagaries.

ICE is a framework for connecting peers, such as two video chat clients. Initially, ICE tries to connect peers *directly*, with the lowest possible latency, via UDP. In this process, STUN servers have a single task: to enable a peer behind a NAT to find out its public address and port. (You can find out more about STUN and TURN from the HTML5 Rocks article WebRTC in the real world.)



Finding connection candidates

If UDP fails, ICE tries TCP. If direct connection fails—in particular, because of enterprise NAT traversal and firewalls—ICE uses an intermediary (relay) TURN server. In other words, ICE will first use STUN with UDP to directly connect peers and, if that fails, will fall back to a TURN relay server. The expression 'finding candidates' refers to the process of finding network interfaces and ports.



WebRTC data pathways

WebRTC engineer Justin Uberti provides more information about ICE, STUN and TURN in the 2013 Google I/O WebRTC presentation. (The presentation slides give examples of TURN and STUN server implementations.)

A simple video chat client

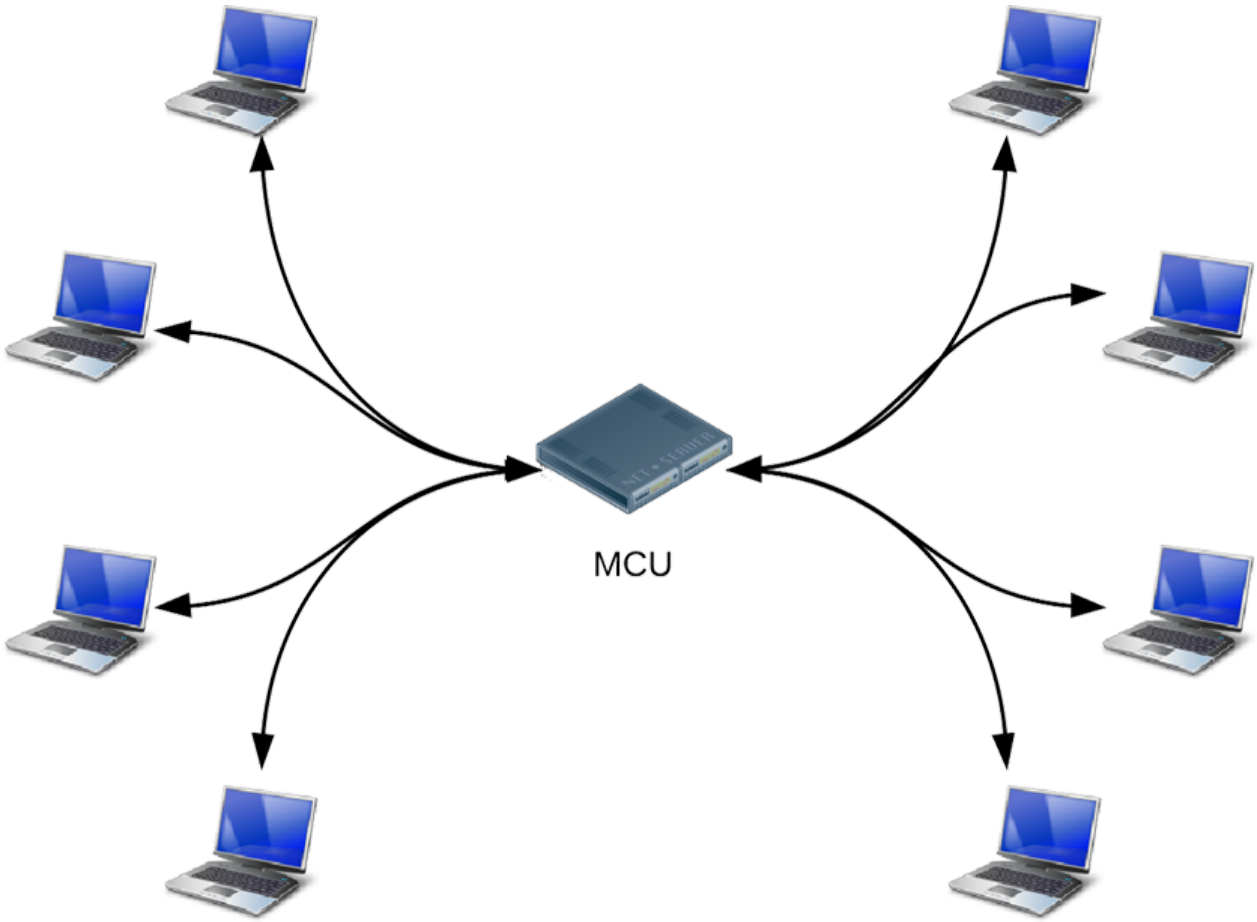
A good place to try out WebRTC, complete with signaling and NAT/firewall traversal using a STUN server, is the video chat demo at appr.tc. This app uses adapter.js, a shim to insulate apps from spec changes and prefix differences. For full interop information, see webrtc.org/web-apis/interop.

The code is deliberately verbose in its logging: check the console to understand the order of events. Below we give a detailed walk-through of the code.

If you find this somewhat baffling, you may prefer our WebRTC codelab. This step-by-step guide explains how to build a complete video chat application, including a simple signaling server running on a Node server.

Network topologies

WebRTC as currently implemented only supports one-to-one communication, but could be used in more complex network scenarios: for example, with multiple peers each communicating each other directly, peer-to-peer, or via a Multipoint Control Unit (MCU), a server that can handle large numbers of participants and do selective stream forwarding, and mixing or recording of audio and video:



Multipoint Control Unit topology example

Many existing WebRTC apps only demonstrate communication between web browsers, but gateway servers can enable a WebRTC app running on a browser to interact with devices such as telephones (aka PSTN) and with VOIP systems. In May 2012, Doubango Telecom open-sourced the sipml5 SIP client, built with WebRTC and WebSocket which (among other potential uses) enables video calls between browsers and apps running on iOS or Android. At Google I/O, Tethr and Tropo demonstrated a framework for disaster communications 'in a briefcase', using an OpenBTS cell to enable communications between feature phones and computers via WebRTC. Telephone communication without a carrier!

Tethr/Tropo: disaster communications in a briefcase

RTCDataChannel

As well as audio and video, WebRTC supports real-time communication for other types of data.

The RTCDataChannel API enables peer-to-peer exchange of arbitrary data, with low latency and high throughput. There are several simple 'single page' demos at webrtc.github.io/samples/#datachannel and our WebRTC codelab shows how to build a simple file transfer application.

There are many potential use cases for the API, including:

- Gaming
- Remote desktop applications
- Real-time text chat
- File transfer
- Decentralized networks

The API has several features to make the most of RTCPeerConnection and enable powerful and flexible peer-to-peer communication:

- Leveraging of RTCPeerConnection session setup.
- Multiple simultaneous channels, with prioritization.

- Reliable and unreliable delivery semantics.
- Built-in security (DTLS) and congestion control.
- Ability to use with or without audio or video.

The syntax is deliberately similar to WebSocket, with a `send()` method and a `message` event:

```
const localConnection = new RTCPeerConnection(servers);
const remoteConnection = new RTCPeerConnection(servers);
const sendChannel =
  localConnection.createDataChannel('sendDataChannel');

// ...

remoteConnection.ondatachannel = (event) => {
  receiveChannel = event.channel;
  receiveChannel.onmessage = onReceiveMessage;
  receiveChannel.onopen = onReceiveChannelStateChange;
  receiveChannel.onclose = onReceiveChannelStateChange;
};

function onReceiveMessage(event) {
  document.querySelector("textarea#send").value = event.data;
}

document.querySelector("button#send").onclick = () => {
  var data = document.querySelector("textarea#send").value;
  sendChannel.send(data);
};
```

Communication occurs directly between browsers, so RTCDataChannel can be much faster than WebSocket even if a relay (TURN) server is required when 'hole punching' to cope with firewalls and NATs fails.

RTCDataChannel is available in Chrome, Safari, Firefox, Opera and Samsung Internet. The Cube Slam game uses the API to communicate game state: play a friend or play the bear! The innovative platform Sharefest enabled file sharing via RTCDataChannel, and peerCDN offered a glimpse of how WebRTC could enable peer-to-peer content distribution.

For more information about RTCDataChannel, take a look at the IETF's draft protocol spec.

Security

There are several ways a real-time communication application or plugin might compromise security. For example:

- Unencrypted media or data might be intercepted en route between browsers, or between a browser and a server.
- An application might record and distribute video or audio without the user knowing.
- Malware or viruses might be installed alongside an apparently innocuous plugin or application.

WebRTC has several features to avoid these problems:

- WebRTC implementations use secure protocols such as DTLS and SRTP.
- Encryption is mandatory for all WebRTC components, including signaling mechanisms.
- WebRTC is not a plugin: its components run in the browser sandbox and not in a separate process, components do not require separate installation, and are updated whenever the browser is updated.
- Camera and microphone access must be granted explicitly and, when the camera or microphone are running, this is clearly shown by the user interface.

A full discussion of security for streaming media is out of scope for this article. For more information, see the WebRTC Security Architecture proposed by the IETF.

In conclusion

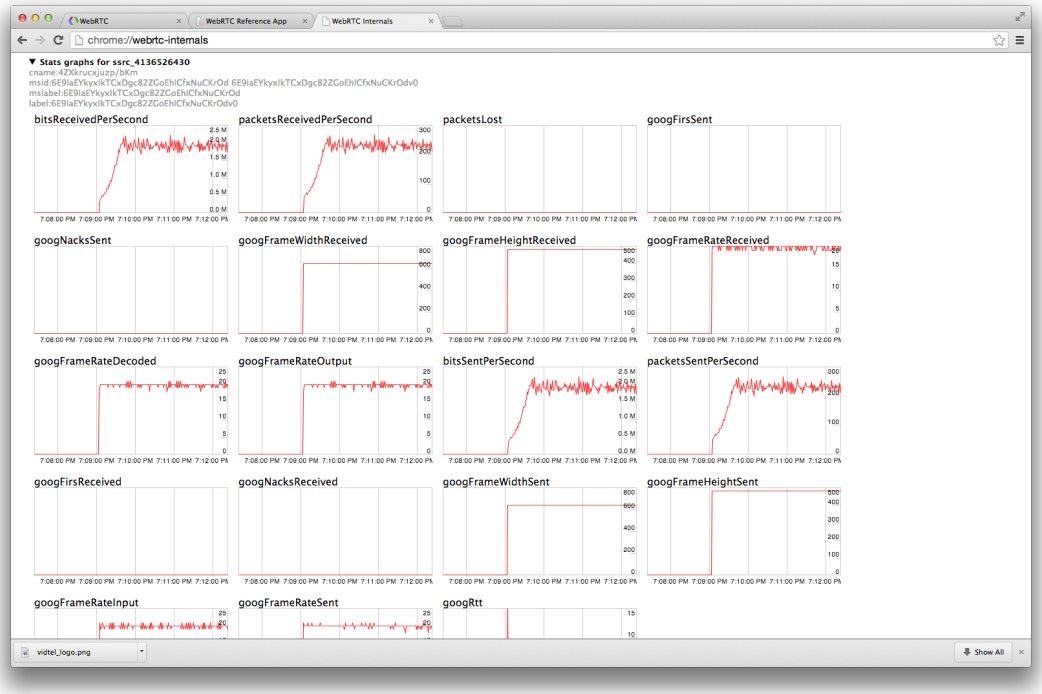
The APIs and standards of WebRTC can democratize and decentralize tools for content creation and communication—for telephony, gaming, video production, music making, news gathering and many other applications.

Technology doesn't get much more disruptive than this.

We look forward to what JavaScript developers make of WebRTC as it becomes widely implemented. As blogger Phil Edholm put it, 'Potentially, WebRTC and HTML5 could enable the same transformation for real-time communications that the original browser did for information.'

Developer tools

- WebRTC stats for an ongoing session can be found at:
    - **chrome://webrtc-internals** page in Chrome
    - **opera://webrtc-internals** page in Opera
    - **about:webrtc** page in Firefox
    - Example:



chrome://webrtc-internals screenshot

- Cross browser interop notes
- adapter.js is a JavaScript shim for WebRTC, maintained by Google with help from the WebRTC community, that abstracts vendor prefixes, browser differences and spec changes
- To learn more about WebRTC signaling processes, check the appr.tc log output to the console
- If it's all too much, you may prefer to use a WebRTC framework or even a complete WebRTC service
- Bug reports and feature requests are always appreciated:
    - WebRTC bugs
    - Chrome bugs
    - Opera bugs
    - Firefox bugs
    - WebRTC demo bugs
    - Adapter.js bugs

Learn more

- WebRTC presentation at Google I/O 2013 (the slides are at io13webrtc.appspot.com)
- Justin Uberti's WebRTC session at Google I/O 2012
- Alan B. Johnston and Daniel C. Burnett maintain a WebRTC book, now in its third edition in print and eBook formats: webrtcbook.com
- webrtc.org is home to all things WebRTC: demos, documentation and discussion

- webrtc.org demo page: links to demos
- discuss-webrtc: Google Group for technical WebRTC discussion
- +webrtc
- @webrtc
- Google Developers Google Talk documentation, which gives more information about NAT traversal, STUN, relay servers and candidate gathering
- WebRTC on GitHub
- Stack Overflow is a good place to look for answers and ask questions about WebRTC

Standards and protocols

- The WebRTC W3C Editor's Draft
- W3C Editor's Draft: Media Capture and Streams (aka getUserMedia)
- IETF Working Group Charter
- IETF WebRTC Data Channel Protocol Draft
- IETF JSEP Draft
- IETF proposed standard for ICE
- IETF RTCWEB Working Group Internet-Draft: Web Real-Time Communication Use-cases and Requirements

WebRTC support summary

MediaStream and getUserMedia

- Chrome desktop 18.0.1008+; Chrome for Android 29+
- Opera 18+; Opera for Android 20+
- Opera 12, Opera Mobile 12 (based on the Presto engine)
- Firefox 17+
- Microsoft Edge 16+
- Safari 11.2+ on iOS; 11.1+ on Mac
- UC 11.8+ on Android
- Samsung Internet 4+

RTCPeerConnection

- Chrome desktop 20+; Chrome for Android 29+ (flagless)
- Opera 18+ (on by default); Opera for Android 20+ (on by default)
- Firefox 22+ (on by default)
- Microsoft Edge 16+
- Safari 11.2+ on iOS; 11.1+ on Mac
- Samsung Internet 4+

RTCDataChannel

- Experimental version in Chrome 25, more stable (and with Firefox interoperability) in Chrome 26+; Chrome for Android 29+
- Stable version (and with Firefox interoperability) in Opera 18+; Opera for Android 20+
- Firefox 22+ (on by default)

For more detailed information about cross-platform support for APIs such as getUserMedia and RTCPeerConnection, see caniuse.com and chromestatus.com.

Native APIs for RTCPeerConnection are also available: documentation on webrtc.org.