Get started with WebRTC

Sam Dutton
(https://twitter.com/sw12) (https://github.com/samdutton) (https://glitch.com/@samdutton) (https://techhub.social/@samdutton) (https://samdutton.com)

> 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 communicate on a common platform. Imagine it was easy to add video chat and peer-to-peer data sharing to your web app. That's the vision of WebRTC.

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

1. Open appr.tc (https://appr.tc) in your browser.
2. Click **Join** to join a chat room and let the app use your webcam.
3. Open the URL displayed at the end 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 only want code?

- To get an overview of WebRTC, watch the following Google I/O video or view these slides (https://io13webrtc.appspot.com/):

  Real-time communication with WebRTC: Google I/O 2013

Genius Sports Ex. 1038
p. 1

- If you haven't used the `getUserMedia` API, see Capture audio and video in HTML5 (https://www.html5rocks.com/en/tutorials/getusermedia/intro) and simpl.info getUserMedia (https://www.simpl.info/getusermedia).
- To learn about the `RTCPeerConnection` API, see the following example and 'simpl.info RTCPeerConnection' (https://simpl.info/rtcpeerconnection).
- To learn how WebRTC uses servers for signaling, and firewall and NAT traversal, see the code and console logs from appr.tc (https://appr.tc).
- Can't wait and just want to try WebRTC right now? Try some of the more-than 20 demos (https://webrtc.github.io/samples) that exercise the WebRTC JavaScript APIs.
- Having trouble with your machine and WebRTC? Visit the WebRTC Troubleshooter (https://test.webrtc.org).

Alternatively, jump straight into the WebRTC codelab (https://codelabs.developers.google.com/codelabs/webrtc-web/), 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 through voice and video: real-time communication or RTC for short. RTC should be as natural in a web app as entering text in a text input. Without it, you're limited in your 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 uses Talk (as did Gmail). Google bought GIPS, a company that developed many components required for RTC, such as codecs and echo cancellation techniques. Google open sourced the technologies developed by GIPS and engaged with relevant standards bodies at the Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) to ensure industry consensus. In May 2011, Ericsson built the first implementation of WebRTC (https://labs.ericsson.com/developer-community/blog/beyond-html5-peer-peer-conversational-video).

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. These included Skype, Facebook, and 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, such as Google Meet. WebRTC has also been integrated with WebKitGTK+ (https://labs.ericsson.com/developer-community/blog/beyond-html5-conversational-voice-and-video-implemented-webkit-gtk) and Qt native apps.

WebRTC implements these three APIs: - `MediaStream` (also known as `getUserMedia`) - `RTCPeerConnection` - `RTCDataChannel`

The APIs are defined in these two specs:

- WebRTC (https://w3c.github.io/webrtc-pc/)
- `getUserMedia` (https://www.w3.org/TR/mediacapture-streams)

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

Genius Sports Ex. 1038
p. 3

`getUserMedia`: For demos and code, see WebRTC samples (https://webrtc.github.io/samples) or try Chris Wilson's amazing examples (https://webaudiodemos.appspot.com) that use `getUserMedia` as input for web audio.

`RTCPeerConnection`: For a simple demo and a fully functional video-chat app, see WebRTC samples Peer connection (https://webrtc.github.io/samples/src/content/peerconnection/pc1/) and appr.tc (https://appr.tc), respectively. This app uses adapter.js (https://github.com/webrtc/adapter), a JavaScript shim maintained by Google with help from the WebRTC community (https://github.com/webrtc/adapter/graphs/contributors), to abstract away browser differences and spec changes.

`RTCDataChannel`: To see this in action, see WebRTC samples (https://webrtc.github.io/samples/) to check out one of the data-channel demos.

The WebRTC codelab (https://codelabs.developers.google.com/codelabs/webrtc-web/#0) shows how to use all three APIs to build a simple app for video chat and file sharing.

Your first WebRTC

WebRTC apps need to do several things:

- Get streaming audio, video, or other data.
- Get network information, such as IP addresses and ports, and exchange it with other WebRTC clients (known as **peers**) to enable connection, even through NATs (https://en.wikipedia.org/wiki/NAT_traversal) 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` (https://dvcs.w3.org/hg/audio/raw-file/tip/streams/StreamProcessing.html) gets access to data streams, such as from the user's camera and microphone.
- `RTCPeerConnection` (https://dev.w3.org/2011/webrtc/editor/webrtc.html#rtcpeerconnection-interface) enables audio or video calling with facilities for encryption and bandwidth management.
- `RTCDataChannel` (https://dev.w3.org/2011/webrtc/editor/webrtc.html#rtcdatachannel) enables peer-to-peer communication of generic data.

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

`MediaStream` API (also known as `getUserMedia` API)

The `MediaStream` API (https://dev.w3.org/2011/webrtc/editor/getusermedia.html) 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 (https://www.html5rocks.com/en/tutorials/track/basics/).)

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

1. In your browser, navigate to WebRTC samples `getUserMedia` (https://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 `MediaStreamTrack`s is returned by the `getAudioTracks()` and `getVideoTracks()` methods.

For the `getUserMedia` (https://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'`), 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, such as a chat app that gets streams from the front camera, rear camera, microphone, and an app sharing its screen.

A `MediaStream` can be attached to a video element by setting the `srcObject` attribute (https://developer.mozilla.org/docs/Web/API/HTMLMediaElement/srcObject). Previously, this was done by setting the `src` attribute to an object URL created with `URL.createObjectURL()`, but this has been deprecated (https://developer.mozilla.org/docs/Web/API/URL/createObjectURL).

Note: 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 (https://developer.chrome.com/blog/live-web-audio-input-enabled):

Genius Sports Ex. 1038
p. 5

```
// 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. For example, play the sound.
  gainNode.connect(audioContext.destination);
});
```

Chromium-based apps and extensions can also incorporate `getUserMedia`. Adding `audioCapture` and/or `videoCapture` permissions (https://developer.chrome.com/extensions/permission_warnings) to the manifest enables permission to be requested and granted only once upon 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 (https://dev.chromium.org/user-experience/infobars). HTTP access for `getUserMedia()` was deprecated by Chrome at the end of 2015 due to it being classified as a Powerful

feature (https://sites.google.com/a/chromium.org/dev/Home/chromium-security/deprecating-powerful-features-on-insecure-origins).

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

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

- Webcam Toy (https://webcamtoy.com/app/) is a photobooth app that uses WebGL to add weird and wonderful effects to photos that can be shared or saved locally.
- FaceKat (https://www.auduno.com/2012/06/15/head-tracking-with-webrtc/) is a face-tracking game built with headtrackr.js (https://github.com/auduno/headtrackr).
- ASCII Camera (https://idevelop.ro/ascii-camera/) uses the Canvas API to generate ASCII images.

Genius Sports Ex. 1038
p. 7

gUM ASCII art!

Constraints

Constraints (https://tools.ietf.org/html/draft-alvestrand-constraints-resolution-00#page-4) can be used to set values for video resolution for `getUserMedia()`. This also allows support for other constraints (https://w3c.github.io/mediacapture-main/getusermedia.html#the-model-sources-sinks-constraints-and-settings), such as aspect ratio; facing mode (front or back camera); frame rate, height and width; and an `applyConstraints()` (https://w3c.github.io/mediacapture-main/getusermedia.html#dom-mediastreamtrack-applyconstraints) method.

For an example, see WebRTC samples `getUserMedia`: select resolution (https://webrtc.github.io/samples/src/content/getusermedia/resolution).

Gotcha: `getUserMedia` constraints may affect the available configurations of a shared resource. For example, if a camera was opened in 640 x 480 mode by one tab, another tab will not be able to use constraints to open it in a higher-resolution mode because 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 640 x 480 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, see WebRTC samples `getUserMedia`: select resolution (https://webrtc.github.io/samples/src/content/getusermedia/resolution/) for a demo.

Screen and tab capture

Chrome apps also make it possible to share a live video of a single browser tab or the entire desktop through `chrome.tabCapture` (https://developer.chrome.com/dev/extensions/tabCapture) and `chrome.desktopCapture` (https://developer.chrome.com/extensions/desktopCapture) APIs. (For a demo and more information, see Screensharing with WebRTC (https://developer.chrome.com/blog/screensharing-with-webrtc). The article is a few years old, but it's still interesting.)

It's also possible to use screen capture as a `MediaStream` source in Chrome using the experimental `chromeMediaSource` constraint. Note that screen capture requires HTTPS and should only be used for development due to it being enabled through a command-line flag as explained in this post (https://groups.google.com/forum/#!msg/discuss-webrtc/TPQVKZnsF5g/Hlpy8kqaLnEJ).

Signaling: Session control, network, and media information

WebRTC uses `RTCPeerConnection` to communicate streaming data between browsers (also known as 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 (https://appr.tc) example uses XHR and the Channel API as the signaling mechanism. The codelab (https://codelabs.developers.google.com/codelabs/webrtc-web/#0) uses Socket.io (https://socket.io) running on a Node server (https://nodejs.org/).

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 your computer's IP address and port?
- Media capabilities: what codecs and resolutions can be handled by your browser and the browser it wants to communicate with?

The exchange of information through 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 (https://w3c.github.io/webrtc-pc/#simple-peer-to-peer-example), 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: 'stun: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 you get an offer, you 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, 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, which runs when network candidates become available.
2. Alice sends serialized candidate data to Bob through whatever signaling channel they are using, such as WebSocket or some other mechanism.

3. When Bob gets a candidate message from Alice, he calls `addIceCandidate` to add the candidate to the remote peer description.

WebRTC clients (also known as **peers**, or Alice and Bob in this example) 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 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 through their signaling channel. Note that `RTCPeerConnection` won't start gathering candidates until `setLocalDescription()` is called. This is codified in the JSEP IETF draft (https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-4.2.4).
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!

Note: 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 (https://en.wikipedia.org/wiki/Session_Description_Protocol), SDP. Serialized, an SDP object looks like this:

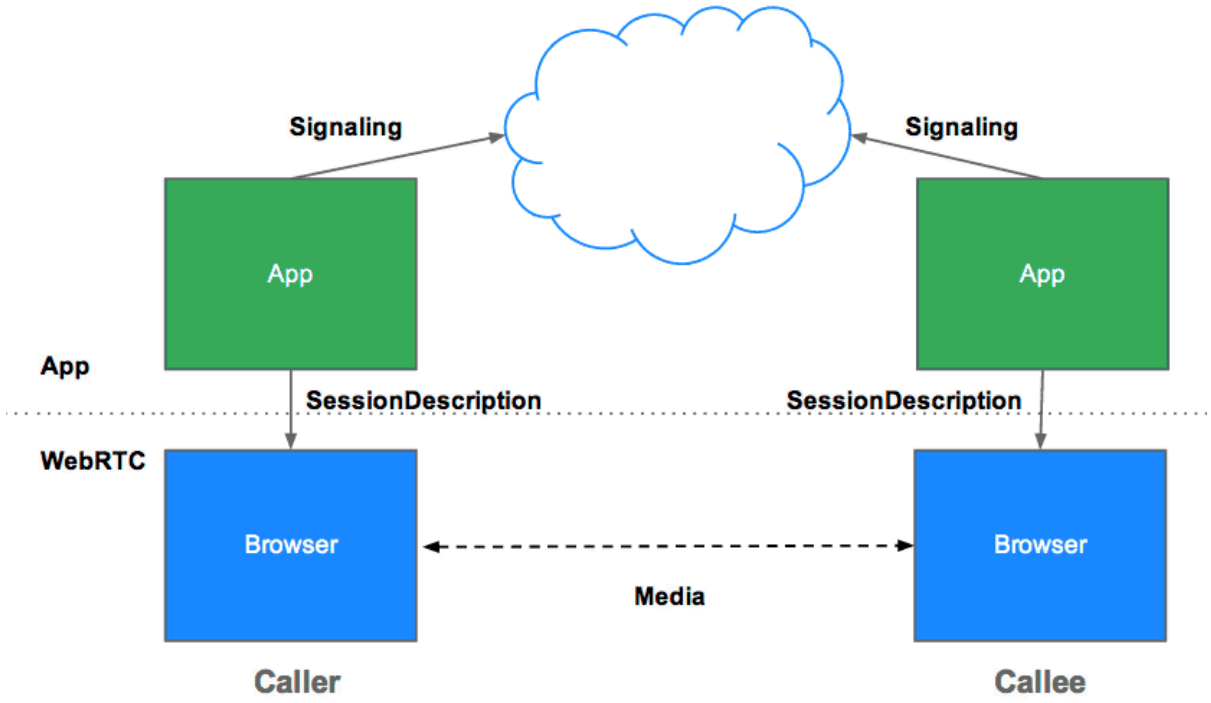
```
v=0
o=- 3883943731 1 IN IP4 127.0.0.1
s=
t=0 0
a=group:BUNDLE audio video
m=audio 1 RTP/SAVPF 103 104 0 8 106 105 13 126

// ...

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 previously described is called JavaScript Session Establishment Protocol (https://rtcweb-wg.github.io/jsep/), or JSEP. (There's an excellent animation explaining the process of signaling and streaming in Ericsson's demo video (https://www.ericsson.com/research-blog/context-aware-communication/beyond-html5-peer-peer-conversational-video/) for its first WebRTC implementation.)



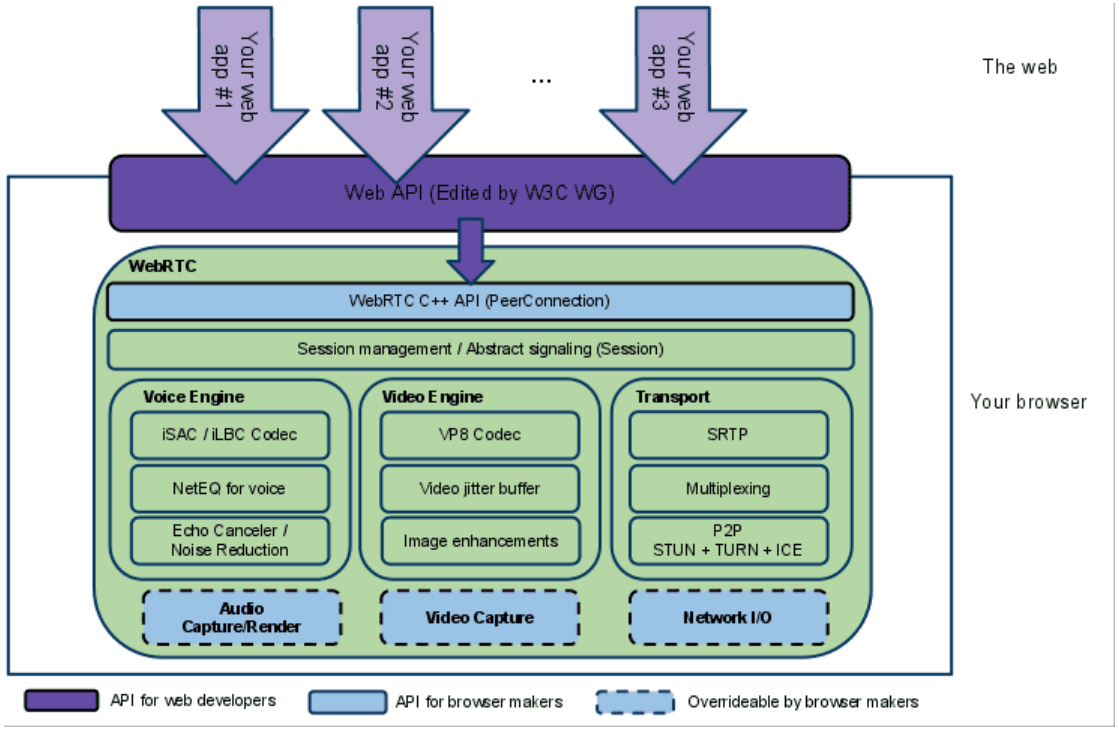
JSEP architecture

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

RTCPeerConnection

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

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



WebRTC architecture (from webrtc.org (https://webrtc.github.io/webrtc-org/architecture/)*)*

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 previous W3C code shows a simplified example of WebRTC from a signaling perspective. The following are walkthroughs of two working WebRTC apps. The first is a simple example to demonstrate `RTCPeerConnection` and the second is a fully operational video chat client.

RTCPeerConnection without servers

The following code is taken from WebRTC samples Peer connection (https://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 because 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()`: ```js // Servers is an optional configuration file. (See TURN and STUN discussion later.) pc1 = new RTCPeerConnection(servers); // ... localStream.getTracks().forEach((track) => { pc1.addTrack(track, localStream); });
1. 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: `js 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: `js pc2 = new RTCPeerConnection(servers); pc2.ontrack = gotRemoteStream; //... function gotRemoteStream(e){ vid2.srcObject = e.stream; }`

`RTCPeerConnection` API 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 apps (peers) exchange network information.
- Peers exchange data about media, such as video format and resolution.

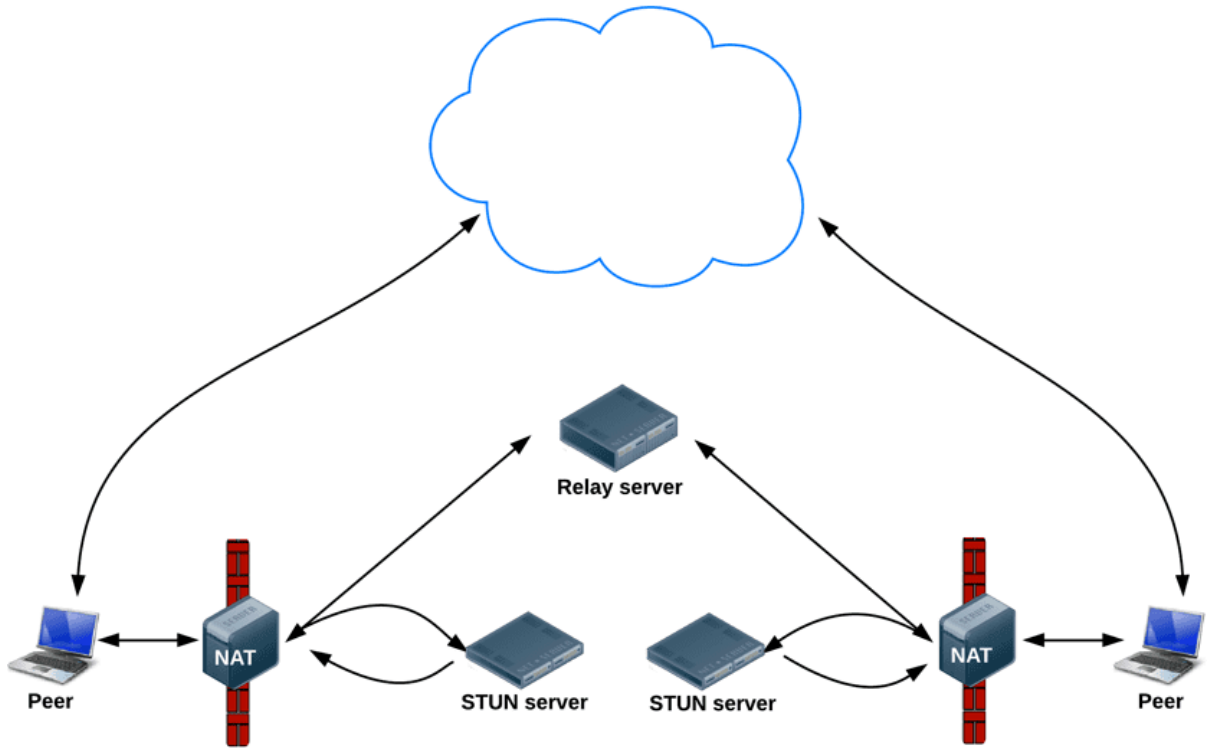
- WebRTC client apps traverse NAT gateways (https://en.wikipedia.org/wiki/NAT_traversal) and firewalls.

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 (https://en.wikipedia.org/wiki/STUN) protocol and its extension, TURN (https://en.wikipedia.org/wiki/Traversal_Using_Relay_NAT), are used by the ICE (https://en.wikipedia.org/wiki/Interactive_Connectivity_Establishment) framework to enable `RTCPeerConnection` to cope with NAT traversal and other network vagaries.

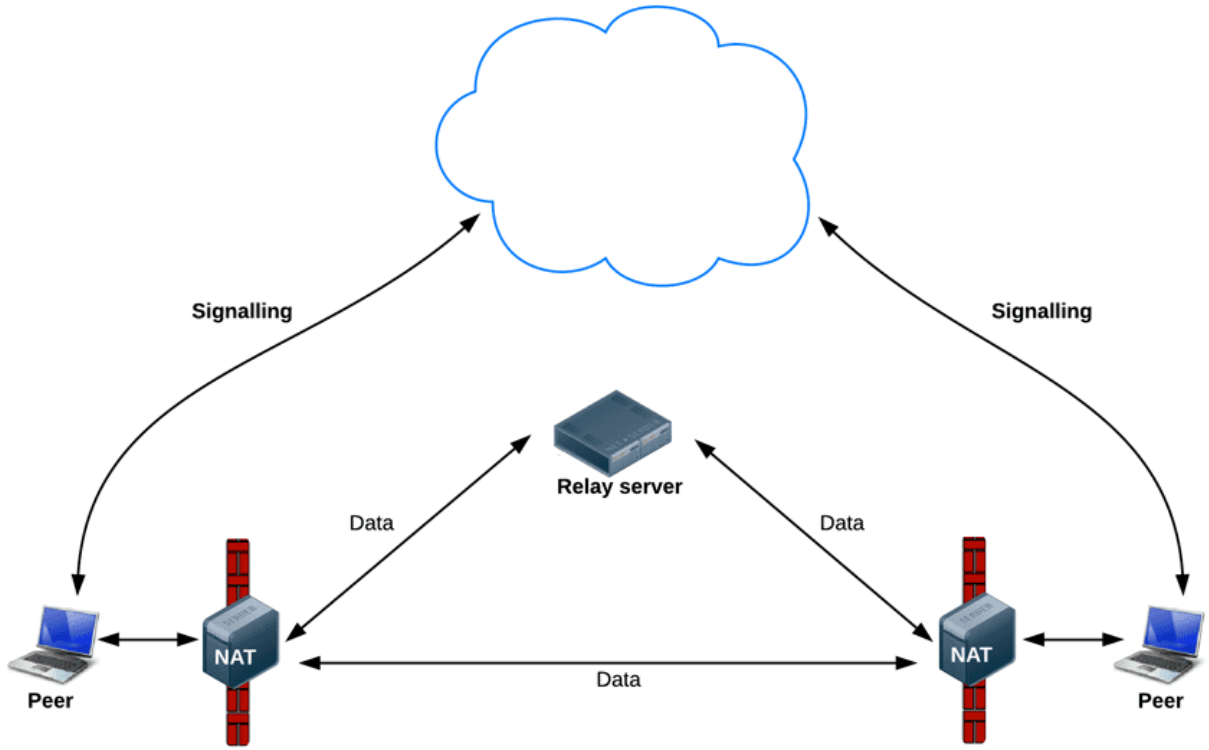
ICE is a framework for connecting peers, such as two video chat clients. Initially, ICE tries to connect peers **directly** with the lowest possible latency through UDP. In this process, STUN servers have a single task: to enable a peer behind a NAT to find out its public address and port. (For more information about STUN and TURN, see Build the backend services needed for a WebRTC app (https://www.html5rocks.com/tutorials/webrtc/infrastructure/).)



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 first uses STUN with UDP to directly connect peers and, if that fails, falls back to a TURN relay server. The expression **finding candidates** refers to the process of finding network interfaces and ports.

Genius Sports Ex. 1038
p. 17



WebRTC data pathways

WebRTC engineer Justin Uberti provides more information about ICE, STUN, and TURN in the 2013 Google I/O WebRTC presentation (https://www.youtube.com/watch?v=p2HzZkd2A40&t=21m12s). (The presentation slides (https://io13webrtc.appspot.com/#52) give examples of TURN and STUN server implementations.)

A simple video-chat client

A good place to try WebRTC, complete with signaling and NAT/firewall traversal using a STUN server, is the video-chat demo at appr.tc (https://appr.tc). This app uses adapter.js (https://github.com/webrtc/adapter), a shim to insulate apps from spec changes and prefix differences.

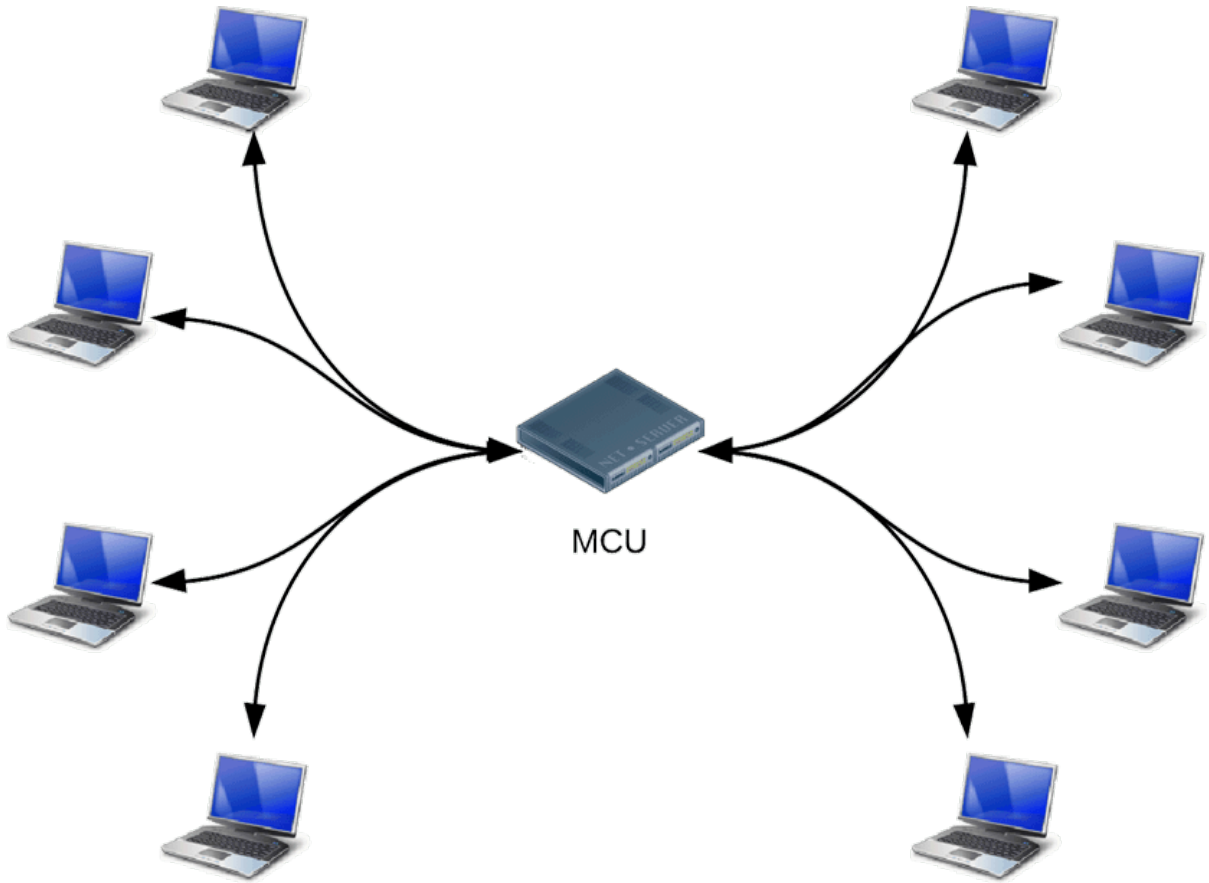
The code is deliberately verbose in its logging. Check the console to understand the order of events. The following is a detailed walkthrough of the code.

Note: If you find this somewhat baffling, you may prefer the WebRTC codelab (https://codelabs.developers.google.com/codelabs/webrtc-web/). This step-by-step guide explains how to build a

complete video-chat app, including a simple signaling server running on a Node server (https://nodejs.org/).

Network topologies

WebRTC, as currently implemented, only supports one-to-one communication, but could be used in more complex network scenarios, such as with multiple peers each communicating with each other directly or through a Multipoint Control Unit (https://en.wikipedia.org/wiki/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 (https://en.wikipedia.org/wiki/Public_switched_telephone_network) (also known as PSTN (https://en.wikipedia.org/wiki/Public_switched_telephone_network)) and with VOIP (https://en.wikipedia.org/wiki/Voice_over_IP) systems. In May 2012, Doubango Telecom open sourced the sipml5 SIP client (https://sipml5.org/) built with WebRTC and WebSocket, which (among other potential uses) enables video calls between browsers and apps running on iOS and Android. At Google I/O, Tethr and Tropo demonstrated a framework for disaster communications (https://tethr.tumblr.com/) **in a briefcase** using an OpenBTS cell (https://en.wikipedia.org/wiki/OpenBTS) to enable communications between feature phones and computers through WebRTC. Telephone communication without a carrier!

Tethr/Tropo: Disaster communications in a briefcase

`RTCDataChannel` API<

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. For single-page demos and to learn how to build a simple file-transfer app, see WebRTC samples (https://webrtc.github.io/samples/#datachannel) and the WebRTC codelab (https://codelabs.developers.google.com/codelabs/webrtc-web/#0), respectively.

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

- Gaming
- Remote desktop apps
- 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 (https://experiments.withgoogle.com/cube-slam) game uses the API to communicate game state. Play a friend or play the bear! The innovative platform Sharefest (https://github.com/Peer5/ShareFest) enabled file sharing through `RTCDataChannel` and peerCDN (https://techcrunch.com/2013/12/17/yahoo-acquires-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 (https://tools.ietf.org/html/draft-jesup-rtcweb-data-protocol-00).

Security

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

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

WebRTC has several features to avoid these problems:

- WebRTC implementations use secure protocols, such as DTLS (https://en.wikipedia.org/wiki/Datagram_Transport_Layer_Security) and SRTP (https://en.wikipedia.org/wiki/Secure_Real-time_Transport_Protocol).
- 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 Proposed WebRTC Security Architecture (https://www.ietf.org/proceedings/82/slides/rtcweb-13.pdf) proposed by the IETF.

In conclusion

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

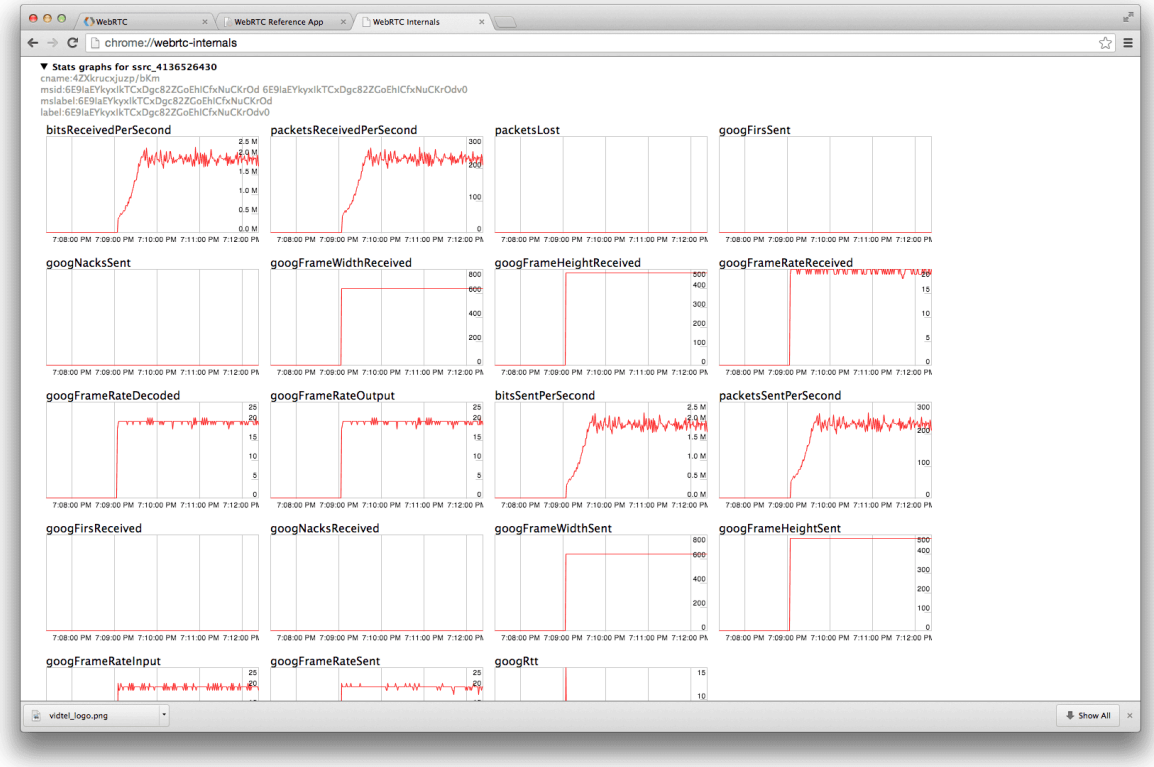
Technology doesn't get much more disruptive (https://en.wikipedia.org/wiki/Disruptive_innovation) than this.

As blogger Phil Edholm put it (https://www.nojitter.com/webrtc-it-game-changer), "Potentially, WebRTC and HTML5 could enable the same transformation for real-time communication that the original browser did for information."

Developer tools

Note: Starting from version 124 (https://developer.chrome.com/blog/new-in-devtools-124#throttling), Chrome DevTools lets you test WebRTC over UDP for open `RTCPeerConnections`. To test your WebRTC applications, create a custom throttling profile (https://developer.chrome.com/docs/devtools/network/reference#throttling-profile) and specify packet-related parameters.

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



chrome://webrtc-internals screenshot

- Cross browser interop notes (https://webrtc.github.io/webrtc-org/web-apis/interop/)
- adapter.js (https://github.com/webrtc/adapter) is a JavaScript shim for WebRTC maintained by Google with help from the WebRTC community (https://github.com/webrtc/adapter/graphs/contributors) that abstracts vendor prefixes, browser differences, and spec changes.
- To learn more about WebRTC signaling processes, check the appr.tc (https://appr.tc) log output to the console.
- If it's all too much, you may prefer to use a WebRTC framework (https://io13webrtc.appspot.com/#69) or even a complete WebRTC service (https://io13webrtc.appspot.com/#72).
- Bug reports and feature requests are always appreciated:
  - WebRTC bugs (https://code.google.com/p/webrtc/issues/entry)
  - Chrome bugs (https://www.crbug.com/new)

- Opera bugs (https://bugs.opera.com/wizard/)
- Firefox bugs (https://bugzilla.mozilla.org/)
- WebRTC demo bugs (https://github.com/webrtc/samples/issues/new)
- Adapter.js bugs (https://github.com/webrtcHacks/adapter/issues/new)

Learn more

- Justin Uberti's WebRTC session at Google I/O 2012 (https://www.youtube.com/watch?v=E8C8ouiXHHk)
- Alan B. Johnston and Daniel C. Burnett maintain a WebRTC book now in its third edition in print and eBook formats at webrtcbook.com (https://www.webrtcbook.com).
- webrtc.org (https://www.webrtc.org/) is home to all things WebRTC, including demos, documentation, and discussion.
- discuss-webrtc (https://groups.google.com/forum/?fromgroups#!forum/discuss-webrtc) is a Google Group for technical WebRTC discussion.
- @webrtc (https://twitter.com/webrtc)
- Google Developers Talk documentation (https://developers.google.com/talk/talk_developers_home) provides more information about NAT traversal, STUN, relay servers, and candidate gathering.
- WebRTC on GitHub (https://github.com/webrtc)
- Stack Overflow (https://stackoverflow.com/questions/tagged/webrtc) is a good place to look for answers and ask questions about WebRTC.

Standards and protocols

- The WebRTC W3C Editor's Draft (https://dev.w3.org/2011/webrtc/editor/webrtc.html)
- W3C Editor's Draft: Media Capture and Streams (https://dev.w3.org/2011/webrtc/editor/getusermedia.html) (also known as `getUserMedia`)
- IETF Working Group Charter (https://tools.ietf.org/wg/rtcweb/charters)
- IETF WebRTC Data Channel Protocol Draft (https://tools.ietf.org/html/draft-jesup-rtcweb-data-protocol-01)
- IETF JSEP Draft (https://tools.ietf.org/html/draft-uberti-rtcweb-jsep-02)
- IETF proposed standard for ICE (https://tools.ietf.org/html/rfc5245)
- IETF RTCWEB Working Group Internet-Draft: Web Real-Time Communication Use-cases and Requirements (https://tools.ietf.org/html/draft-ietf-rtcweb-use-cases-and-requirements-10)

WebRTC support summary

`MediaStream` and `getUserMedia` APIs

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

`RTCPeerConnection` API

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

`RTCDataChannel` API

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

For more detailed information about cross-platform support for APIs, such as `getUserMedia` and `RTCPeerConnection`, see caniuse.com (https://caniuse.com) and Chrome Platform Status (https://chromestatus.com).

Native APIs for `RTCPeerConnection` are also available at documentation on webrtc.org (https://webrtc.googlesource.com/src/+/refs/heads/master/docs/native-code/index.md).

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/), and code samples are licensed under the Apache 2.0 License (https://www.apache.org/licenses/LICENSE-2.0). For details, see the Google Developers Site Policies (https://developers.google.com/site-policies). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2012-07-23 UTC.