Contents

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COLLEGE OF ENGINEERING AND TECHNOLOGY,

BHUBANESWAR

CERTIFICATE

This is to certify that this is a bona fide record of the seminar work titled ***"WebRTC: Plugin-less Real Time Communication for the Web"*** Submitted by ***TARASHISH MISHRA*** of ***6th semester, Computer Science & Engineering***,in the year 2014 in partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science & Engineering of College of Engineering & Technology, Bhubaneswar.

Mr. Amitabh Mohapatra
Seminar Guide

Ms.Pranati Mishra
Seminar Guide

ACKNOWLEDGEMENT

First of all, I would like to express my sincere gratitude to our seminar guides Mr. Amitav Mahapatra and Ms. Pranati Mishra for their encouragement and constant support. I express my heartfelt thanks to the Department of Computer Science and Technology for giving me this opportunity to pursue a topic to this depth. I thank all the lovely people on Mozilla's WebRTC IRC channel (#media on irc.mozilla.org) for all the help to clear up my doubts. And finally, I owe a sincere thanks to all my friends and my family for supporting me and lending me a helping hand whenever I needed one.

Tarashish Mishra
Registration No.: 1101106116
Branch: CSE
6th Semester

ABSTRACT

WebRTC: Plugin-less Real Time Communication for the Web

WebRTC(**Web Real-Time Communication**) is an API definition being drafted by the World Wide Web Consortium (W3C) to enable direct peer to peer video/audio streaming and data sharing between browsers. It is an free and open project being developed collaboratively by Google, Mozilla and Opera to bring real time communication to web browsers via simple JavaScript APIs.

"WebRTC is a new front in the long war for an open and unencumbered web."
– Brendan Eich, inventor of JavaScript

Historically, real time communication(RTC) has been proprietary, corporate and complex. Integrating RTC technology with existing data and services has been difficult and time consuming, particularly on the web. It often requires to download and install various proprietary plugins or software.

WebRTC has implemented open standards for real-time, plugin-free video, audio and data communication. 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.

Introduction

For many years, RTC(Real Time Communication) components were expensive, complex and needed to be licensed – putting RTC out of the reach of individuals and smaller companies.

Gmail video chat became popular in 2008, and in 2011 Google introduced Hangouts, which use the Google Talk service (as does Gmail). Google bought GIPS, a company which had developed many of the 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, the IETF and W3C, and other browser makers like Mozilla and Opera to ensure industry consensus. And thus WebRTC was born.

WebRTC is a free, open project that enables web browsers with Real-Time Communications (RTC) capabilities via simple JavaScript APIs. The WebRTC components have been optimized to best serve this purpose. The goal is to enable rich, high quality, RTC applications to be developed in the browser via simple JavaScript APIs and HTML5 without the need of external, platform dependent native plugins.

Just like HTML5, WebRTC is not just a whole new shiny piece of technology. It is combination of several pieces of technologies grouped together to make real time web application development easier.

The main functions of WebRTC can be broadly categorized into three types.

1. Access and acquire video and audio streams

2. Establish a connection between peers and stream audio/video.
3. Communicate arbitrary data.

This functions are performed by the use of various JavaScript APIs such as getUserMedia, Web Audio and WebSocket which emerged at the same time as WebRTC. Future integration of WebRTC with technologies like WebGL will bring more and more exciting things for the web. For example, we will be able to play full 3D multiple games with stunning graphics right in our browsers soon; thanks to WebGL and WebRTC.

WHY WebRTC?

There are already several ways to do real time communication. So why should we use webrtc? But WebRTC has several advantages over them. WebRTC Let us take a look at various advantages that WebRTC offers compared to existing technologies.

Open Source is better than Proprietary:

Most of the software used now for real time communication are proprietary. Take Skype, Google Hangouts for example. There is no way to view, distribute or modify the source code legally. Hence building your own real time application is hard. Codecs used for encoding and decoding the media is proprietary too. So there is lots of licensing issues involved.

On the other hand, WebRTC is completely open sourced from the beginning. Everyone is free to view, reuse, modify and distribute the code. There is no licensing issues involved. And also from an ethical point of view, knowledge should always be free and open. So WebRTC is certainly better on this front.

Nobody likes Plugins:

In the existing scenario, if you want to use an web based real time communication application, you have to download and install a native plugin first. For example, if you have ever used Google Hangouts or Facebook Video Chat, the application will prompt you to download and install a plugin first. This plugin enables the browser to access the webcam and microphone of the user. Users generally don't like downloading and installing things.

On the contrary, with WebRTC everything is built into the browser itself. So there is no need to install anything. An web application can directly access the webcam and audio.

And WebRTC offers improved security too. Plugins are natively executed. So they can harm your computer if there is malicious code involved. But with WebRTC, everything runs inside the sandbox of your browser. Hence, there is lower security risk.

Peer-to-peer is better than Centralised:

In normal web architecture, the data travels from one client to a centralised server and then to another client. Hence the load on the server, in terms of both bandwidth use and request processing is very high.

But since, WebRTC uses a peer to peer model, the data travels directly from one client to the other. This also provides greater privacy. Your data does not go through the server, so there is no way anyone can tap into the server and snoop on you. WebRTC is a great step towards privacy on web.

Also a peer to peer model offers lower latency which is important in case of real time application.

Platform independent code is better than platform dependent code:

PLugins and native applications are generally platform dependent. But since WebRTC is web based and runs in the browser itself, WebRTC applications are platform independent. This facilitates for easier

development and maintenance of applications for all platforms, be it Linux, OS X, Microsoft Windows or any mobile platform.

How it works?

As stated before, the main function of WebRTC application can be divided into three broad categories.

1. Access and acquire video and audio streams
2. Establish a connection between peers and stream audio/video.
3. Communicate arbitrary data.

WebRTC uses three different JavaScript APIs to perform these three functions. These APIs are:

1. MediaStream (aka getUserMedia)
2. RTCPeerConnection
3. RTCDataChannel

MediaStream API performs the task of accessing the webcam and/or microphone of the device and acquire the video and/or audio stream from them. RTCPeerConnection API establishes connection between peers and streams audio and video data. This API also does all the encoding and decoding of audio/video data. The third API, RTCDataChannel helps to communicate arbitrary data from one client to the other.

Together, these three APIs are called WebRTC. These APIs are abstraction layers on top of all the low level work that browsers do to make real time communication possible. The whole point of WebRTC technology is to provide web developers with simple APIs to build

real time web applications without needing them to worry about the complexities underneath. So these abstraction layers do exactly that.

Now let us take a more detailed look at each of these APIs.

MediaStream API:

A media stream, in general, represents a stream of audio or video data. Similarly, the MediaStream interface of the WebRTC API describes a stream of audio or video data. A MediaStream object represents a linear, potentially infinite timeline. It is not preloadable, nor is it seekable. For example, a stream taken from camera and microphone input has synchronized video and audio tracks.

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

The getUserMedia() method takes three parameters:

- A constraints object.
- A success callback which, if called, is passed a MediaStream.
- A failure callback which, if called, is passed an error object.

Each MediaStream has a label, such as 'Xk7EuLhsuHKbnjLWkW4yYGNJJ8ON'. An array of MediaStreamTracks is returned by the getAudioTracks() and getVideoTracks() methods.

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.

The URL.createObjectURL() method converts a MediaStream to a URL which can be set as the src of a video element. (In Firefox and Opera, the src of the video can be set from the stream itself.)

Here is a code snippet with navigator.getUserMedia in action.

```
navigator.getUserMedia  = navigator.getUserMedia ||
                          navigator.webkitGetUserMedia ||
                          navigator.mozGetUserMedia ||
                          navigator.msGetUserMedia;
var video = document.querySelector('video');
if (navigator.getUserMedia) {
  navigator.getUserMedia({audio: true, video: true}, function(stream)
{
    video.src = window.URL.createObjectURL(stream);
  }, errorCallback);
} else {
  video.src = 'somevideo.webm'; // fallback.
}
```

The application has to be granted permission by the user first to be able to use the webcam or microphone of the user. So there is no way an application can use your webcam or microphone without your explicit permission.

The intention is eventually 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.

Note that for any webkit based browser, getUserMedia() must be used on a server , not the local file system, otherwise a PERMISSION_DENIED: 1 error will be thrown. This is to avoid certain security issues in Chrome.

getUserMedia() can be used in combination with other JavaScript APIs and libraries like WebGL, canvas etc.

Setting media constraints (resolution, height, width):

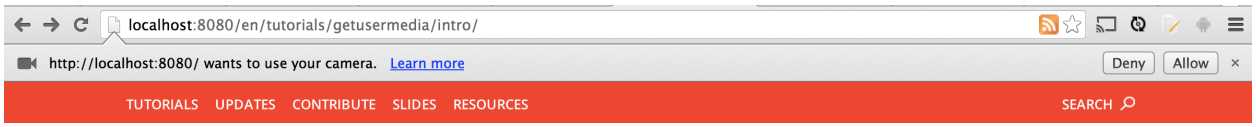
The first parameter to getUserMedia() can also be used to specify more requirements (or constraints) on the returned media stream. For example, instead of just indicating you want basic access to video (e.g. {vide: true}), you can additionally require the stream to be HD:

```
var hdConstraints = {
  video: {
    mandatory: {
      minWidth: 1280,
      minHeight: 720
    }
  }
};

navigator.getUserMedia(hdConstraints, successCallback,
errorCallback);
```

```
...

var vgaConstraints = {
  video: {
    mandatory: {
      maxWidth: 640,
      maxHeight: 360
    }
  }
};
```

Security

Some browsers throw up an infobar upon calling getUserMedia(), which gives users the option to grant or deny access to their camera/mic. The spec unfortunately is very quiet when it comes to security. For example, here is Chrome's permission dialog:



Permission dialog in Chrome

If your app is running from SSL (https://), this permission will be persistent. That is, users won't have to grant/deny access every time.

Providing fallback

For users that don't have support for getUserMedia(), one option is to fallback to an existing video file if the API isn't supported and/or the call fails for some reason:

```
// Not showing vendor prefixes or code that works cross-browser:

function fallback(e) {
  video.src = 'fallbackvideo.webm';
}

function success(stream) {
  video.src = window.URL.createObjectURL(stream);
}

if (!navigator.getUserMedia) {
  fallback();
} else {
  navigator.getUserMedia({video: true}, success, fallback);
}
```

In general, device access on the web has been a tough cookie to crack. Many people have tried, few have succeeded. Most of the early ideas have never taken hold outside of a proprietary environment nor have they gained widespread adoption.

The real problem is that the web's security model is *very* different from the native world. For example, I probably don't want every Joe Shmoe web site to have random access to my video camera. It's a tough problem to get right.

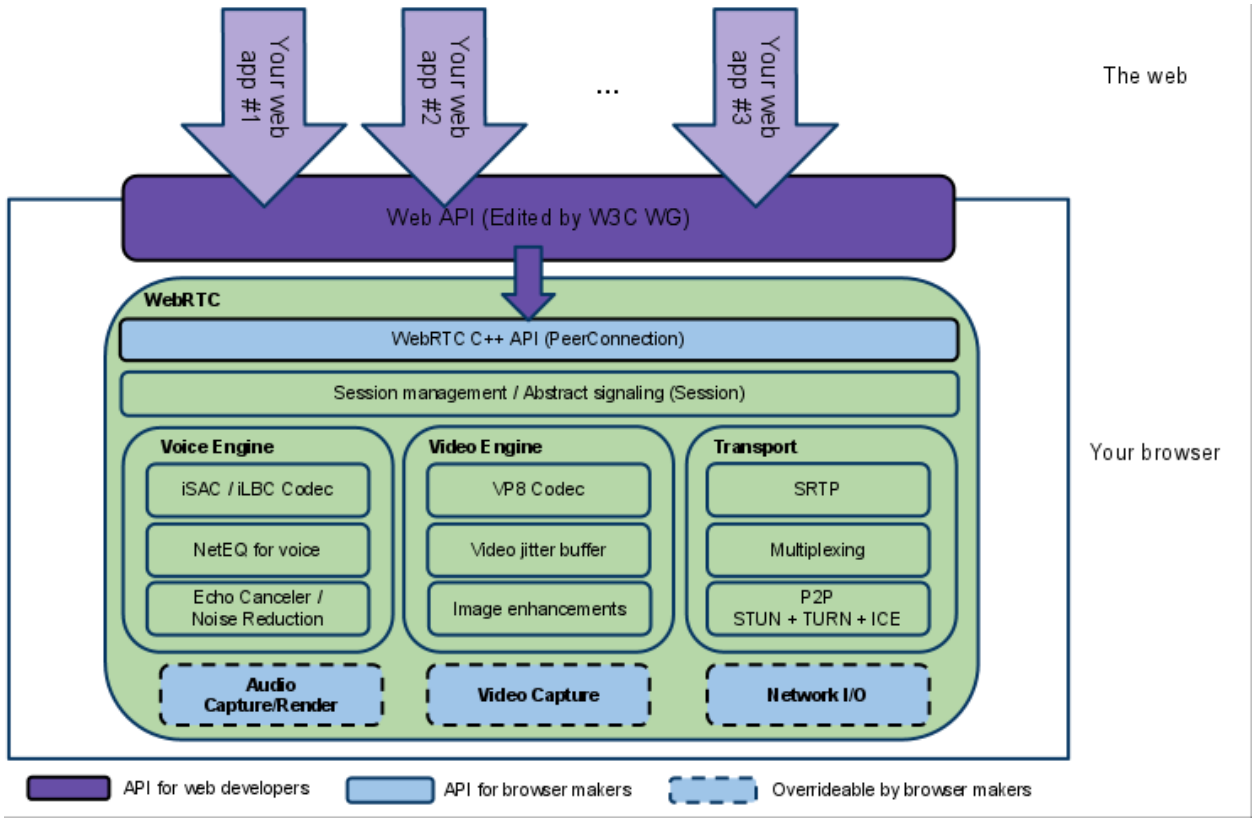
Bridging frameworks like PhoneGap have helped push the boundary, but they're only a start and a temporary solution to an underlying problem. To make web apps competitive to their desktop counterparts, we need access to native devices.

getUserMedia() is but the first wave of access to new types of devices. I hope we'll continue to see more in the very near future!

RTCPeerConnection API:

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! The RTCPeerConnection API works on this architecture as an abstraction layer to simplify things for the web developers.



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'

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, Websockets or XMPP, and any appropriate duplex (two-way) communication channel.

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 WebRTC W3C Working Draft, 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, RTCPeerConnection is currently prefixed.

```
var signalingChannel = createSignalingChannel();
var pc;
var configuration = ...;

// run start(true) to initiate a call
function start(isCaller) {
    pc = new RTCPeerConnection(configuration);

    // send any ice candidates to the other peer
    pc.onicecandidate = function (evt) {
        signalingChannel.send(JSON.stringify({ "candidate":
evt.candidate }));
    };

    // once remote stream arrives, show it in the remote video
element
    pc.onaddstream = function (evt) {
        remoteView.src = URL.createObjectURL(evt.stream);
    };

    // get the local stream, show it in the local video element and
send it
    navigator.getUserMedia({ "audio": true, "video": true }, function
```

```
(stream) {
        selfView.src = URL.createObjectURL(stream);
        pc.addStream(stream);

        if (isCaller)
            pc.createOffer(gotDescription);
        else
            pc.createAnswer(pc.remoteDescription, gotDescription);

        function gotDescription(desc) {
            pc.setLocalDescription(desc);
            signalingChannel.send(JSON.stringify({ "sdp": desc }));
        }
    });
}

signalingChannel.onmessage = function (evt) {
    if (!pc)
        start(false);

    var signal = JSON.parse(evt.data);
    if (signal.sdp)
        pc.setRemoteDescription(new
RTCSessionDescription(signal.sdp));
    else
        pc.addIceCandidate(new RTCIceCandidate(signal.candidate));
};
```

1. First up, Alice and Bob exchange network information. Alice creates an RTCPeerConnection object with an onicecandidatehandler.
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 callsaddIceCandidate, to add the candidate to the remote peer description.

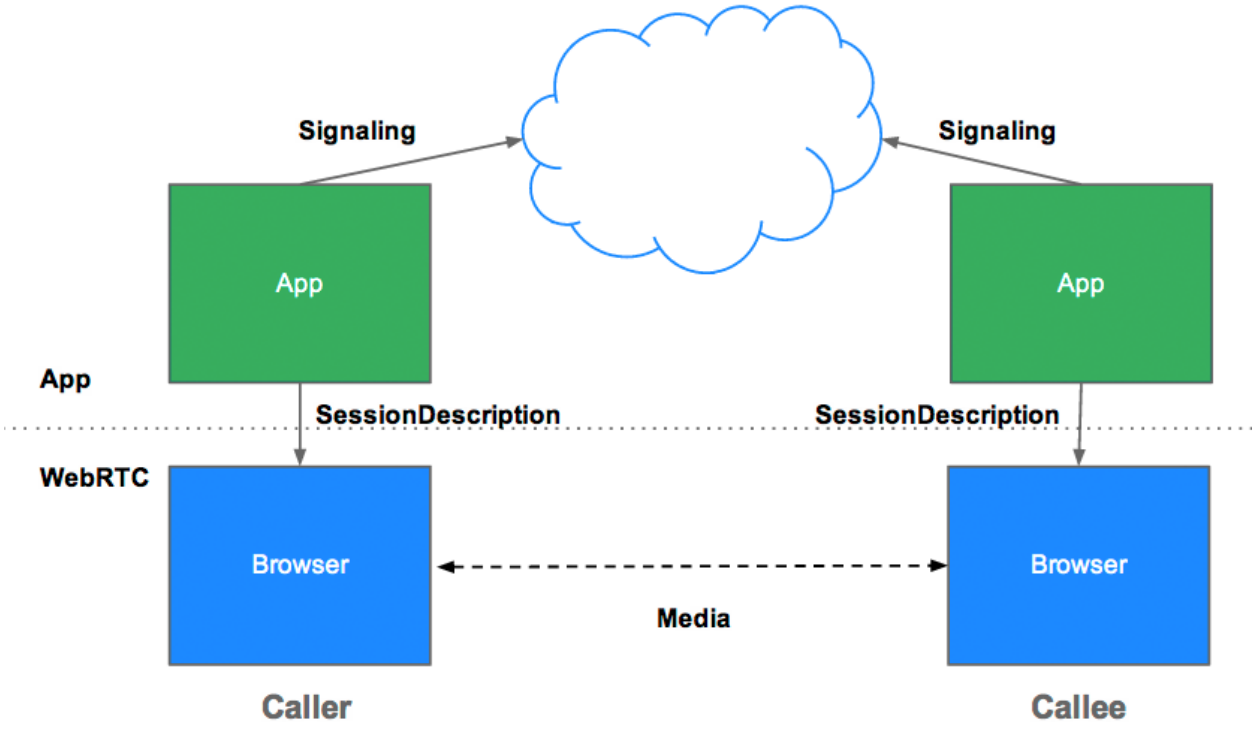
WebRTC clients (known as **peers**, aka Alice and Bob) also need to ascertain and exchange local and remote audio and video media information, such as resolution and codec capabilities. Signaling to exchange media configuration information proceeds by exchanging an *offer* and an *answer* using the Session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

1. Alice runs the RTCPeerConnection createOffer() method. The callback argument of this is passed an RTCSessionDescription: Alice's local session description.
2. In the callback, Alice sets the local description usingsetLocalDescription() 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 usingsetRemoteDescription().
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!

The offer/answer architecture described above is called JSEP, JavaScript Session Establishment Protocol.



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. Streaming is the job of RTCPeerConnection.

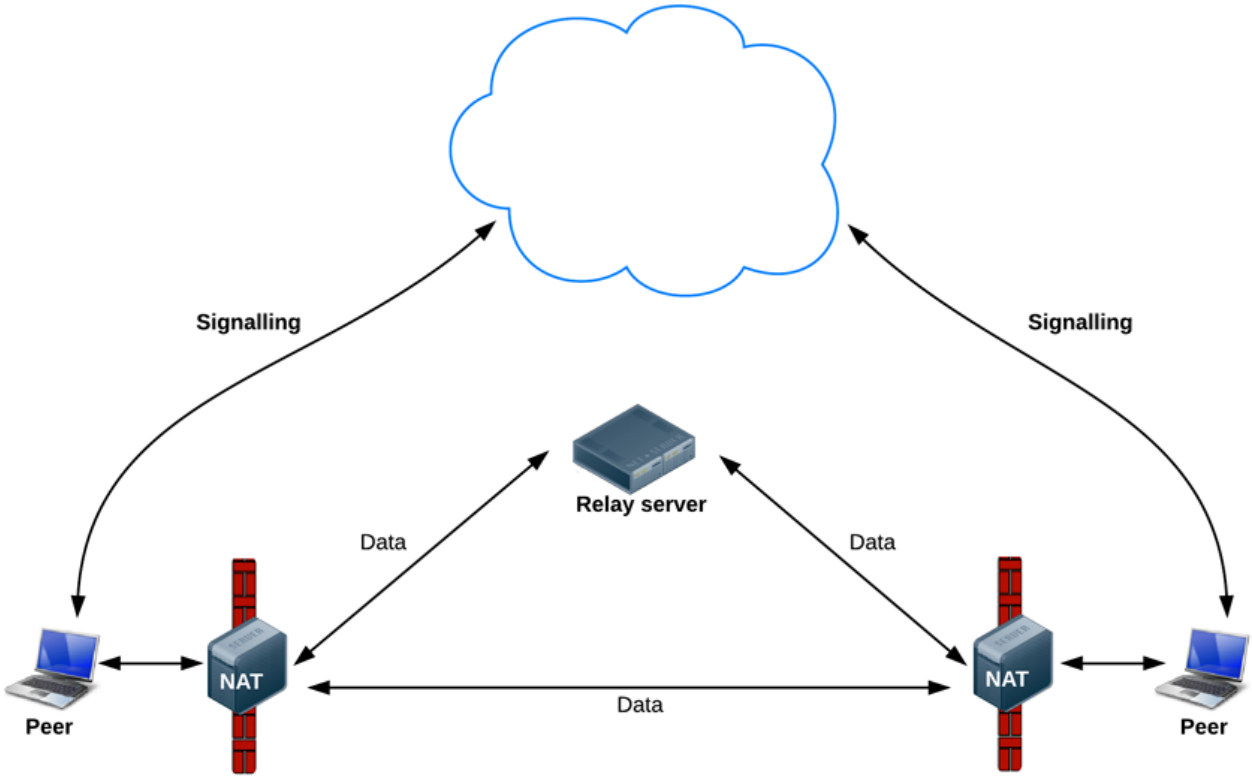
WebRTC needs four types of server-side functionality:

- 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 report. 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.

If UDP fails, ICE tries TCP: first HTTP, then HTTPS. 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

RTCDataChannel API:

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

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.

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.

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.'

REFERENCES

1. http://www.webrtc.org/
2. http://en.wikipedia.org/wiki/WebRTC
3. http://dev.w3.org/2011/webrtc/editor/webrtc.html
4. http://www.html5rocks.com
5. http://docs.webplatform.org/wiki/apis/webrtc
6. http://www.youtube.com/watch?v=p2HzZkd2A40&feature=player_embedded
7. https://groups.google.com/forum/#!forum/discuss-webrtc