WebRTC

WebRTC Tutorial

Alan Johnston, PhD
Distinguished Engineer
Avaya

Dan Burnett, PhD
Representing Voxeo, an Aspect Company
http://standardsplay.com

WebRTC Tutorial

- We will try to “show” you WebRTC instead of “tell” you about it
- We will primarily use the demo application from our book.
  - http://demo.webrtcbook.com
- We will show
  - Code (client and server)
  - Browser developer tools
  - Wireshark protocol captures
    - http://wireshark.org
- We hope it will be useful and informative!

Third Edition

- 3rd Edition of WebRTC book available
  - http://webrtcbook.com
- Paperback available on Amazon and DRM-free eBook available on Kindle and Smashwords (multiple formats including EPUB, MOBI, PDF, etc.)

What is WebRTC?

The Browser RTC Function

- New Browser Real-Time Communication (RTC) Function built-in to browsers
- Contains
  - Audio and video codecs
  - Ability to negotiate peer-to-peer connections
  - Echo cancellation, packet loss concealement
- In Chrome and Firefox today, planned for Internet Explorer, and Safari perhaps one day

Benefits of WebRTC

For Developer

- Streamlined development – one platform
- Simple APIs – detailed knowledge of RTC protocols not needed
- NAT traversal only uses expensive relays when no other choice
- Advanced voice and video codecs without licensing

For User

- No download or install – easy to use
- All communicaton encrypted – private
- Reliable session establishment – “just works”
- Excellent voice and video quality
- Many more choices for real-time communication

WebRTC Triangle

- Both browsers running the same web application from web server
- Peer Connection established between them with the help of the web server

WebRTC Trapezoid

- Similar to SIP Trapezoid
- Web Servers communicate using SIP or Jingle or proprietary
- Could become important in the future.

WebRTC Usage in Brief

WebRTC usage in brief

API Overview

Two main API sets

- Media capture
  - W3C "Media Capture and Streams" specification
  - Defines getUserMedia()
- Media transmission
  - W3C "WebRTC" specification
  - Defines RTCPeerConnection

About the demos today

- Based on Node.js
  - http://nodejs.org/
- Designed for teaching (not UI or performance)
- All code is available for download at our book site
  - http://webrtcbook.com/code3.html
- Running on single Amazon EC2 micro instance

Local Media

Local Media demo

Media capture and use

- **`Navigator.getUserMedia()`**
  - Request camera, microphone access
  - Permission check required for http page
- <video>**`.srcObject =`** <mediastream>
  - *Direct assignment*
  - Works for <audio> as well
- **`new MediaStream()`**
  - Can build from tracks in other streams

localmedia.html

Signaling

Signaling channel demo

Intro to Signaling

Signaling Example: HTTP Polling

Signaling Example: WebSocket Proxy

WebRTC Signaling using SIP

- Browser runs a SIP User Agent by running JavaScript from Web Server
- SRTP media connection uses WebRTC APIs
- Details in new RFC 7118 that defines SIP transport over WebSockets

WebRTC Signaling Approaches

- Signaling is required for exchange of candidate transport addresses, codec information, media keying information
- Many options – choice is up to web developer

Approach	Server Requirements	Advantages
WebSocket Proxy	WebSocket server with server code	No signaling infrastructure needed
XML HTTP Request	Web server with server code	No signaling infrastructure needed
SIP	SIP Registrar/Proxy Server which supports SIP WebSocket transport	Easy to interoperate with SIP endpoints or infrastructure, no server code needed
Jingle	XMPP server which supports XMPP WebSocket transport	Easy to interoperate with Jingle endpoints or infrastructure, no server code needed
Data Channel	WebSocket or web server to establish Data Channel	Low latency signaling and signaling privacy

Using the signaling channel

- createSignalingChannel() provides
  - Async connect(), send() methods
  - onwaiting, onconnected, onmessage handlers
- 2 party chat room by key
  - First to connect waits for the second
  - When second arrives, they connect
  - Another connection resets and becomes first

signaling.html

Peer Connection and Offer/ Answer

Peer Connection and Offer/Answer demo

SDP Offer/Answer

Session Description Protocol (SDP)

- Used by browser and WebRTC APIs to describe media session
- SDP is widely used in SIP VoIP and video systems
- Browser use of SDP is based on Offer/Answer
  - Defined in JSEP - JavaScript Session Establishment Protocol

SDP Extensions in WebRTC

- A number of new SDP extensions being developed for WebRTC. They include:
  - BUNDLE – a way to signal that a set of m= media lines should be multiplexed over the same transport address and port
  - MSID – a way to signal the API stream and track ID in SDP
- These extensions are still under active development

RTCPeerConnection

- Stream stuff
  - addStream(), removeStream()
  - onaddstream, onremovestream
- Offer/answer stuff
  - createOffer(), createAnswer()
  - setLocalDescription(), setRemoteDescription()
- Much more

Will soon be track-based

Offer/Answer State Machine

peerconnection.html

Media

WebRTC Media Encryption

- Media is ***always*** encrypted in WebRTC
- Secure RTP (SRTP) is used to assure confidentiality and authentication of RTP and RTCP packets
- SRTP is negotiated as part of SAVPF profile
- SRTP uses DTLS (Datagram TLS) to generate session keys during DTLS Handshake

Codecs

Codec	Use	Specification
Opus	Narrowband to wideband Internet audio codec for speech and music	RFC 6716
G.711	PCM audio encoding for PSTN interworking and backwards compatibility with VoIP systems	RFC 3551
Telephone Events	Transport of Dual Tone Multi Frequency (DTMF) tones	RFC 4733
H.264	Proposed video codec requiring licensing	RFC 6184
VP8	Proposed open source video codec	RFC 6386

- Mandatory to Implement (MTI) audio codecs are settled on Opus and G.711 (finally!)
- Video MTI is making progress: VP8 and H.264 for browsers had strong consensus at IETF-91 meeting – text not yet in draft

Data Channel

Data Channel real-time text demo

Data Channel

Data Channel Protocols

- Data channel provides a non-media channel between browsers
- Stream Control Transport Protocol (SCTP) provides reliability, congestion control, and message delivery
- Multiplexed over same ports as RTP media

Data channel

- RTCPeerConnection.createDataChannel()
  - send() method for async message sending
  - onmessage handler for async message receipt
- Bi-directional by default
- Can send strings or ArrayBuffer types

data.html

NAT Traversal and STUN

Peer-to-Peer Media with WebRTC

Media can go directly between browsers!

NAT Complicates Peer-to-Peer Media

Most browsers are behind NATs on the Internet, which complicates the establishment of peer-to-peer media sessions.

NAT – Network Address Translator

A NAT maps an "inside" address to an "outside" address allowing multiple hosts to share the same IP address.

WebRTC NAT Traversal

- Built into WebRTC browsers
  - Browser gathers address candidates which are then tested using connectivity checks
  - ICE (Interactive Connectivity Establishment) is the protocol for "hole punching"
- Candidates types
  - Host: Often a private IP address
  - Reflexive: Public IP address of NAT
  - Relayed: Media relay as last resort
- Still requires some servers (ICEservers)

STUN Session Traversal Utilities for NAT

Browser sends STUN test packet to STUN server to learn its public IP address (address of the NAT).

No authentication of STUN!

TURN and Firewall Traversal

TURN – Traversal Using Relay around NAT

In some cases, hole punching fails, and a TURN Media Relay on the public Internet must be used.

TURN Requires Authentication

- TURN relays media for duration of call, so it needs authentication
- TURN uses long-term authentication in STUN
  - Uses username and password credential per realm
  - Used to control access to TURN server resources
- TURN Servers can be
  - In the cloud
  - In an enterprise firewall or DMZ
- Active work in IETF TRAM WG to extend TURN authentication

TURN Firewall Traversal

Firewall traversal can be done with a TURN Server in DMZ trusted by the firewall.

Great online resources

- Informational sites
  - http://webrtc.org
  - http://html5rocks.com/en/tutorials/webrtc/basics
  - http://webrtchacks.com
- Games/demos/apps
  - http://www.cubeslam.com
  - http://shinydemos.com/facekat
  - http://sharefest.me (github.com/Peer5/Sharefest)

Other APIs

- To help WebRTC programming
  - Signaling services
    - PubNub, FireBase
  - Libraries, code snippets
    - EasyRTC, sipML5, JsSIP, reSIProcate
- Alternate (higher-level) APIs
  - Phono, Tokbox, APIDaze, Crocodile, vLine, Twilio

Standards Update

- Core (IETF) protocols stabilizing
  - Multi-track support finally settling
  - Video MTI codec looks close to being settled
- Core Media Capture APIs stabilizing
  - Will now be Promises based
- WebRTC APIs
  - Controls will be track-based
  - New per-track info objects originally from ORTC
  - Great progress on screen sharing
- Summary: you can build apps today with core APIs

Get Certified!

WebRTC School

Learn and Qualify

The WebRTC School is the web's premier location for WebRTC Integrator and Developer education. Associated with the training programs are the industry supported certifications, the WebRTC School Qualified Integrator (WSQI™) and WebRTC School Qualified Developer (WSQD™). For more information and online demos, please visit:

http://www.webrtcschool.com

WebRTC | WSQD™ WebRTC School Qualified Developer

WebRTC | WSQI™ WebRTC School Qualified Integrator

Questions?

http://webrtcbook.com

Backup Slides

WebRTC Protocols and IETF Standards

WebRTC Protocols

WebRTC: A Joint Standards Effort

- Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) are working together on WebRTC
- IETF
  - Protocols – "bits on wire"
  - Main protocols are already RFCs, but many extensions in progress
  - RTCWEB (Real-Time Communications on the Web) Working Group is the main focus, but other WGs involved as well
    - MMUSIC WG (ICE and SDP extensions)
    - TRAM WG (STUN and TURN extensions)
    - AVTCORE WG (RTP extensions)
    - RMCAT WG (RTP Media Congestion Avoidance)
- W3C
  - APIs – used by JavaScript code in HTML5
  - WEBRTC WG and Media Capture TF are main groups

IETF RTCWEB Documents

Document	Title
Overview	"Overview: Real Time Protocols for Browser-based Applications"
Use Cases and Requirements	"Web Real-Time Communication Use-cases and Requirements"
RTP Usage	"Web Real-Time Communication (WebRTC): Media Transport and Use of RTP"
Security Architecture	"RTCWEB Security Architecture"
Threat Model	"Security Considerations for RTC-Web"
Data Channel	"RTCWeb Data Channels"
Data Protocol	"WebRTC Data Channel Establishment Protocol"
JSEP	"JavaScript Session Establishment Protocol"
Audio	"WebRTC Audio Codec and Processing Requirements"
Consent Freshness	"STUN Usage for Consent Freshness"
Transports	"Transports for RTCWEB"

Signaling channel code

- No WebRTC standards here
- Based on XMLHttpRequest (but uses JSON)
- Outgoing: send() does XHR call to '/send'
- Incoming: polls once each second
  - Each is XHR to '/get'
- "Connection"
  - Gets id
  - Begins polling

clientXHRSignalingChannel.js

```
var createSignalingChannel = function(key, handlers) {
  . . .
  function connect(failureCB) { . . . }

  function poll() { . . . }
  function get() { . . . }

  function send() { . . . }
}
```

clientXHRSignalingChannel.js (connect)

```
// Set up connection with signaling server
function connect(failureCB) {
  . . .
// open XHR and send the connection request with the key
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("GET", "/connect?key=" + key);
  client.send();
}
```

clientXHRSignalingChannel.js (connect handler)

```
waitingHandler = initHandler(handlers.onWaiting),
connectedHandler = initHandler(handlers.onConnected),
. . .
// Handle connection response
function handler() {
  . . .
  // save status and server-generated id, start async polling
  id = res.id;
  status = res.status;
  poll();

  // run user-provided handlers for waiting and connected states
  if (status === "waiting") { waitingHandler();
  } else { connectedHandler();
  }
  return;
}
```

clientXHRSignalingChannel.js (polling get)

```
// poll() waits n ms between gets to the server.  n is at 10 ms
// for 10 tries, then 100 ms for 10 tries, then 1000 ms from then
// on. n is reset to 10 ms if a message is actually received.
function poll() {
  // pollWaitDelay defined with reset() and increase() methods

  // getLoop is defined and used immediately here.  It retrieves
  // messages from the server and then schedules itself to run
  // again after pollWaitDelay.value() milliseconds.
  (function getLoop() {
    get(getHandler);
  }());
}
```

clientXHRSignalingChannel.js (polling get handler)

```
function getHandler (response) {
  // if messages property exists, then we are connected
  if (response.msgs && (status !== "connected")) {
     // switch status to connected since it is now!
     status = "connected";
     connectedHandler();
  }
  if (msgs.length > 0) {          // we got messages
    pollWaitDelay.reset();
    for (i=0; i<msgs.length; i+=1) {
      handleMessage(msgs[i]);
    }
  } else {                        // didn't get any messages
    pollWaitDelay.increase();
  }
  // now set timer to check again
  setTimeout(getLoop, pollWaitDelay.value());
}
```

clientXHRSignalingChannel.js (polling get)

```
// This function is part of the polling setup to check for
// messages from the other browser.  It is called by getLoop()
// inside poll().
function get(getResponseHandler) {
  // response should either be error or a JSON object.  If the
  // latter, send it to the user-provided handler.
  function handler() {
    var res = JSON.parse(this.response);
    getResponseHandler(res);
    return res;
  }

  // open XHR and request messages for my id
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("POST", "/get");
  client.send(JSON.stringify({"id":id}));
}
```

clientXHRSignalingChannel.js (send)

```
// Send a message to the other browser on the signaling
channel
function send(msg, responseHandler) {
  // parse response and send to handler
  function handler() {
    var res = JSON.parse(this.response);
    responseHandler(res);
    return;
  }

  // open XHR and send my id and message as JSON string
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("POST", "/send");
  var sendData = {"id":id, "message":msg};
  client.send(JSON.stringify(sendData));
}
```

serverXHRSignalingChannel.js

```
// handle XML HTTP Request to connect using a given key
function connect(info) {
}
exports.connect = connect;

// Returns all messages queued for info.postData.id
function getMessages(info) {
}
exports.get = getMessages;

// Queues message in info.postData.message for sending to the
// partner of the id in info.postData.id
function sendMessage(info) {
}
exports.send = sendMessage;
```

serverXHRSignalingChannel.js (connect)

```
// handle XML HTTP Request to connect using a given key
function connect(info) {
  . . .
  log("Request handler 'connect' was called.");
  if (query && query.key) {
    var thisconnection = connections[query.key] ||
                         {status:"new"};
    if (thisconnection.status == "waiting") { // first half ready
      connectSecondParty(); return;
    } else {              // must be new or status of "connected"
      connectFirstParty(); return;
    }
  } else {
    webrtcError("No recognizable query key", res);
  }
}
exports.connect = connect;
```

serverXHRSignalingChannel.js (connect first party)

```
var connectFirstParty = function() {
  if (thisconnection.status == "connected") {
    // delete pairing and any stored messages
    delete partner[thisconnection.ids[0]];
    delete partner[thisconnection.ids[1]];
    delete messagesFor[thisconnection.ids[0]];
    delete messagesFor[thisconnection.ids[1]];
  }
  connections[query.key] = {};
  thisconnection = connections[query.key];
  thisconnection.status = "waiting";
  thisconnection.ids = [newID()];
  webrtcResponse({"id":thisconnection.ids[0],
                  "status":thisconnection.status}, res);
}
```

serverXHRSignalingChannel.js (connect second party)

```
var connectSecondParty = function() {
  thisconnection.ids[1] = newID();
  partner[thisconnection.ids[0]] = thisconnection.ids[1];
  partner[thisconnection.ids[1]] = thisconnection.ids[0];
  messagesFor[thisconnection.ids[0]] = [];
  messagesFor[thisconnection.ids[1]] = [];
  thisconnection.status = "connected";
  webrtcResponse({"id":thisconnection.ids[1],
                  "status":thisconnection.status}, res);
};
```

serverXHRSignalingChannel.js (getMessages)

```
// Returns all messages queued for info.postData.id
function getMessages(info) {
  var postData = JSON.parse(info.postData),
      res = info.res;

. . . Typecheck data . . .

  log("Sending messages ***" +
      JSON.stringify(messagesFor[postData.id]) + "*** to id "
+
      postData.id);
  webrtcResponse({'msgs':messagesFor[postData.id]}, res);
  messagesFor[postData.id] = [];
}
exports.get = getMessages;
```

serverXHRSignalingChannel.js (sendMessage)

```
// Queues message in info.postData.message for sending to the
// partner of the id in info.postData.id
function sendMessage(info) {
  log("postData received is ***" + info.postData + "***");
  var postData = JSON.parse(info.postData),
      res = info.res;

 . . . Typecheck data . . .

  messagesFor[partner[postData.id]].push(postData.message);
  log("Saving message ***" + postData.message +
      "*** for delivery to id " + partner[postData.id]);
  webrtcResponse("Saving message ***" + postData.message +
                 "*** for delivery to id " +
                 partner[postData.id], res);
}
exports.send = sendMessage;
```