What is ORTC?

A W3C Community Group to design an-object based API for RTC (ORTC == Object RTC)

The hope is to merge the work of the CG into the WebRTC WG as WebRTC 1.1.
Why is there an object model in WebRTC 1.0?

- Need a way to tweak params on individual tracks sent over the wire
  - Bitrate
  - Direction (sendonly/recvonly etc.)
- Existing control surfaces insufficient:
  - createOffer params - not per-track
  - AddStream params - not modifiable post-add
  - MST constraints - affects raw media, not encoding

We solved this for stats, by adding objects for each thing we care about, but these objects are hidden inside PeerConnection
WebRTC 1.0 to ORTC 1.1

JavaScript Application (Sender) —> SDP —> PeerConnection —> Track

JavaScript Application (Receiver) —> SDP —> PeerConnection —> Track

Can't control directly

JavaScript Application (Sender) —> Objects —> Track

JavaScript Application (Receiver) —> Objects —> Track

Control directly
ORTC Benefits

- Direct control of existing objects
- Signalling flexibility
- No SDP necessary
- Simulcast, Scalable Video Coding (SVC)
- Media forking (e.g. full mesh conferencing)
- Backwards compatible with WebRTC 1.0 API
- Continuing direction WebRTC 1.0 is headed
- Mobile and web friendly
ORTC Progress

- 07-2013 - ORTC CG (Community Group) formed
- 01-2014 - First ORTC draft API complete
- 05-2014 - WebRTC 1.0 added RtpSender / RtpReceiver / DtlsTransport (Washington DC Interim meeting)
- 07-2014 - First implementable ORTC draft API complete
- 09-2014 - ~ 75 members in ORTC CG

- First implementations to begin soon or are already underway (e.g. https://github.com/openpeer/ortc-lib)
ORTC Myths

- It's a revolution (it's an evolution)
- Competes with 1.0 (it's intended to fold into WG as 1.1)
- Disrupts 1.0 (it's a CG to avoid disrupting the WG)
- "Owned" by Microsoft (it's a community effort)
- Only for non-SIP/SDP signalling (it helps SDP aficionados as well)
- It’s only about simulcast/SVC (has many other benefits)
WebRTC 1.0 can be implemented on ORTC 1.1
W3C ORTC Community Group

- **W3C ORTC CG website:**
  - http://www.w3.org/community/ortc/

- **Public mailing list:** public-ortc@w3.org
  - Join [Here](mailto:public-ortc@w3.org) - link on the right hand side
  - Non-members can post to this list.
  - Non-member contributions are problematic.

- **Contributor’s mailing list:** public-ortc-contrib@w3.org
  - Join [Here](mailto:public-ortc-contrib@w3.org) - link on the right hand side
  - Members only, preferred list for contributions to the specification.
Associated Sites

● API Public Draft: http://ortc.org (upper right hand side)

● ORTC developer website: http://ortc.org/
  ○ Editor’s drafts, pointers to github repos, etc.

● ORTC API Issues List: https://github.com/openpeer/ortc/issues?state=open
Thank you

Slides that follow are extra resources for those who are reading the slides post presentation.
Part II: Detailed Walkthrough
ORTC 1.1 Big Picture
interface RTCRtpSender {
    static RTCRtpCapabilities getCapabilities ();
    void setTransport (RTCDtlsTransport transport);
    void setTrack (MediaStreamTrack track);
    void send (RTCRtpParameters parameters);
};

interface RTCRtpReceiver {
    static RTCRtpCapabilities getCapabilities ();
    void setTransport (RTCDtlsTransport transport);
    void receive (RTCRtpParameters parameters);
};
interface RTCDtlsTransport {
  RTCDtlsParameters getLocalParameters ();
  void start (RTCDtlsParameters remoteParameters);
};

interface RTCIceTransport {
  void start (RTCIceGatherer gatherer,
              RTCIceParameters remoteParameters,
              ...);
  void addRemoteCandidate (RTCIceGatherCandidate remoteCandidate);
};

interface RTCIceGatherer {
  RTCIceParameters getLocalParameters ();
  sequence<RTCIceCandidate> getLocalCandidates ();
  attribute EventHandler? onlocalcandidate;
};
dictionary RTCRtpParameters {
    DOMString muxId = "";
    sequence<RTCRtpCodecParameters> codecs;
    sequence<RTCRtpHeaderExtensionParameters> headerExtensions;
    sequence<RTCRtpEncodingParameters> encodings;
    RTCRtcpParameters rtcp;
};

dictionary RTCRtpCodecParameters { 
    DOMString name;
    payloadtype payloadType;
    // ...
};

dictionary RTCRtpHeaderExtensionParameters { 
    DOMString uri;
    unsigned short id;
    // ...
};
dictionary RTCRtpEncodingParameters {
    unsigned long          ssrc;
    payloadtype            codecPayloadType;
    RTCRtpFecParameters    fec;
    RTCRtpRtxParameters    rtx;
    double                 priority = 1.0;
    double                 maxBitrate;
    double                 minQuality = 0;
    double                 framerateBias = 0.5;
    double                 resolutionScale;
    double                 framerateScale;
    boolean                active = true;
    DOMString              encodingId;
    sequence<DOMString>    dependencyEncodingIds;
};
RTCRtpSender

- Encodes 1 media track (audio or video)
- Chooses which RTCDtlsTransport to use

[Constructor(MediaStreamTrack track, RTCDtlsTransport transport, optional RTCDtlsTransport rtcpTransport)]

interface RTCRtpSender : RTCStatsProvider {
    readonly attribute MediaStreamTrack track;
    readonly attribute RTCDtlsTransport transport;
    readonly attribute RTCDtlsTransport rtcpTransport;
    void setTransport (RTCDtlsTransport transport, optional RTCDtlsTransport rtcpTransport);
    void setTrack (MediaStreamTrack track);
    static RTCRtpCapabilities getCapabilities (optional DOMString kind);
    void send (RTCRtpParameters parameters);
    void stop ();
    attribute EventHandler? onerror;
    attribute EventHandler? onssrcconflict;
}
RTCRtpReceiver

- Decodes 1 media track (audio or video)
- Indicates which RTCDtlsTransport to receive upon

```
[Constructor(RTCDtlsTransport transport, optional RTCDtlsTransport rtcpTransport)]
interface RTCRtpReceiver : RTCStatsProvider {
    readonly attribute MediaStreamTrack? track;
    readonly attribute RTCDtlsTransport transport;
    readonly attribute RTCDtlsTransport rtcpTransport;
    void setTransport (RTCDtlsTransport transport, optional RTCDtlsTransport rtcpTransport);
    static RTCRtpCapabilities getCapabilities (optional DOMString kind);
    void receive (RTCRtpParameters parameters);
    void stop ();
    attribute EventHandler? onerror;
};
```
RTCDtlsTransport

- Negotiates secure data and media channel
- Maps RTCRtpSender / RTPRtcReceiver to RTCIceTransport

```javascript
[Constructor(RTCIceTransport transport)]
interface RTCDtlsTransport : RTCStatsProvider {
    readonly attribute RTCIceTransport transport;
    readonly attribute RTCDtlsTransportState state;
    RTCDtlsParameters getLocalParameters ();
    RTCDtlsParameters? getRemoteParameters ();
    sequence<ArrayBuffer> getRemoteCertificates ();
    void start (RTCDtlsParameters remoteParameters);
    void stop ();
    attribute EventHandler? ondtlsstatechange;
    attribute EventHandler? onerror;
}
```
RTCIceTransport

- Performs ICE connectivity checks
- Opens peer to peer channel for sending/receiving RTP/data

```javascript
[Constructor()]
interface RTCIceTransport : RTCStatsProvider {
    readonly attribute RTCIceGatherer? iceGatherer;
    readonly attribute RTCIceRole role;
    readonly attribute RTCIceComponent component;
    readonly attribute RTCIceTransportState state;
    sequence<RTCIceCandidate> getRemoteCandidates ();
    RTCIceCandidatePair? getNominatedCandidatePair ();
    void start (RTCIceGatherer gatherer, RTCIceParameters remoteParameters, optional RTCIceRole role);
    void stop ();
    RTCIceParameters? getRemoteParameters ();
    RTCIceTransport createAssociatedTransport ();
    void addRemoteCandidate (RTCIceGatherCandidate remoteCandidate);
    void setRemoteCandidates (sequence<RTCIceCandidate> remoteCandidates);
    attribute EventHandler? onicestatechange;
    attribute EventHandler? oncandidatepairchange;
};
```
RTCIceGatherer

• The object formerly own as “RTCIceListener”.
• Gathers IP addresses and ports to be used in possible ICE connectivity checks
• Gathers relay and firewall IPs and ports

[Constructor(RTCIceGatherOptions options)]
interface RTCIceGatherer {
  RTCIceParameters getLocalParameters ();
  sequence<RTCIceCandidate> getLocalCandidates ();
  attribute EventHandler? onerror;
  attribute EventHandler? onlocalcandidate;
};
RTCIceTransportController

- Regulates ICE connectivity across ICE transports
- Helps bandwidth estimation and control over transports

[Constructor()]
interface RTCIceTransportController {
  sequence<RTCIceTransport> getTransports ();
  void addTransport (RTCIceTransport transport, optional unsigned long index);
};
RTCRtpListener

- Used to receive notifications of unhandled RTP media traffic (so RTCRtpSenders / RTCRtpReceivers can be created on-the-fly)

```javascript
[Constructor(RTCDtlsTransport transport)]
interface RTCRtpListener {
    readonly attribute RTCDtlsTransport transport;
    attribute EventHandler? onunhandledrtp;
};
```
RTCDATACHANNEL

- Sending / receiving real time data

[Constructor(RTCDataTransport transport, RTCDataChannelParameters parameters)]

interface RTCDataChannel : EventTarget {
    readonly attribute RTCDataTransport transport;
    readonly attribute RTCDataChannelParameters parameters;
    readonly attribute RTCDataChannelState readyState;
    readonly attribute unsigned long bufferedAmount;
    attribute DOMString binaryType;

    void close ();
        attribute EventHandler onopen;
        attribute EventHandler onerror;
        attribute EventHandler onclose;
        attribute EventHandler onmessage;

    void send (DOMString data);
    void send (Blob data);
    void send (ArrayBuffer data);
    void send (ArrayBufferView data);
};
RTCSctpTransport

- Use for construction of real-time data channels
- Notifies of incoming real-time data channels

[Constructor(RTCDtlsTransport transport)]

```typescript
interface RTCSctpTransport : RTCDataTransport {
    readonly attribute RTCDtlsTransport transport;
    static RTCSctpCapabilities getCapabilities ();
    void start (RTCSctpCapabilities remoteCaps);
    void stop ();
    attribute EventHandler ondatachannel;
}
```
SVC (Scalable Video Coding)

- Enabling SVC need not be complicated.
- On the receiver side:
  - Not necessary to configure support for SVC if a compliant decoder can decode anything that the encoder can send. This is true of VP8 (which supports temporal scalability).
- On the sender side:
  - Support for SVC can be inferred from RTCRtpCapabilities (e.g. maxTemporalLayers > 0).
  - SVC MUST be enabled in RTCRtpEncodingParameters for the encoder to start using it.
    - SVC may not be appropriate for all situations, so developers need to be able to control when it is used.
    - Ongoing discussion of how to direct the browser to “automatically” configure SVC layering.
Part III: Example
Example

// Assume we have an audioTrack and a videoTrack to send.
// Assume also that we have a signaling function called signaller and a myCapsToSendParams library function.
// Create the ICE gatherer and transport. Since we are multiplexing RTP/RTCP and A/V we only need one.
var gatherOptions = new RTCIceGatherOptions;
gatherOptions.gatherPolicy = RTCIceGatherPolicy.all;
gatherOptions.iceservers = ...;
var iceGatherer = new RTCIceGatherer(gatherOptions);
iceGatherer.onerror = errorHandler;
iceGatherer.onlocalcandidate = function (event) {signaller.mySendLocalCandidate(event.candidate);}
var iceTransport = new RTCIceTransport();
iceTransport.onicestatechange = ...;
mySignaller.onRemoteCandidate = function(remote) {iceTransport.addRemoteCandidate(remote.candidate);}
// Create the DTLS transport. We only need one.
var dtlsTransport = new RTCDtlsTransport(iceTransport);
// Create the sender and receiver objects
var audioSender = new RtpSender(audioTrack, dtlsTransport);
var videoSender = new RtpSender(videoTrack, dtlsTransport);
var audioReceiver = new RtpReceiver(dtlsTransport);
var videoReceiver = new RtpReceiver(dtlsTransport);

// Retrieve the receiver and sender capabilities
var recvAudioCaps = RTCRtpReceiver.getCapabilities("audio");
var recvVideoCaps = RTCRtpReceiver.getCapabilities("video");
var sendAudioCaps = RTCRtpSender.getCapabilities("audio");
var recvVideoCaps = RTCRtpSender.getCapabilities("video");
// At this point, ICE/DTLS parameters and Send/Receive capabilities can be exchanged.
mySignaller.myOfferTracks({
    // Offer the ICE and DTLS parameters
    "ice": iceGatherer.getLocalParameters(),
    "dtls": dtlsTransport.getLocalParameters(),
    // Offer the receiver and sender audio and video capabilities.
    "recvAudioCaps": recvAudioCaps,
    "recvVideoCaps": recvVideoCaps,
    "sendAudioCaps": sendAudioCaps,
    "sendVideoCaps": sendVideoCaps
},)
Example (cont’d)

function(\text{answer}) {  
    \text{// The responder answers with its preferences, parameters and capabilities}
    \text{// Derive the send and receive parameters, assuming that RTP/RTCP mux will be enabled.}
    \text{var audioSendParams = myCapsToSendParams(sendAudioCaps, answer.recvAudioCaps);}
    \text{var videoSendParams = myCapsToSendParams(sendVideoCaps, answer.recvVideoCaps);}
    \text{var audioRecvParams = myCapsToRecvParams(recvAudioCaps, answer.sendAudioCaps);}
    \text{var videoRecvParams = myCapsToRecvParams(recvVideoCaps, answer.sendVideoCaps);}
    \text{// Since we only have a single ICE transport and DTLS transport,}
    \text{// no need for the ICE Transport Controller.}
    \text{iceTransport.start(iceGatherer,answer.ice,RTCIceRole.controlling);}
    \text{dtlsTransport.onerror = errorHandler;}
    \text{dtlsTransport.start(remote.dtls);}
};
Example (cont’d)

// Set the audio and video send and receive parameters.
audioSender.send(audioSendParams);
videoSender.send(videoSendParams);
audioReceiver.receive(audioRecvParams);
videoReceiver.receive(videoRecvParams);
});

// Now we can render/play
// audioReceiver.track and videoReceiver.track.

// Helper functions
function errorHandler (error) {
    console.log('Error encountered: ' + error.name);
}

//
Example (cont’d)

RTCRtpParameters function myCapsToSendParams (RTCRtpCapabilities sendCaps, RTCRtpCapabilities remoteRecvCaps) {
  // Function returning the sender RTCRtpParameters, based on the local sender and remote receiver capabilities.
  // The goal is to enable a single stream audio and video call with minimum fuss.
  // Steps to be followed:
  // 1. Determine the RTP features that the receiver and sender have in common.
  // 2. Determine the codecs that the sender and receiver have in common.
  // 3. Within each common codec, determine the common formats, header extensions and rtcpFeedback mechanisms.
  // 4. Determine the payloadType to be used, based on the receiver preferredPayloadType.
  // 5. Set RTCRtcpParameters such as mux to their default values.
  // 6. Return RTCRtpParameters enabling the jointly supported features and codecs.
}

RTCRtpParameters function myCapsToRecvParams (RTCRtpCapabilities recvCaps, RTCRtpCapabilities remoteSendCaps) {
  // Function returning the receiver RTCRtpParameters, based on the local receiver and remote sender capabilities.
  return myCapsToSendParams(remoteSendCaps, recvCaps);
}