



# Introduction to the WebRTC/ORTC object model



**Bernard Aboba**  
Principal Architect at Microsoft

**Robin Raymond**  
CTO at Optical Tone



# Goals of Today's Tutorial

- To describe the philosophy behind the object approach to Realtime Communications
- To introduce the realtime object model described in the ORTC API
  - <http://draft.ortc.org/>
- To explain the differences between the current ORTC specification and the implementation in Microsoft Edge
  - <https://rawgit.com/aboba/edgertc/master/msortc-rs3.html>
- To demonstrate approaches to ORTC application development
  - ORTC native application
  - WebRTC 1.0 application running on ORTC
- To introduce the WebRTC 1.0 object model

# Other WebRTC/ORTC Sessions

<https://www.cluecon.com/schedule.html>

- 3D Cloud Streaming for Mobile and Web
  - Monday, August 7, 1:30 PM – 2:30 PM
  - Will cover UWP SDKs, Azure, Hololens
- WebRTC Roundtable
  - Tuesday, August 8: 1 PM – 2:30 PM
  - Will cover status of Firefox, Chrome and Edge

# ORTC Philosophy

- ORTC *not* just for use in browsers.
  - Goal was to provide an API that could be useful for IoT, mobile, server *and* web development.
  - Desire for wide applicability implies need to support many usage scenarios.
- **Philosophy: Less is More**
  - Focus solely on the media plane.
  - Support for signaling protocols can be provided in add-on libraries (if needed).
  - Modular approach reduces CPU and memory demands, and improves battery life
  - (Imperfect) operating system analogy: micro-kernel architectures
- **Result: flexibility and the ability to support advanced capabilities**
  - Support for other APIs can be built on top of ORTC (e.g. WebRTC 1.0)
  - Wide range of signaling protocols can be supported: SIP, H.323, JSON signaling
  - Wide range of codecs can be fully supported: H.264/AVC, H.264/SVC, VP8, VP9/SVC, etc.
  - Support for advanced video technology: simulcast, scalable video coding, etc.

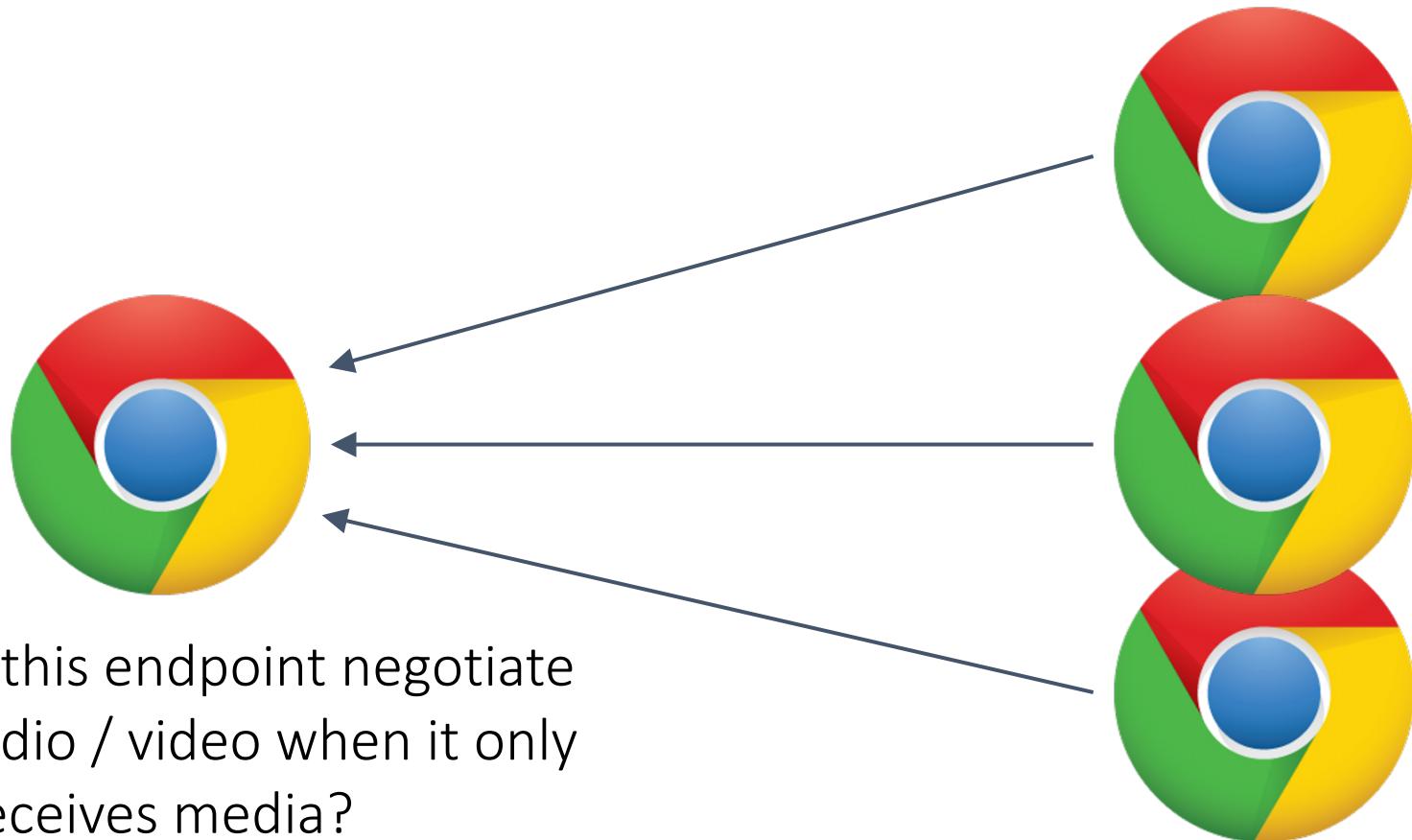
# ORTC Implementations

- ORTC Lib
  - <http://ortclib.org/>
  - An open-source library for UWP, iOS and Android
  - Enhancements for mobile and IoT (seamless roaming, improved battery life, etc.)
  - Details covered during the “ORTC Update” session on Tuesday.
- Mediasoup
  - <http://mediasoup.org/>
  - Selective Forwarding Unit (SFU) for node.js
- Microsoft Edge:
  - <https://developer.microsoft.com/en-us/microsoft-edge/testdrive/>
  - Browser implementation supporting ORTC (as well as WebRTC 1.0 via the adapter.js library)
  - More on Edge during the “Browser Update” session.

# ORTC Design Goals

- To provide a “**do as I say**” API without a dependency on the Session Description Protocol (SDP).
  - “Do as I say” means that the API either will do what has been requested, or an error will result.
  - SDP can be supported if needed via an add-on library, but can avoid the overhead of an SDP parser and Offer/Answer if your application does not need it.
- To support the functionality of WebRTC 1.0.
  - Enables the WebRTC 1.0 API to be emulated on top of ORTC.
- To support forking, simulcast (sending and receiving) and scalable video coding.
- To support a wide range of communications scenarios (such as asymmetric audio/video and negotiation-free adding/removing of media)

# Asymmetric Audio / Video



# Important ORTC related sites

[ortc.org](http://ortc.org)

(portal to all things ORTC related)

**ORTC**  
Object API for RTC - Mobile,  
Server, Web

**Home**  
Architecture  
Implementations  
Meetings  
History  
Future  
FAQ  
Developers

[Twitter](#) [GitHub](#) [Source](#)

**W3C ORTC COMMUNITY GROUP**  
[Join W3C ORTC CG](#)  
[Official W3C ORTC CG Website](#)  
[Public ORTC CG Email List](#)  
[Obsolete / archived ORTC CG](#)

[Share this!](#) [Twitter](#) [Facebook](#) [G+ Design](#)

## Welcome to ORTC!

ORTC (Object Real-Time Communications) is an API allowing developers to build next generation real-time communication applications for web, mobile, or server environments.

The [ORTC API](#) was designed by the [W3C ORTC CG](#) (Community Group) and originally founded by [Ihookflash](#) in 2013. This innovative community group consists of over 100 participating members, notably including [Google](#), [Microsoft](#), and many other industry leaders. See [history](#).

Microsoft has implemented the [ORTC API](#) to the [Edge browser](#), and the [ORTC Lib](#) open source project was created to allow mobile developers to take advantage of the ORTC API within mobile applications. See [implementations](#).

The ORTC API is on-the-wire compatible with WebRTC 1.0 and serves as the real-world implementation input for the future direction of the WebRTC API. In fact, many of ORTC's objects are already incorporated into the WebRTC 1.0 API. See [future](#).

The ORTC API was designed to allow the [WebRTC 1.0 API](#) to be written as a shim on top of the ORTC API. As demonstrated with [adapter.js](#) and implemented in ORTC lib, this allows developers to use the more familiar WebRTC 1.0 API and later take full advantage of what the object model offers. See [adapter](#).

[Share this!](#)

[Twitter](#) [Facebook](#) [G+ Design](#)

[draft.ortc.org](http://draft.ortc.org)

(latest Object RTC API draft)

 W3C Community Group  
Draft Report

**TABLE OF CONTENTS**

- 1. [Overview](#)
  - 1.1 [Terminology](#)
  - 1.2 [Scope](#)
- 2. [The RTCIceGatherer Object](#)
  - 2.1 [Overview](#)
  - 2.2 [Operation](#)
  - 2.3 [Interface Definition](#)
  - 2.4 [The RTCIceParameters Object](#)
  - 2.5 [The RTCIceCandidate Object](#)
    - 2.5.1 [The RTCIceProtocol](#)
    - 2.5.2 [The RTCIceTcpCandidateType](#)
    - 2.5.3 [The RTCIceCandidateType](#)
    - 2.6 [dictionary RTCIceCandidateComplete](#)
    - 2.7 [enum RTCIceGathererState](#)
    - 2.8 [RTCIceGathererIceErrorEvent](#)
    - 2.9 [RTCIceGathererEvent](#)
    - 2.10 [dictionary RTCIceGatherOptions](#)
    - 2.11 [enum RTCIceGatherPolicy](#)
    - 2.12 [enum RTCIceCredentialType](#)
    - 2.13 [The RTCIceServer Object](#)
    - 2.14 [Example](#)
  - 3. [The RTCIceTransport Object](#)
    - 3.1 [Overview](#)
    - 3.2 [Operation](#)
    - 3.3 [Interface Definition](#)
    - 3.4 [enum RTCIceComponent](#)
    - 3.5 [enum RTCIceRole](#)
    - 3.6 [enum RTCIceTransportState](#)
    - 3.7 [RTCIceCandidatePairChanneledEvent](#)

## Object RTC (ORTC) API for WebRTC

Draft Community Group Report 04 October 2016



**Editor:**

[Robin Raymond, Optical Tone Ltd.](#)

**Authors:**

[Bernard Aboba, Microsoft Corporation](#)  
[Justin Uberti, Google](#)

**Participate:**

[Mailing list](#)  
[Browse open issues](#)  
[IETF RTCWEB Working Group](#)

Copyright © 2016 the Contributors to the Object RTC (ORTC) API for WebRTC Specification, published by the Object-RTC API Community Group under the W3C Community Contributor License Agreement (CLA). A human-readable summary is available.

## Abstract

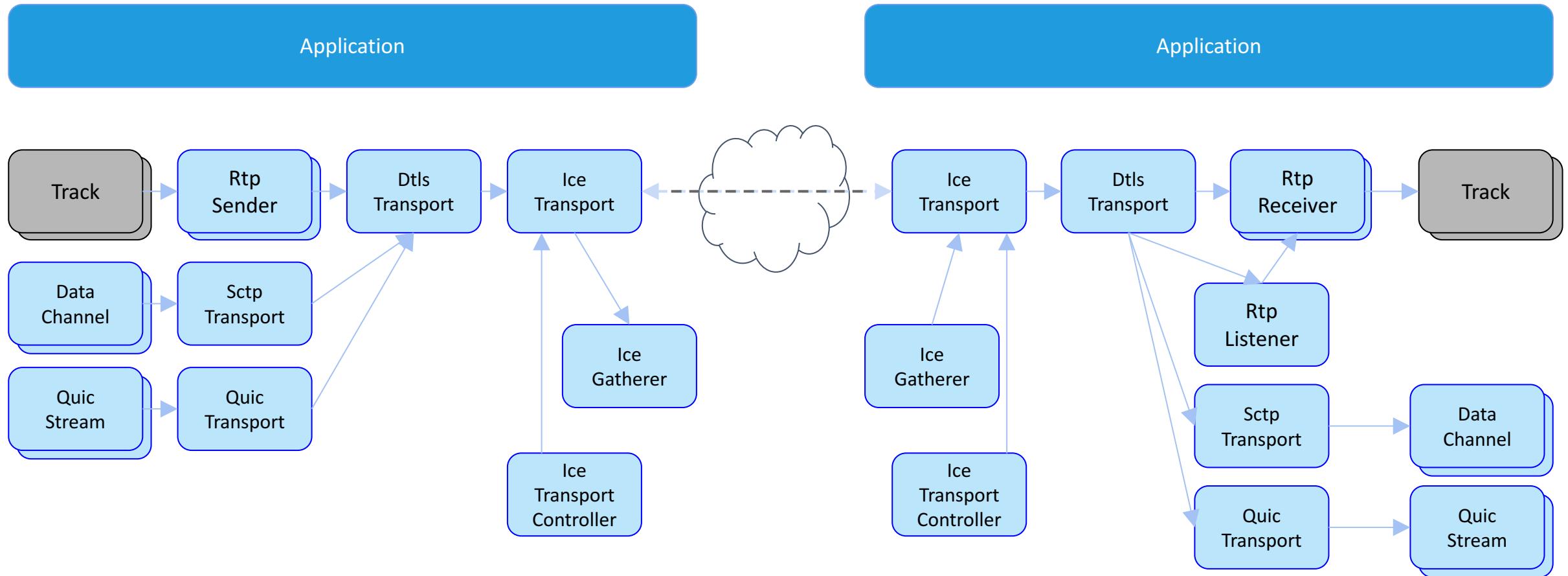
This document defines a set of ECMAScript APIs in WebIDL to allow media to be sent and received from another browser or device implementing the appropriate set of real-time protocols. However, unlike the WebRTC 1.0 API, Object Real-Time Communications (ORTC) does not utilize Session Description Protocol (SDP) in the API, nor does it mandate support for the Offer/Answer state machine (though an application is free to choose SDP and Offer/Answer as an on-the-wire signaling mechanism). Instead, ORTC uses "sender", "receiver" and "transport" objects, which have "capabilities" describing what they are capable of doing, as well as "parameters" which define what they are configured to do. "Tracks" are encoded by senders and sent over transports, then decoded by receivers while "data channels" are sent over transports directly.

## Status of This Document

This specification was published by the [Object-RTC API Community Group](#). It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the [W3C Community Contributor License Agreement \(CLA\)](#) there is a limited opt-out and other conditions apply. Learn more about [W3C Community and Business Groups](#).

If you wish to make comments regarding this document, please send them to [public-ortc@w3.org](mailto:public-ortc@w3.org) ([subscribe](#), [archives](#)).

# ORTC Object Model

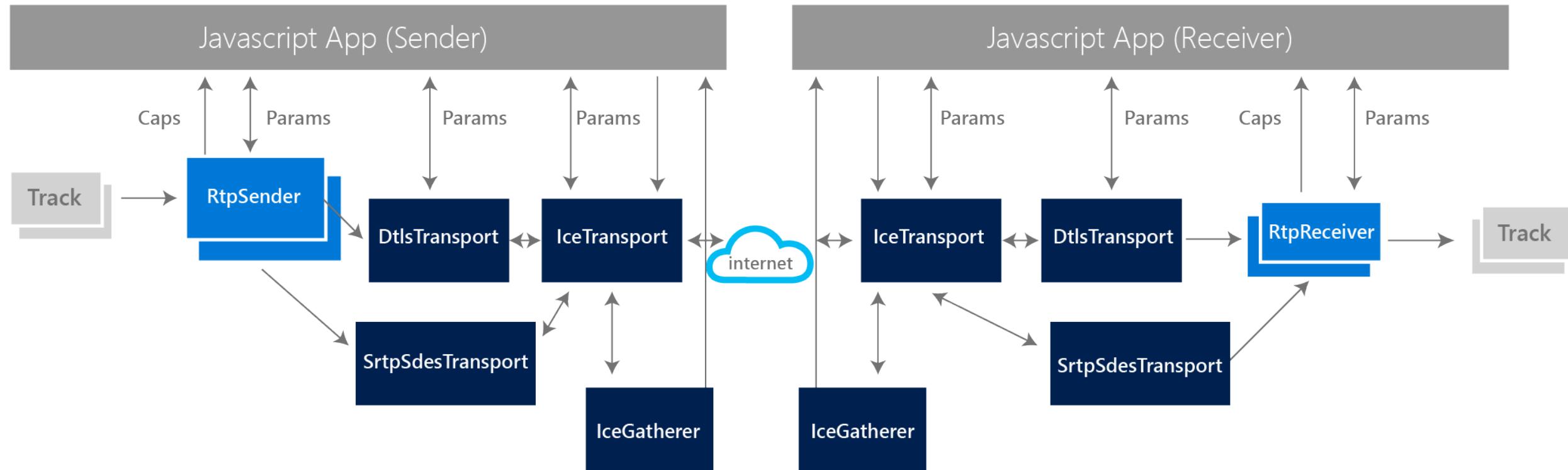


# Important Objects in ORTC

- IceGatherer\*
- IceTransport / IceTransportController
- DtlsTransport
- QuicTransport
- RtpSender / RtpReceiver
- RtpListener
- DataChannel
- SctpTransport

\* all objects are prefixed with “RTC” in the ORTC API specification

# Microsoft Edge ORTC Object Model



# Differences

- **IceGatherer**
    - Edge: Only one IceTransport per IceGatherer (no forking)
  - **IceTransportController**
    - Edge: Not yet supported, freezing handled via order of IceTransport creation.
  - **RtpListener**
    - Edge: Not supported, unroutable packets dropped.
  - **DataChannel/SctpTransport**
    - Edge: “Under Consideration”
- Implemented in Edge but not in ORTC
- **SrtpSdesTransport**

# Class Poll #1

- Question 1: Who is still awake?
- Question 2: Of those who are still awake: Is it important to support forking? Why?
- Question 3: Of those who are still awake: Is it important to support SRTP/SDES? Why?

# What is in Microsoft Edge RTC?

- API support: ORTC, native WebRTC 1.0
- Audio codecs: G.711, G.722, Opus, SILK, CN, DTMF, RED and external FEC (both with burst loss resilience)
- Video codecs:
  - H.264/AVC and VP8 (on by default)
  - RTX
  - Feedback messages: PLI, Generic NACK, REMB
  - H.264UC + ULPFECUC (ORTC only)
    - Implementation of RFC 6190 with support for simulcast, temporal scalability and MRST transport
    - Forward error correction with burst loss resilience
- Statistics:
  - `getStats()` metrics
  - `msGetStats()` supplemental call statistics: support for session quality, burst loss and FEC performance metrics
- Audio/Video multiplexing (BUNDLE), RTP/RTCP mux
- IPv4 and IPv6 (with “happy eyeballs” support)
- Support for DTLS 1.2, SRTP/SDES transports

# Limitations of Edge RTC

- No data channel
- Screen sharing (in development)
- RTP/RTCP mux required with DTLS/SRTP
  - Non-mux supported for SRTP/SDES
- No forking support
- No IceGatherer state attribute or state change events
  - null event.candidate used to indicate end-of-candidates
- No unhandled RTP or SSRC conflict events
- send() and receive() methods may only be called once
  - New RtpSender/RtpReceiver needed to change parameters
- No support for certificate management
  - RSA certificates generated by default
  - Can validate both RSA and ECDSA certificates

# RTCIceGatherer

- Gathers host, reflexive, and relay ICE candidates
- Gathers for both UDP and TCP protocols
- Has a local ICE usernameFragment / password
- Listens for incoming ICE, SRTP, and DTLS packets
- Routes ICE packets to **RTCIceTransport(s)** based on remote ICE usernameFragments (i.e. supports forking)
- Routes SRTP / DTLS packets to **RTCIceTransport(s)** based on 5-tuple IP pairings for confirmed ICE paths

# Interface Definition

## WebIDL

```
[Constructor(RTCIceGatherOptions options)]
interface RTCIceGatherer : RTCStatsProvider {
    readonly attribute RTCIceComponent component;
    readonly attribute RTCIceGathererState state;
    void close();
    void gather(optional RTCIceGatherOptions options);
    RTCIceParameters getLocalParameters();
    sequence<RTCIceCandidate> getLocalCandidates();
    RTCIceGatherer createAssociatedGatherer();
    attribute EventHandler onstatechange;
    attribute EventHandler onerror;
    attribute EventHandler onlocalcandidate;
};
```

# Interface Definition (cont'd)

## WebIDL

```
dictionary RTCIceParameters {
    DOMString usernameFragment;
    DOMString password;
    boolean iceLite;
};
```

## WebIDL

```
enum RTCIceProtocol {
    "udp",
    "tcp"
};
```

## WebIDL

```
typedef (RTCIceCandidate or RTCIceCandidateComplete) RTCIceGatherCandidate;
```

## WebIDL

```
dictionary RTCIceCandidate {
    required DOMString foundation;
    required unsigned long priority;
    required DOMString ip;
    required RTCIceProtocol protocol;
    required unsigned short port;
    required RTCIceCandidateType type;
    required RTCIceTcpCandidateType tcpType;
    DOMString relatedAddress;
    unsigned short relatedPort;
};
```

RTCIceCandidateComplete is a dictionary signifying that all RTCIceCandidates are gathered.

## WebIDL

```
dictionary RTCIceCandidateComplete {
    boolean complete = true;
};
```

# Interface Definition (cont'd)

## WebIDL

```
dictionary RTCIceGatherOptions {
    RTCIceGatherPolicy gatherPolicy;
    sequence<RTCIceServer> iceServers;
};
```

## WebIDL

```
enum RTCIceGatherPolicy {
    "all",
    "nohost",
    "relay"
};
```

## WebIDL

```
enum RTCIceCredentialType {
    "password",
    "oauth"
};
```

## WebIDL

```
enum RTCIceGathererState {
    "new",
    "gathering",
    "complete",
    "closed"
};
```

## WebIDL

```
dictionary RTCOAuthCredential {
    required DOMString macKey;
    required DOMString accessToken;
};
```

# RTCIceGatherer Interface

- ICE Candidates obtained via onlocalcandidate EventHandler (one candidate at a time) or via getLocalCandidates() method (multiple candidates at once)
- getLocalParameters(): Retrieves local ICE usernameFragment / password
- Errors
  - error Events are never fatal
  - Can only enter “closed” state by calling close()
- IceGatherer state machine
  - gather() method: transition from “new” to “gathering”
  - close() method: transition to “closed” (terminal state)
  - Transition from “complete” to “gathering” if a new interface comes up

# RTP/RTCP Multiplexing

- For RTP/RTCP multiplexing, only one IceGatherer is needed
- For RTP/RTCP non-mux, construct rtplceGatherer, then create RTCP IceGatherer (which shares same RTCIceGatherOptions)
  - rtplceGatherer = new RTCIceGatherer(gatherOptions);
  - rtcplceGatherer = rtplceGatherer.createAssociatedGatherer();
    - rtplceGatherer.component === “RTP”
    - rtcplceGatherer.component === “RTCP”

# Microsoft Edge: Interface Definition

## WebIDL

```
[Constructor(RTCIceGatherOptions options)]
interface RTCIceGatherer : RTCStatsProvider {
    readonly attribute RTCIceComponent component;
    RTCIceParameters getLocalParameters ();
    sequence<RTCIceCandidate> getLocalCandidates ();
    RTCIceGatherer createAssociatedGatherer ();
    attribute EventHandler? onerror;
    attribute EventHandler? onlocalcandidate;
};
```

## Differences

- Edge IceGatherer gathers upon construction (no gather() method)
- No close() method.
- No onstatechange EventHandler (or state attribute)
- No support for forking (IceGatherer can only be associated with a single IceTransport)
- When ICE gathering is complete, onlocalcandidate emits a null candidate (not RTCIceCandidateComplete)

# When is ICE Gathering Complete?

- In ORTC, completion of local ICE candidate gathering can be determined by:
  - onlocalcandidate providing event.candidate === RTCIceCandidateComplete OR
  - RTCIceGatherer.state === “complete”
  - Calling IceTransport.addRemoteCandidate(RTCIceComplete) tells the IceTransport that the remote peer has completed gathering.
- In Edge, ICE candidate gathering completion can *only* be determined by onlocalcandidate providing event.candidate == null
  - Calling IceTransport.addRemoteCandidate(null) tells the IceTransport that the remote peer has completed gathering.
  - Why? Because emission of a null event.candidate is widely supported in other browsers.

# RTCIceTransport

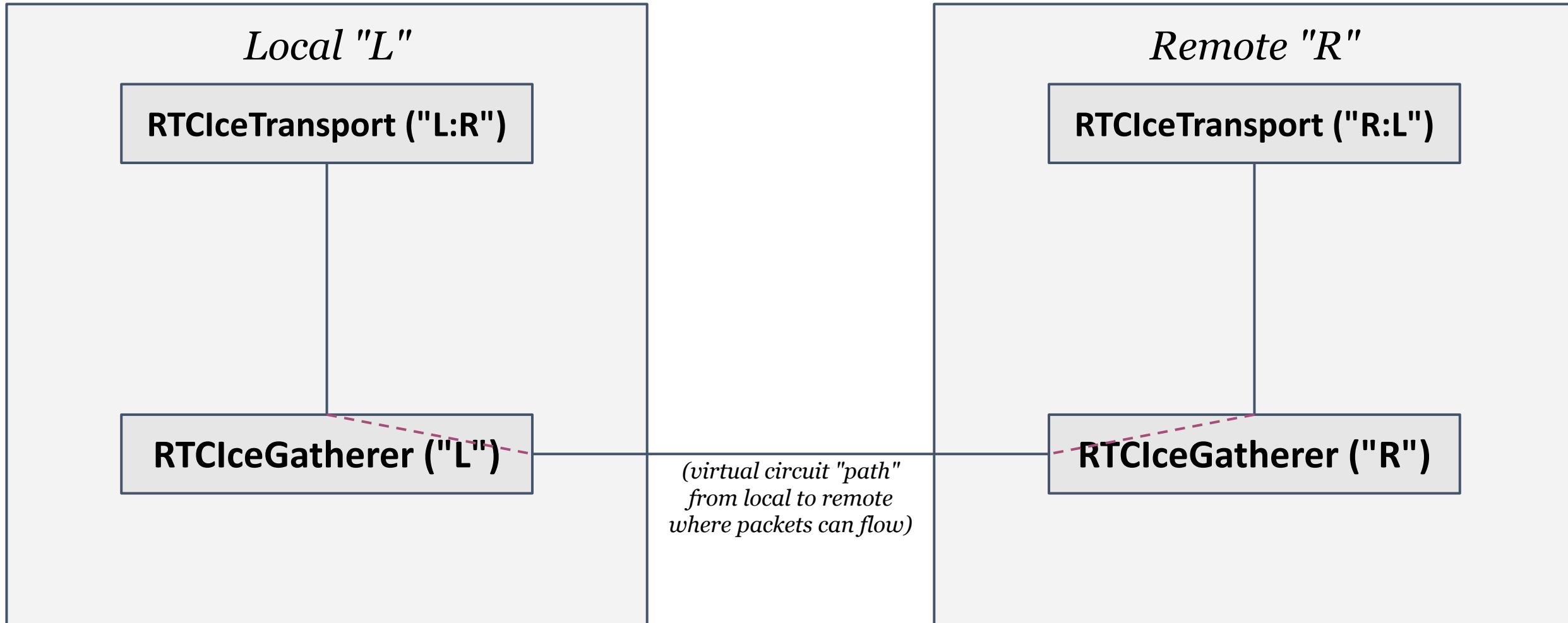
- Associated to a single local IceGatherer.
- Multiple IceTransport objects can share an IceGatherer (forking)
- Sends ICE connectivity tests to test communication paths between a local and remote peer
- Responds to incoming ICE connectivity checks with the specified remote username fragment
- Forms a virtual circuit over which DTLS and SRTP media packets can flow

## Class Poll #2

- Question 1: Who is still awake?
- Question 1: For those still awake: Does it make sense to have separate objects for the IceGatherer and IceTransport?

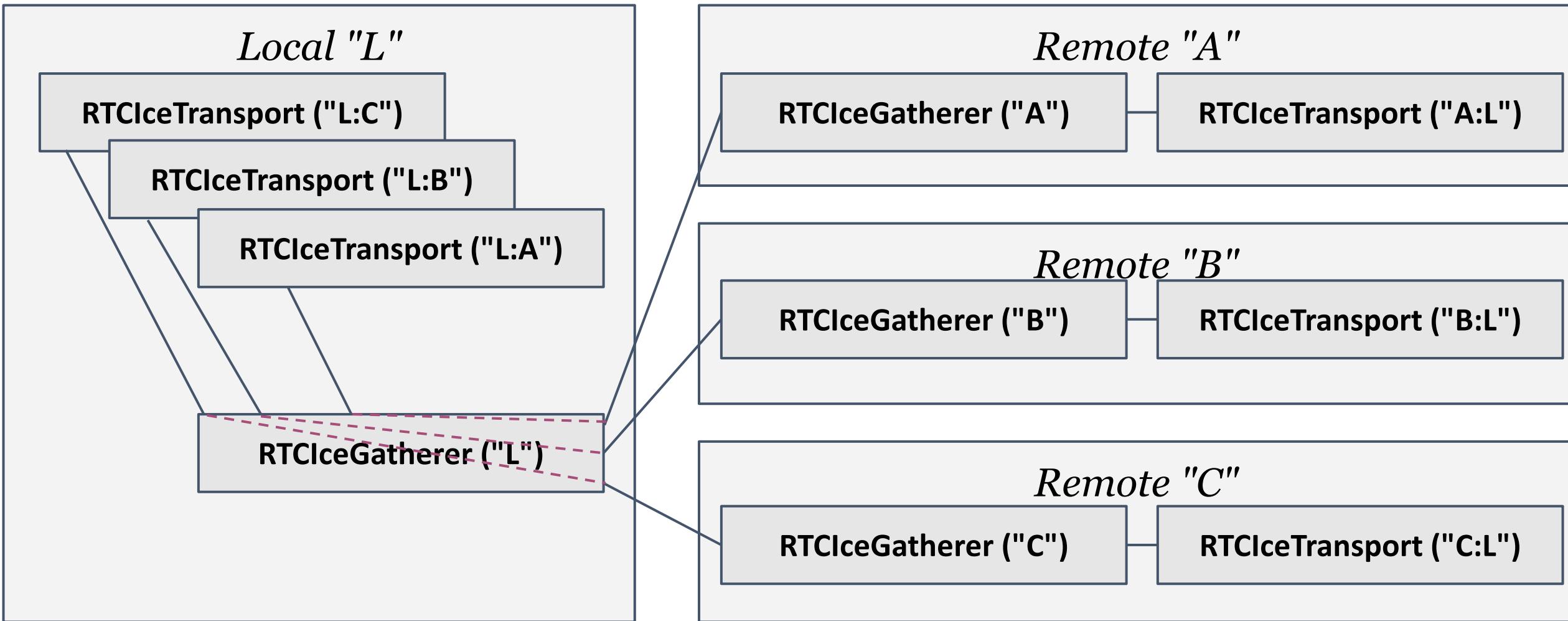
# RTCIceGatherer/RTCIceTransport local / remote object relationships for "1-to-1" scenario

1:1 local IceTransports to local IceGatherer \* 1:1 local to remote IceGatherers \* 1:1 local IceTransport to remote IceTransport



# RTCIceGatherer/RTCIceTransport local / remote object relationships for "forked 1-to-many" scenario

N:1 local IceTransports to local IceGatherer \* 1:N local to remote IceGatherers \* 1:1 local IceTransport to remote IceTransport



# Interface Definition

## WebIDL

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

# Interface Definition (cont'd)

WebIDL

```
enum RTCIceComponent {  
    "RTP",  
    "RTCP"  
};
```

WebIDL

```
enum RTCIceRole {  
    "controlling",  
    "controlled"  
};
```

WebIDL

```
enum RTCIceTransportState {  
    "new",  
    "checking",  
    "connected",  
    "completed",  
    "disconnected",  
    "failed",  
    "closed"  
};
```

# RTCIceTransport Interface

- IceTransport constructed from an (optional) IceGatherer object.
  - If an IceGatherer is not provided in the constructor, it can be provided when the start() method is called.
- Once start() is called, an IceTransport can respond to incoming ICE connectivity checks based on its configured role and can also initiate its own checks.
  - To perform an ICE restart, call iceTransport.start again with a new ICE gatherer. This flushes all remote candidates, so addRemoteCandidate() or setRemoteCandidates() needs to be called again.
- For RTP/RTCP non-mux, construct rtplceTransport, then create RTCP IceTransport
  - var rtcplceTransport = rtplceTransport.createAssociatedTransport()
  - rtplceTransport.component === “RTP”
  - rtcplceTransport.component === “RTCP”

# Microsoft Edge Interface Definition

```
[Constructor(optional RTCIceGatherer gatherer)]
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;
};

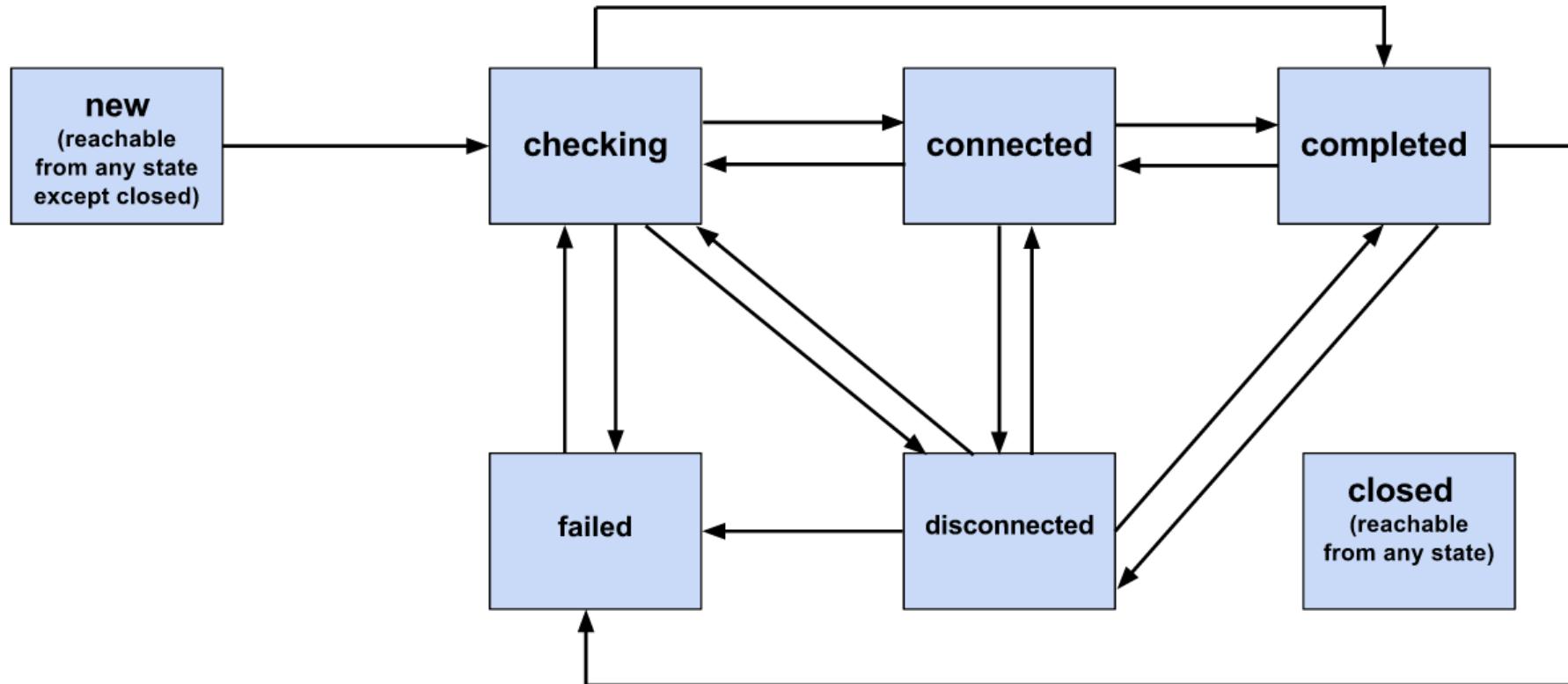
}
```

## Differences

- `getNominatedCandidatePair` instead of `getSelectedCandidatePair()`
- `onicestatechange` instead of `onstatechange` `EventHandler`
- No `oncandidatepairchange` `EventHandler`

# RTCIceTransportState Transitions

[http://draft.ortc.org/#rtcicetransportstate\\*](http://draft.ortc.org/#rtcicetransportstate)



# Forking

- The desire to support forking has had a major impact on the design of ORTC.
- Forking implies that several local IceTransport objects may utilize the same ufrag/password with multiple remote IceTransport objects
  - A local IceTransport is created for each single fork.
  - The need for multiple IceTransport objects to share a ufrag/password combination motivated the introduction of the IceGatherer object.
- Forking implies that several local DtlsTransport objects may utilize the same certificates/fingerprints with multiple remote DtlsTransport objects
  - The need to for multiple DtlsTransport objects to share certificates/fingerprints motivated inclusion of certificates in the DtlsTransport constructor.
- Use cases:
  - Multiple user agents (e.g. mobile device as well as a desk phone).
  - Peer-to-peer signaling over a broadcast medium (e.g. a chat room).
- Forking supported in ORTC Lib, but not in Edge implementation of ORTC.

## **IceGatherer/IceTransport Example**

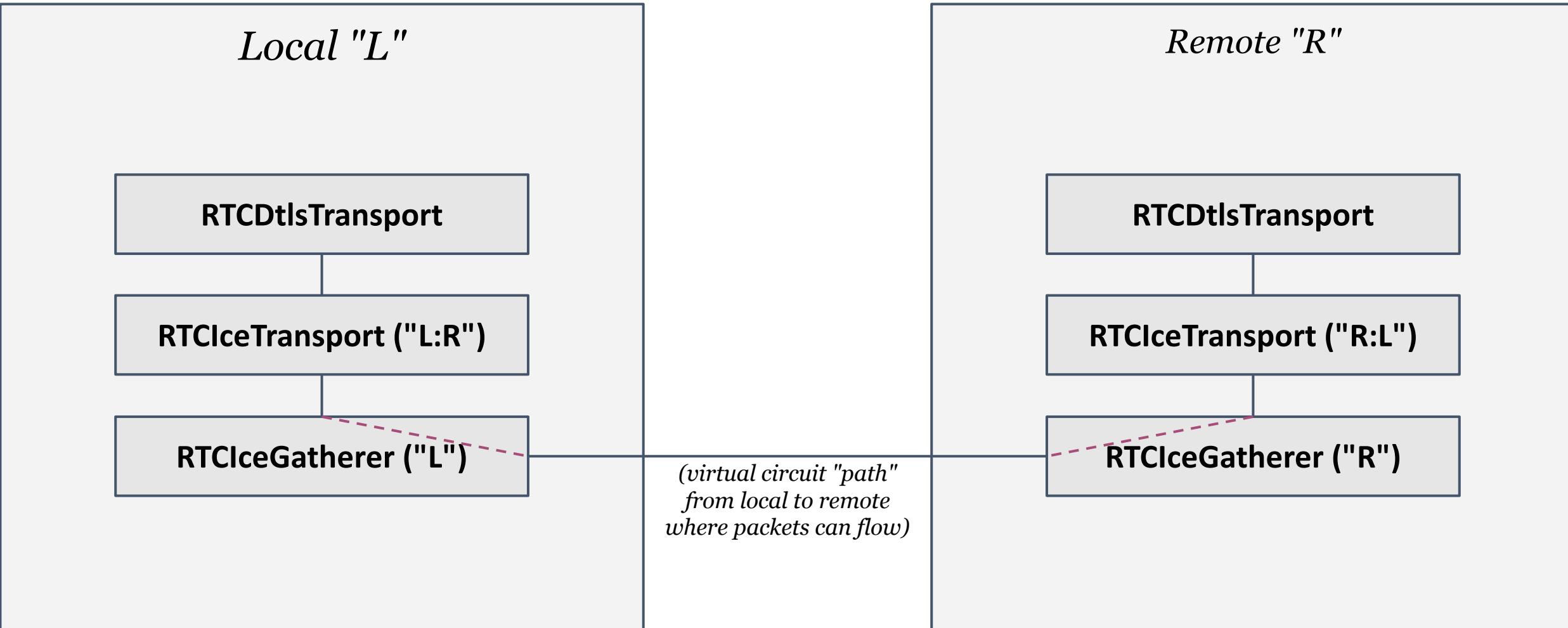
[http://draft.ortc.org/#rtcicegatherer-example\\*](http://draft.ortc.org/#rtcicegatherer-example)

# RTCDtlsTransport

- Supports forking via RTCCertificate interface (to enable forked DTLS transports to reuse the same certificate/fingerprint)
- Derives SRTP keys via DTLS/SRTP exchange
- Encrypts/decrypts data channel packets
- Associated to a single RTCIceTransport
- Sends/receives packets over virtual RTCIceTransport circuit path from local to remote party
- Requires fingerprint validation of DTLS certificate to prevent man-in-the-middle attacks

# RTCIceGatherer / RTCIceTransport / RTCDtlsTransport local/remote object relationships

1:1 local DtlsTransports to local IceTransport \* 1:1 local DtlsTransport to remote DtlsTransport



# Interface Definition

WebIDL

ReSpec

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

# Interface Definition (cont'd)

## WebIDL

```
dictionary RTCDtlsParameters {
    RTCDtlsRole role = "auto";
    sequence<RTCDtlsFingerprint> fingerprints;
};
```

## WebIDL

```
dictionary RTCDtlsFingerprint {
    DOMString algorithm;
    DOMString value;
};
```

## WebIDL

```
enum RTCDtlsTransportState {
    "new",
    "connecting",
    "connected",
    "closed",
    "failed"
};
```

# RTCDtlsTransport Interface

- DtlsTransport constructed from an IceTransport and a sequence of certificates
  - Enables multiple DtlsTransport objects to be constructed with the same certificate(s) to enable forking.
- A newly constructed DtlsTransport MUST listen and respond to incoming DTLS packets before start() is called.
  - After the DTLS handshake completes (but before the remote fingerprint is verified) incoming media packets may be received.
- start() provides the remoteParameters needed to verify the remote fingerprint.
  - Incoming media MUST NOT be rendered prior to completion of remote fingerprint verification, to avoid man-in-the-middle attacks.

# Microsoft Edge: Interface Definition

## WebIDL

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

## Differences

- No certificates passed in the constructor
- No getRemoteCertificates() method
- ondtlsstatechange instead of onstatechange EventHandler
- No “failed” state (transition to “closed” on receipt of a DTLS Alert)

# RTCDtlsRole

## WebIDL

```
enum RTCDtlsRole {  
    "auto",  
    "client",  
    "server"  
};
```

## Enumeration description

**auto** The DTLS role is determined based on the resolved ICE role: the ICE **controlled** role acts as the DTLS client and the ICE **controlling** role acts as the DTLS server.

**client** The DTLS client role.

**server** The DTLS server role.

# RTCCertificate

## WebIDL

```
interface RTCCertificate {
    readonly attribute DOMTimeStamp expires;
    readonly attribute RTCDtlsFingerprint fingerprint;
    AlgorithmIdentifier getAlgorithm();
    static Promise<RTCCertificate> generateCertificate(AlgorithmIdentifier keygenAlgorithm);
};
```

## Notes

- RFC 4572-Update now allows multiple fingerprints per certificate (one of which needs to use the signature hash algorithm).
- WebRTC 1.0 updated to enable <FrozenArray> RTCDtlsFingerprint fingerprints

# Why Mandate Certificates in the Constructor?

- With certificates supplied in the constructor:
  - Asynchronous RTCCertificate.generateCertificate() method used to generate a certificate.
  - After certificate(s) have been generated, DtlsTransport is constructed using the sequence of certificates.
  - Since certificates are pre-constructed, DtlsTransport constructor can return quickly and getLocalParameters() can be synchronous.

# What if Certificates are Not Provided?

- Without certificates supplied in the constructor, a choice emerges:
  - DtlsTransport could create its own certificate(s) during construction.
    - This seems to work for Edge because it only runs on PCs (and envisages moving to generating ECDSA certificates by default in the future)
    - On a low cost mobile processor or in an IoT application, the UI thread could be blocked (> 20 ms), particularly if an RSA-2048 certificate is needed.
  - Alternatively, DtlsTransport constructor could return immediately with a null certificates attribute
    - Certificates/fingerprints guaranteed to be available when getLocalParameters() returns (would need to be async)

## Class Poll #3

- Question 1: Should certificates be passed in the DtlsTransport constructor?
- Question 2: Do you care about SRTP/SDES support in Edge?

## dtlsTransport Example

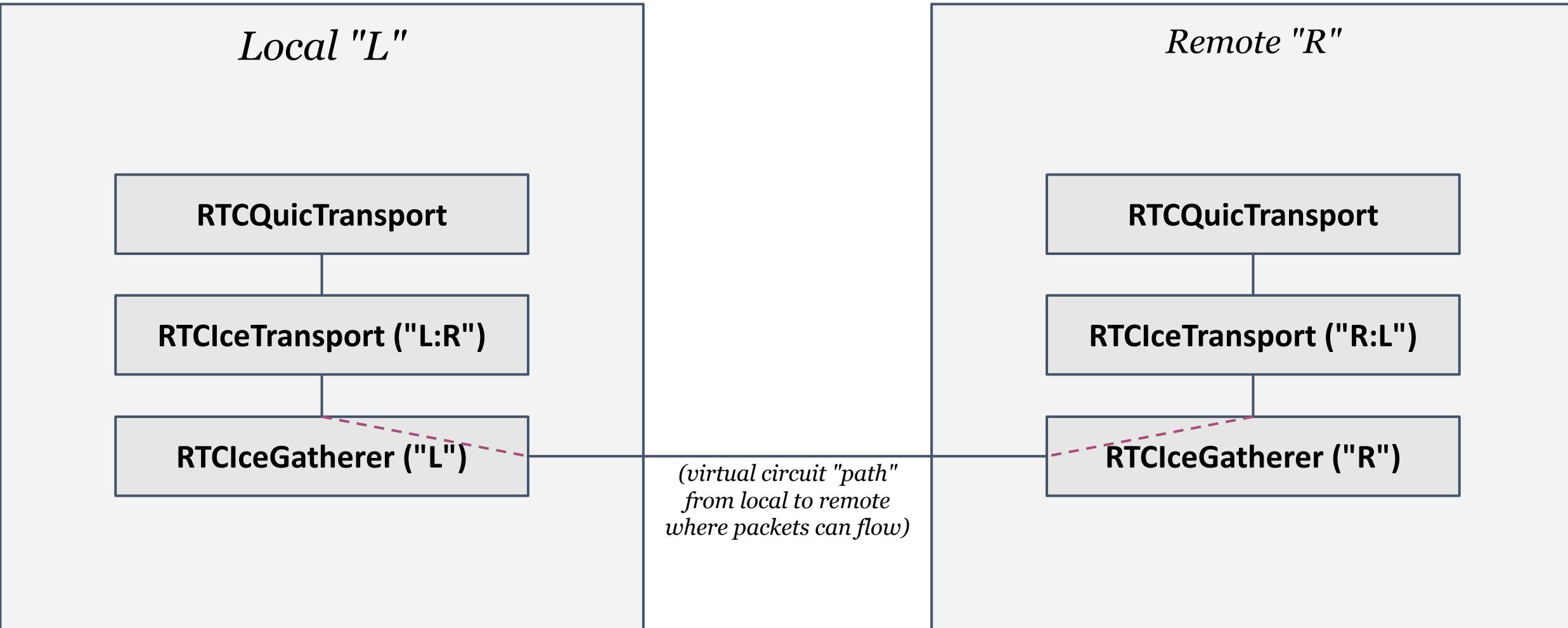
[http://draft.ortc.org/#rtcicetransport-example1\\*](http://draft.ortc.org/#rtcicetransport-example1)

# RTCQuicTransport

- New interface to support QUIC transport under development in the IETF QUIC WG.
- Currently only used for transport of data via QuicStreams.
- Associated to a single RTCIceTransport
- Sends/receives packets over virtual RTCIceTransport circuit path from local to remote party
- Requires fingerprint validation of QUIC certificate to prevent man-in-the-middle attacks

# RTCIceGatherer / RTCIceTransport / RTCQuicTransport local/remote object relationships

1:1 local QuicTransports to local IceTransport \* 1:1 local QuicTransport to remote QuicTransport



# Interface Definition

## WebIDL

```
[Constructor(RTCceTransport transport, sequence<RTCCertificate> certificates)]
interface RTCQuicTransport {
    readonly attribute RTCIceTransport transport;
    readonly attribute RTCQuicTransportState state;
    RTCQuicParameters getLocalParameters();
    RTCQuicParameters? getRemoteParameters();
    sequence<RTCCertificate> getCertificates();
    sequence<ArrayBuffer> getRemoteCertificates();
    void start(RTCQuicParameters remoteParameters);
    void stop();
    attribute EventHandler onstatechange;
    attribute EventHandler onerror;
};
```

# Interface Definition (cont'd)

WebIDL

```
dictionary RTCQuicParameters {
    RTCQuicRole role = "auto";
    sequence<RTCDtlsFingerprint> fingerprints;
};
```

WebIDL

```
dictionary RTCDtlsFingerprint {
    DOMString algorithm;
    DOMString value;
};
```

WebIDL

```
enum RTCQuicTransportState {
    "new",
    "connecting",
    "connected",
    "closed",
    "failed"
};
```

# Interface Definition (cont'd)

## WebIDL

```
enum RTCQuicRole {  
    "auto",  
    "client",  
    "server"  
};
```

## Enumeration description

**auto** The QUIC role is determined based on the resolved ICE role: the ICE **controlled** role acts as the QUIC client and the ICE **controlling** role acts as the QUIC server.

**client** The QUIC client role.

**server** The QUIC server role.

# RTCQuicTransport Interface

- QuicTransport constructed from an IceTransport and a sequence of certificates
  - Enables multiple QuicTransport objects to be constructed with the same certificate(s) to enable forking.
- A newly constructed QuicTransport MUST listen and respond to incoming QUIC packets before start() is called.
  - After the QUIC handshake completes (but before the remote fingerprint is verified) incoming media packets may be received.
- start() provides the remoteParameters needed to verify the remote fingerprint.
  - Incoming media MUST NOT be rendered prior to completion of remote fingerprint verification, to avoid man-in-the-middle attacks.

# RTCSrtpSdesTransport (Edge Only)

- Negotiates SRTP keying via SDES/SRTP
- Associated to a single RTCIceTransport
- Sends/receives packets over virtual RTCIceTransport circuit path from local to remote party
- Supports non-multiplexed as well as multiplexed RTP/RTCP
- Support likely to be phased out at some point

# Microsoft Edge: Interface Definition

```
[Constructor(RTCIceTransport transport, RTCSrtpSdesParameters  
encryptParameters, RTCSrtpSdesParameters decryptParameters)]  
interface RTCSrtpSdesTransport {  
    readonly attribute RTCIceTransport transport;  
    static sequence<RTCSrtpSdesParameters> getLocalParameters ();  
    attribute EventHandler? onerror;  
};
```

# Interface Definition (cont'd)

## WebIDL

```
dictionary RTCsrtpSdesParameters {
    unsigned short tag;
    DOMString cryptoSuite;
    sequence<RTCsrtpKeyParam> keyParams;
    sequence<DOMString> sessionParams;
};
```

## WebIDL

```
dictionary RTCsrtpKeyParam {
    DOMString keyMethod;
    DOMString keySalt;
    DOMString lifetime;
    unsigned short mkiValue;
    unsigned short mkiLength;
};
```

# srtpSdesTransport Example

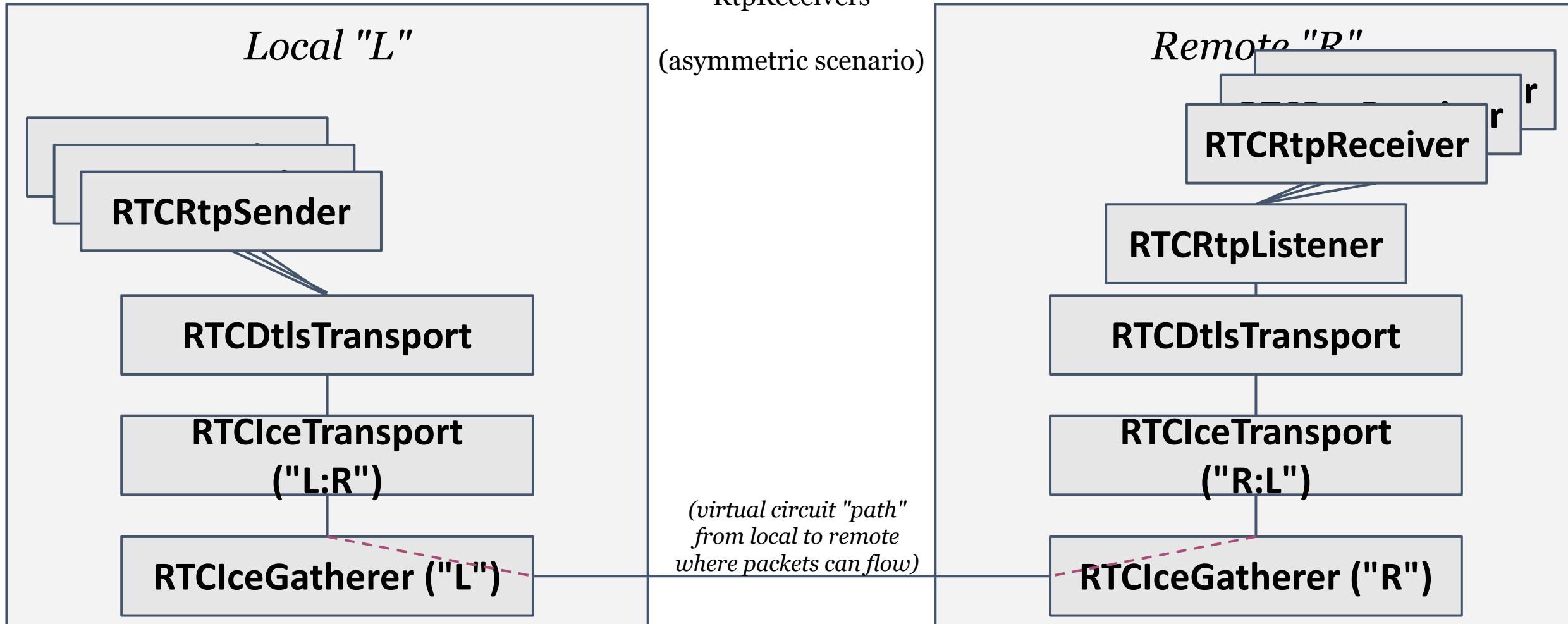
[http://internaut.com:8080/~baboba/ms/msortc-rs.html#rtcsrtpsdestransport-example\\*](http://internaut.com:8080/~baboba/ms/msortc-rs.html#rtcsrtpsdestransport-example*)

# RTCRtpSender / RTCRtpReceiver

- Many senders / receivers can be associated to a single DtlsTransport (BUNDLE)
- Sender takes a single MediaStreamTrack
- Receiver emits a single MediaStreamTrack
- Media "kind" is typically "audio" or "video" ( but could also be "depth")
- Capabilities describe what objects can be configured to do.
- Parameters describe how to encode/decode media on the wire.

# RTCIceGatherer / RTCIceTransport / RTCDtlsTransport RTCRtpSender / Receiver / RTCRtpReceiver local/remote object relationships

N:1 local local RtpSender to DtlsTransports \* 1:1 remote DtlsTransport to remote RtpListener \* 1:N remote RtpListener to remote RtpReceivers



# RtpSender Interface Definition

```
[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);
    Promise<void> setTrack(MediaStreamTrack track);
    Promise<void> replaceTrack(MediaStreamTrack track);
    static RTCRtpCapabilities getCapabilities(DOMString kind);
    Promise<void> send(RTCRtpParameters parameters);
    void stop();
    attribute EventHandler onssrcconflict;
};
```

# RTCRtpSender Interface

- `send(parameters)` method is now asynchronous to allow extended time for determining error conditions.
  - When send was synchronous, the `onerror` EventHandler could fire *after* send had returned (when a problem was discovered later).
- `setTrack()`/`replaceTrack()` method provided to allow seamless switching between input tracks (e.g. switching between a front and backward-facing camera).
- `setTransport()` method enables seamless response to a DTLS or ICE connection failure.
  - Construct new `DtlsTransport(s)` and then call `setTransport()` to send with them.
- No known implementation of the `onssrcconflict` EventHandler

## Class Poll #4

- Question 1: Should send() be synchronous or asynchronous?
- Question 2: Does an SSRC conflict event make sense?

# Microsoft Edge: RtpSender Interface Definition

## WebIDL

```
typedef (RTCDtlsTransport or RTCSrtpSdesTransport) RTCTransport;
```

## WebIDL

```
[Constructor(MediaStreamTrack track, RTCTransport transport)]
interface RTCRtpSender : RTCStatsProvider {
    readonly attribute MediaStreamTrack track;
    readonly attribute RTCTransport transport;
    void setTransport (RTCTransport transport);
    void setTrack (MediaStreamTrack track);
    static RTC RTP Capabilities getCapabilities (optional DOMString kind);
    void send (RTC RTP Parameters parameters);
    void stop ();
    attribute EventHandler? onerror;
};
```

## Differences

- Uses RTCTransport instead of RTCDtlsTransport (for extensibility)
- Only supports non-mux RTP/RTCP with an RTCSrtpSdesTransport
- setTransport, setTrack and send are synchronous, rather than async
- onerror EventHandler
- No onssrcconflict EventHandler

# RtpReceiver Interface Definition

```
[Constructor(DOMString kind, 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
  Promise<void>
  sequence<RTCRtpContributingSource>
  sequence<RTCRtpSynchronizationSource>
  void
    getCapabilities(DOMString kind);
    receive(RTCRtpParameters parameters);
    getContributingSources();
    getSynchronizationSources();
    stop();
};
```

# RTCRtpReceiver Interface

- `receive(parameters)` method is asynchronous to allow extended time for determining error conditions.
  - When receive was synchronous, the `onerror` EventHandler could fire *after* receive had returned to indicate a problem discovered later.
- `setTransport()` method enables seamless response to a DTLS or ICE connection failure.
  - Construct new `DtlsTransport(s)` and then call `setTransport()` to send with them.
- `getContributingSources()` method used for:
  - Determining contributing sources (in a mixed stream): Supported in Edge
  - Determining audio levels (in a mixed stream): Not supported in Edge
- `getSynchronizationSources()` method used for:
  - Determining synchronization sources: Supported in ORTC Lib
  - Determining audio levels (in a P2P stream): Supported in ORTC Lib

# Microsoft Edge: RtpReceiver Interface Definition

## WebIDL

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

## Differences

- Uses RTCTransport instead of RTCDtlsTransport
- Only supports non-mux RTP/RTCP with an RTCSrtpSdesTransport
- setTransport and receive are synchronous, rather than async
- onerror EventHandler

# RTCRtpCapabilities

## WebIDL

```
dictionary RTCRtpCapabilities {
    sequence<RTCRtpCodecCapability> codecs;
    sequence<RTCRtpHeaderExtension> headerExtensions;
    sequence<DOMString> fecMechanisms;
};
```

## WebIDL

```
dictionary RTCRtpCodecCapability {
    DOMString name;
    DOMString mimeType;
    DOMString kind;
    unsigned long clockRate;
    payloadtype preferredPayloadType;
    unsigned long maxptime;
    unsigned long ptime;
    unsigned long numChannels;
    sequence<RTCRtcpFeedback> rtpFeedback;
    Dictionary parameters;
    Dictionary options;
    unsigned short maxTemporalLayers = 0;
    unsigned short maxSpatialLayers = 0;
    boolean svcMultiStreamSupport;
};
```

# RTCRtpCapabilities

- Indicates what codecs, header extensions and FEC mechanisms are supported by RtpSenders and RtpReceivers.
  - For each codec, information is provided, including name/mimeType, clockRate, maxptime/ptime/numChannels (for audio), rtcpFeedback, etc.
    - Parameters and options dictionaries provided for codec-specific capabilities.
  - codecs attribute includes more than just media codecs
    - RED, Retransmission (RTX), DTMF, CN and FEC mechanisms (e.g. “ulpfec”, “flexfec”, etc.) are included as well.
    - One RTX entry for each codec that can be retransmitted.

# Demo

- Object Inventory and Capability dumper:
  - <http://internaut.com:8080/~baboba/iit-tutorial/cap-dumper/>
- Tells you what objects a browser supports
- Dumps RtpSender/RtpReceiver.getCapabilities(*kind*) for “audio” and “video”

# RTCRtpParameters

## WebIDL

```
dictionary RTCRtpParameters {
    DOMString muxId = "";
    required sequence<RTCRtpCodecParameters> codecs;
    sequence<RTCRtpHeaderExtensionParameters> headerExtensions;
    sequence<RTCRtpEncodingParameters> encodings;
    RTCRtcpParameters rtcp;
    RTCDegradationPreference degradationPreference = "balanced";
};
```

## WebIDL

```
dictionary RTCRtpCodecParameters {
    required DOMString name;
    required DOMString  mimeType;
    required payloadtype payloadType;
    required unsigned long clockRate;
    required unsigned long maxptime;
    required unsigned long ptime;
    required unsigned long numChannels;
    sequence<RTCRtcpFeedback> rtcpFeedback;
    Dictionary parameters;
};
```

# RTCRtpParameters

- Indicates what codecs, header extensions, encodings, rtcp settings, etc. are to be used for sending and receiving.
  - Codec parameters resemble codec capabilities.
    - Parameters dictionaries are provided for codec-specific settings.
  - codecs attribute includes more than just media codecs
    - RED, Retransmission (RTX), DTMF, CN and FEC mechanisms (e.g. “ulpfec”, “flexfec”, etc.) are included as well.
    - An RTX entry is provided for each codec that supports retransmission.

# Capabilities Exchange

- RTCRtpCapabilities designed for “capabilities exchange”
  - Peers exchange sender/receiver capabilities (including preferred Payload Types)
  - Capabilities used to compute RTCRtpParameters passed to send(*parameters*) and receive(*parameters*)
  - Intersection of codecs, header extensions, feedback messages can be computed without specific knowledge
    - sending codecs, header extensions and feedback messages computed from intersection of local sender capabilities and remote receiver capabilities.
    - receiving codecs, header extensions and feedback messages computed from intersection of local receiver capabilities and remote sender capabilities.

# Capabilities Exchange (cont'd)

- Determining codec parameters is trickier.
- Several important sender codec parameters can be determined from receiver codec capabilities. Examples:
  - Opus: receiver useinbandfec capability copied to sender useinbandfec parameter
  - H.264/AVC: receiver profile-level-id capability copied to sender profile-level-id parameter
  - VP8/VP9: receiver max-fr/max-fs capability is copied to sender max-fr/max-fs parameter

# Class Poll #5

- Question 1: Does “capabilities exchange” signaling make sense?

RtpSender/RtpReceiver Example

[http://internaut.com:8080/~baboba/ms/msortcrs.html#rtcrtpreceiver-example\\*](http://internaut.com:8080/~baboba/ms/msortcrs.html#rtcrtpreceiver-example*)

# RTCRtpEncodingParameters

## WebIDL

```
dictionary RTCRtpEncodingParameters {
    unsigned long          ssrc;
    payloadtype            codecPayloadType;
    RTCRtpFecParameters   fec;
    RTCRtpRtxParameters   rtx;
    RTPriorityType         priority;
    unsigned long           maxBitrate;
    double                 resolutionScale;
    double                 framerateScale;
    unsigned long           maxFramerate;
    boolean                active = true;
    DOMString               encodingId;
    sequence<DOMString>   dependencyEncodingIds;
};
```

Attribute	Type	Receiver/Sender
<u>ssrc</u>	unsigned long	Receiver/Sender
<u>codecPayloadType</u>	payloadType	Receiver/Sender
<u>fec</u>	RTCRtpFecParameters	Receiver/Sender
<u>rtx</u>	RTCTransportFeedbackParameters	Receiver/Sender
<u>priority</u>	RTPriorityType	Sender
<u>maxBitrate</u>	unsigned long	Sender
<u>resolutionScale</u>	double	Sender
<u>framerateScale</u>	double	Sender
<u>maxFramerate</u>	unsigned long	Sender
<u>active</u>	boolean	Receiver/Sender
<u>encodingId</u>	DOMString	Receiver/Sender
<u>dependencyEncodingIds</u>	sequence<DOMString>	Receiver/Sender

# Encoding Parameters (receiver/sender)

- *ssrc* (if provided) specifies the specific SSRC to be used for the encoding. If omitted when sending, the browser chooses.
- *codecPayloadType* refers to a codec included in the codecs sequence. If omitted, the first codec in the sequence is implied.
- *fec* indicates whether this encoding is protected with forward error correction (and if so, what ssrc is used for FEC).
- *rtx* indicates whether this encoding is retransmitted (and if so, what ssrc is used for retransmission).

# Encoding Parameters (receiver/sender)

- *active* indicates whether this encoding is being sent or received.
  - Can be used to implement “hold” scenarios.
- *encodingId* indicates the RID to be used for this encoding.
- *dependencyEncodingIds* indicate the other encodings that this encoding depends on (for scalable video coding only)

# Encoding Parameters (sender only)

- *priority* indicates how QoS marking is to be applied for this encoding.
  - No implementations of this parameter.
- *maxBitrate* indicates the maximum bitrate which this encoding can use.
- *resolutionScale* indicates the how much the video width/height should be decreased, compared with `sender.track`.
  - 2.0 means send at half width/height.
  - Used in simulcast and scalable video coding.

# Encoding Parameters (sender cont'd)

- *framerateScale* indicates the how much the framerate should be decreased, compared with sender.track.
  - 2.0 means send at half the framerate.
  - Primarily for temporal scalability.
  - No implementations of this parameter.
- *maxFramerate* indicates the maximum framerate to be used for this encoding.
  - Usage scenario: simulcast screen sharing.
  - No implementations of this parameter.

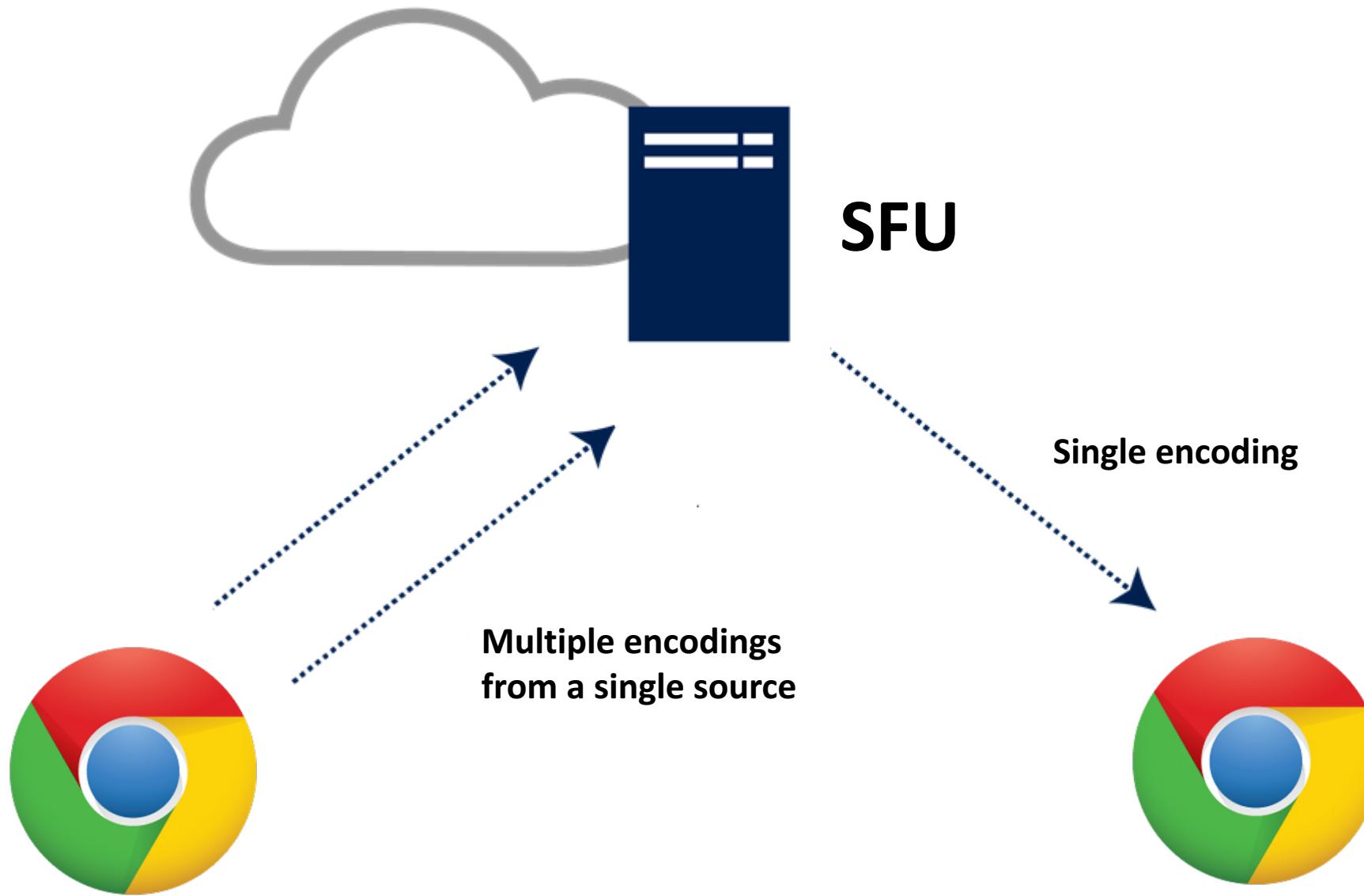
# Class Poll #6

- Question 1: Who is still awake?
- Question 2: Of those still awake, how many know what simulcast and scalable video coding is?

# Simulcast

- Simulcast involves sending (and possibly receiving) multiple streams that differ in characteristics such as resolution or frame rate.
- Typical scenario:
  - Browser sends multiple streams (e.g. a high resolution and a low resolution stream) to a Selective Forwarding Unit (SFU).
  - SFU selects which stream to forward to each conference participant.
    - Mobile device receives the low resolution stream
    - PC receives the high resolution stream
- Both WebRTC 1.0 and ORTC object models support sending multiple streams.
- ORTC also supports receiving multiple streams (and outputting a single track).

# Typical Simulcast Scenario

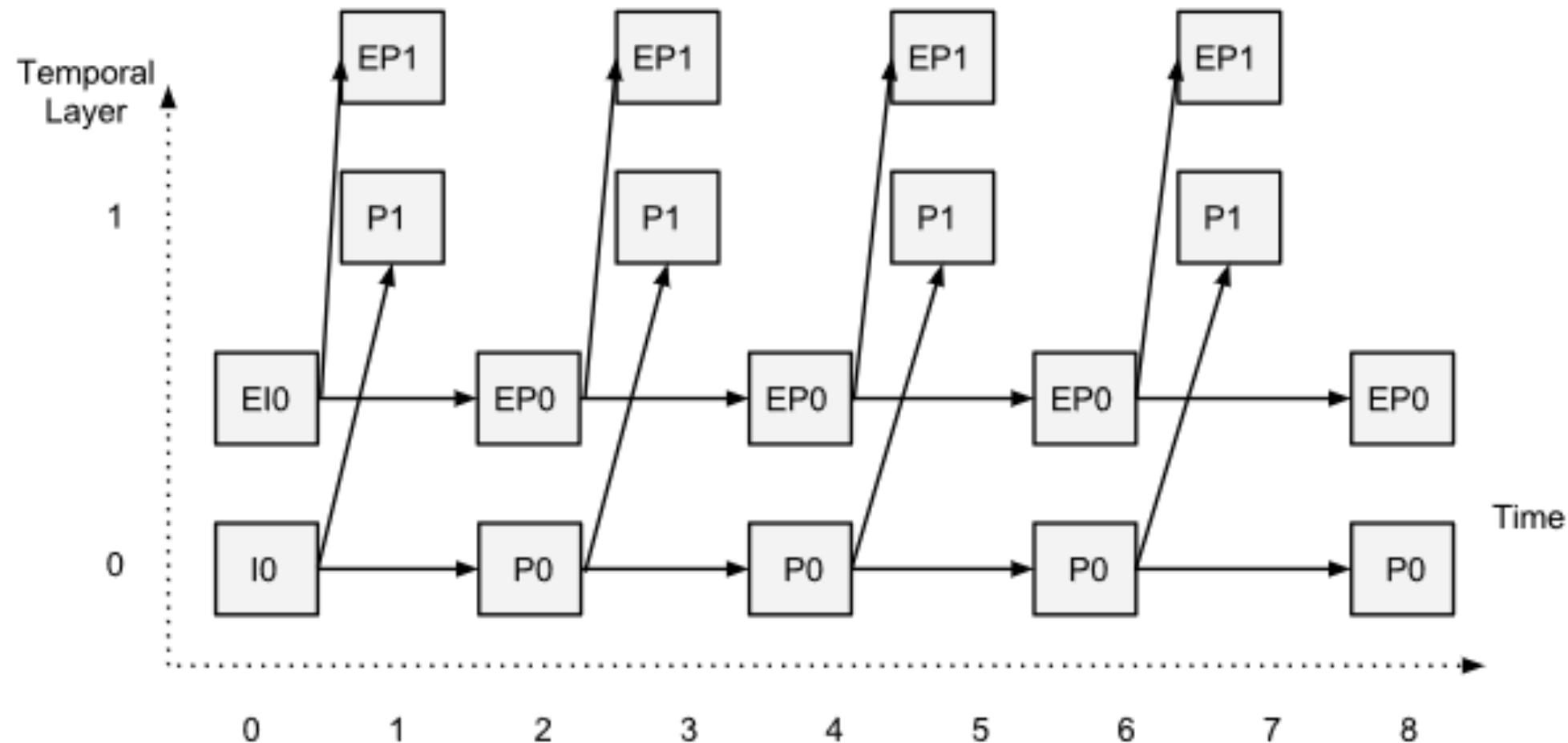


# Scalable Video Coding

- Codecs implemented in WebRTC support scalable video coding (SVC):
  - VP8: temporal scalability
  - VP9: temporal and spatial scalability
  - H.264/SVC: temporal scalability
- SVC now mainstream
  - Utilized in many products: Google (Hangouts), Microsoft (Skype), Vidyo, Polycom, Avaya/RADVISION, LifeSize, etc.
- ORTC makes it possible to enable SVC in the RtpSender/encoder
  - Developer can specify the number of layers but typically just wants the browser to “do the right thing” (send as many layers as conditions permit).
  - As long as the decoder supports SVC, no need to configure the RtpReceiver.

# Simulcast + Scalable Video Coding

(2 layers spatial simulcast + 2 layers temporal scalability)



# 2-layer simulcast + 2 layers temporal scalability

```
var encodings = [ {  
    // Low resolution base layer (half the input framerate, half the input resolution)  
    encodingId: "0",  
    resolutionScale: 2.0,  
    framerateScale: 2.0  
}, {  
    // High resolution Base layer (half the input framerate, full input resolution)  
    encodingId: "E0",  
    resolutionScale: 1.0,  
    framerateScale: 2.0  
}, {  
    // Temporal enhancement to the low resolution base layer (full input framerate, half resolution)  
    encodingId: "1",  
    dependencyEncodingIds: ["0"],  
    resolutionScale: 2.0,  
    framerateScale: 1.0  
}, {  
    // Temporal enhancement to the high resolution base layer (full input framerate and resolution)  
    encodingId: "E1",  
    dependencyEncodingIds: ["E0"],  
    resolutionScale: 1.0,  
    framerateScale: 1.0  
} ];
```

# RTCRtpListener

- Routes incoming media packets to correct RTCRtpReceiver
- Emits "unknown" media event when no RTCRtpReceiver attached that handles the incoming media
- RTCRtpListener routing engine exists whether it is created explicitly or not if an RTCRtpReceiver is used

# Interface Definition

## WebIDL

```
[Constructor(RTCDtlsTransport transport)]
interface RTC Rtp Listener {
    readonly attribute RTCDtlsTransport transport;
    attribute EventHandler onunhandledrtp;
};
```

## WebIDL

```
[Constructor(DOMString type, RTC Rtp Unhandled Event Init eventInitDict)]
interface RTC Rtp Unhandled Event : Event {
    readonly attribute DOMString muxId;
    readonly attribute DOMString rid;
    readonly attribute payloadtype payloadType;
    readonly attribute unsigned long ssrc;
};
```

# Class Poll #7

- Question 1: Is the RtpListener a good idea?
- Question 2: Do you want to hear about the DataChannel and IceTransportController objects or take a break?

# RTP Matching Rules

## (draft-ietf-mmusic-sdp-bundle-negotiation Section 10.2)

To prepare for demultiplexing RTP/RTCP packets to the correct "m=" line, the following steps MUST be followed for each BUNDLE group.

Construct a table mapping MID to "m=" line for each "m=" line in this BUNDLE group. Note that an "m=" line may only have one MID.

Construct a table mapping incoming SSRC to "m=" line for each "m=" line in this BUNDLE group and for each SSRC configured for receiving in that "m=" line.

Construct a table mapping outgoing SSRC to "m=line" for each "m=" line in this BUNDLE group and for each SSRC configured for sending in that "m=" line.

Construct a table mapping payload type to "m=" line for each "m=" line in the BUNDLE group and for each payload type configured for receiving in that "m=" line. If any payload type is configured for receiving in more than one "m=" line in the BUNDLE group, do not include it in the table, as it cannot be used to uniquely identify a "m=" line.

Note that for each of these tables, there can only be one mapping for any given key (MID, SSRC, or PT). In other words, the tables are not multimap.

As "m=" lines are added or removed from the BUNDLE groups, or their configurations are changed, the tables above MUST also be updated.

# RTP Matching Rules (cont'd)

For each RTP packet received, the following steps MUST be followed to route the packet:

If the packet has a MID and that MID is not in the table mapping MID to RtpReceiver, fire an rtpunhandledevent.

If the packet has a MID, and the packet's extended sequence number is greater than that of the last MID update, update the incoming SSRC mapping table to include an entry that maps the packet's SSRC to the "m=" line for that MID.

If the packet has a MID and that MID is in the table mapping MID to RtpReceiver, update the SSRC mapping table to include an entry mapping the packet's SSRC to the RtpReceiver.

If the packet's SSRC is in the SSRC mapping table, route the packet to the mapped RtpReceiver and stop.

If the packet's payload type is in the payload type table, update the the SSRC mapping table to include an entry mapping the packet's SSRC to the RtpReceiver. Deliver the packet to the RtpReceiver and stop.

Otherwise, fire an rtpunhandledevent.

## Class Poll #8

- Question 1: Can multiple receivers be set up to receive RTP packets with the same payload types?
- Question 2: Does the Payload Type table make sense?

# RTCDatChannel

- Very similar to data channel API in WebRTC 1.0 (intentional)
- Represents a bi-directional data channel between two peers.
- Enables sending data over an RTCDatTransport.

# Interface Definition

## WebIDL

```
[Constructor(RTCDataTransport transport, RTCDataChannelParameters parameters)]
interface RTCDataChannel : EventTarget {
    readonly attribute RTCDataTransport transport;
    readonly attribute RTCDataChannelState readyState;
    readonly attribute unsigned long bufferedAmount;
    attribute unsigned long bufferedAmountLowThreshold;
    attribute DOMString binaryType;
    RTCDataChannelParameters getParameters();
    void close();
    attribute EventHandler onopen;
    attribute EventHandler onbufferedamountlow;
    attribute EventHandler onerror;
    attribute EventHandler onclose;
    attribute EventHandler onmessage;
    void send(USVString data);
    void send(Blob data);
    void send(ArrayBuffer data);
    void send(ArrayBufferView data);
};
```

# Interface Definition (cont'd)

## WebIDL

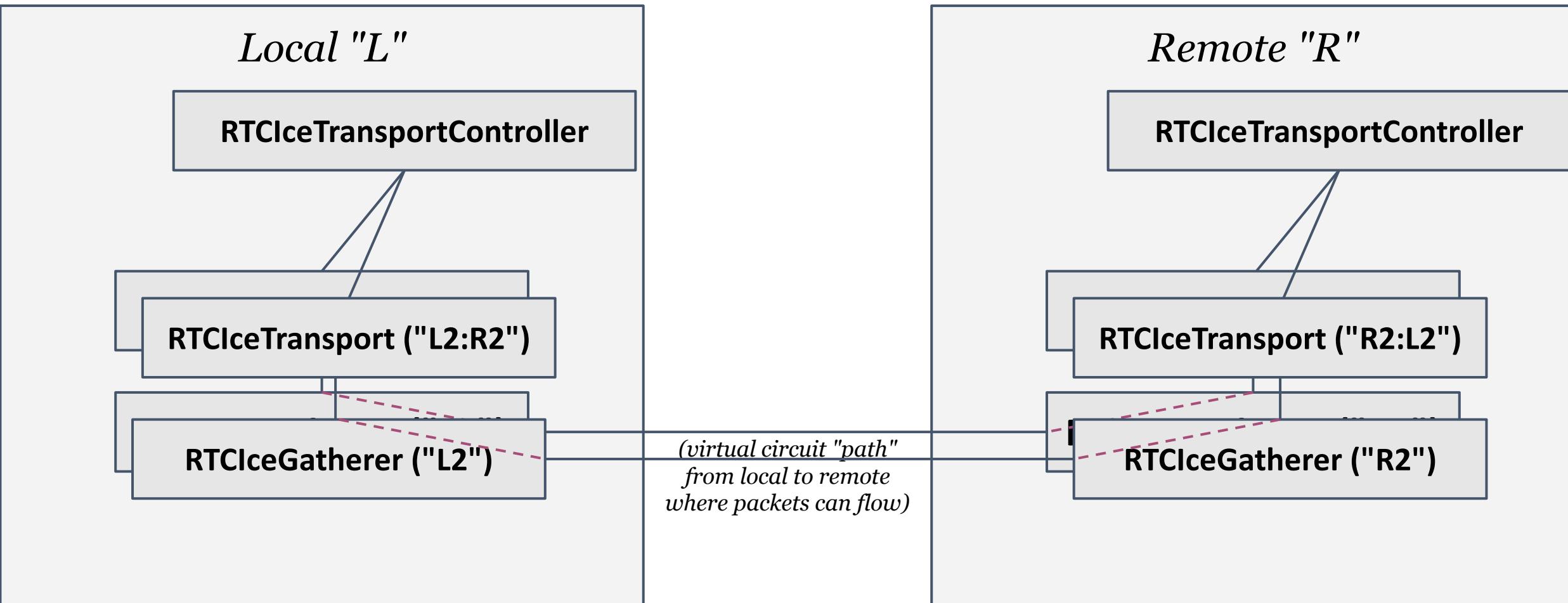
```
[Constructor(RTCDtlsTransport transport, optional unsigned short port)]
interface RTCSctpTransport : RTCDATATransport {
    readonly attribute RTCDtlsTransport transport;
    readonly attribute RTCSctpTransportState state;
    readonly attribute unsigned short port;
    static RTCSctpCapabilities getCapabilities();
    void start(RTCSctpCapabilities remoteCaps);
    void stop();
    attribute EventHandler ondatachannel;
    attribute EventHandler onstatechange;
};
```

# RTCIceTransportController

- Coordinates ICE freezing across IceTransports
- Manages collective bandwidth across IceTransports
- Contains 1 or more relationships to IceTransports
- Used mostly for when audio and video tracks are streamed non-muxed on different virtual IceTransport circuits

# RTCIceTransport / RTCIceTransportController local/remote object relationships

RTPIceTransportController does not route packets. It's not in the media pipeline.  
It coordinates the RTCIceTransports ICE freezing.



# Interface Definition

## WebIDL

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

# Questions?