Object RTC (ORTC) API for WebRTC

Draft Community Group Report 22 June 2015

Editor:

Robin Raymond, Hookflash

Authors:

Bernard Aboba, Microsoft Corporation

Justin Uberti, Google

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.

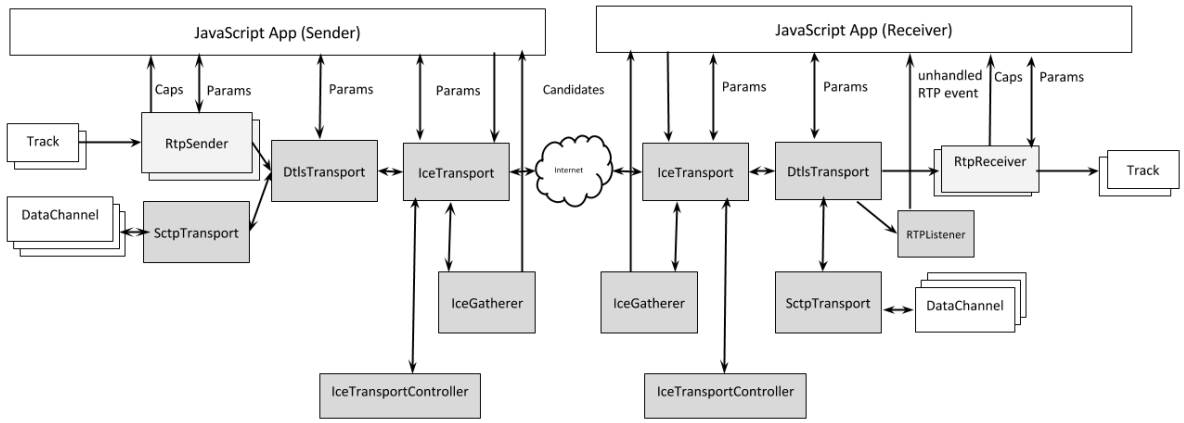
If you wish to make comments regarding this document, please send them to public-ortc@w3.org (subscribe, archives).

Table of Contents

1. Overview

Object Real-Time Communications (ORTC) provides a powerful API for the development of WebRTC based applications. 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.

In a Javascript application utilizing the ORTC API, the relationship between the application and the objects, as well as between the objects themselves is shown below. Horizontal or slanted arrows denote the flow of media or data, whereas vertical arrows denote interactions via methods and events.



In the figure above, the **RTCRtpSender** (Section 5) encodes the track provided as input, which is transported over a **RTCDtlsTransport** (Section 4). An **RTCDataChannel** (Section 11) utilizes an **RTCSctpTransport** (Section 12) which can also be multiplexed over the **RTCDtlsTransport**. Sending of Dual Tone Multi Frequency (DTMF) tones is supported via the **RTCDtmfSender** (Section 10).

The **RTCDtlsTransport** utilizes an **RTCIceTransport** (Section 3) to select a communication path to reach the receiving peer's **RTCIceTransport**, which is in turn associated with an **RTCDtlsTransport** which de-multiplexes media to the **RTCRtpReceiver** (Section 6) and data to the **RTCSctpTransport** and **RTCDataChannel**. The **RTCRtpReceiver** then decodes media, producing a track which is rendered by an audio or video tag.

Several other objects also play a role. The **RTCIceGatherer** (Section 2) gathers local ICE candidates for use by one or more **RTCIceTransport** objects, enabling forking scenarios. The **RTCIceTransportController** (Section 7) manages freezing/unfreezing (defined in [RFC5245]) and bandwidth estimation. The **RTCRtpListener** (Section 8) detects whether an RTP stream is received that cannot be delivered to any existing **RTCRtpReceiver**, providing an `onunhandledrtp` event handler that the application can use to correct the situation.

Remaining sections of the specification fill in details relating to RTP capabilities and parameters, operational statistics, media authentication via Identity Providers (IdP) and compatibility with the WebRTC 1.0 API. RTP dictionaries are described in Section 9, the Statistics API is described in Section 13, the Identity API is described in Section 14, an event summary is provided in Section 15, and WebRTC 1.0 compatibility issues are discussed in Section 16.

1.1 Terminology

The `EventHandler` interface represents a callback used for event handlers as defined in [HTML5].

The concepts ***queue a task*** and ***fires a simple event*** are defined in [HTML5].

The terms ***event***, ***event handlers*** and ***event handler event types*** are defined in [HTML5].

The terms ***MediaStream*** and ***MediaStreamTrack*** are defined in [GETUSERMEDIA].

For Scalable Video Coding (SVC), the terms single-session transmission (***SST***) and multi-session transmission (***MST***) are defined in [RFC6190]. This specification only supports SST but not MST. The term Single Real-time transport protocol stream Single Transport (***SRST***), defined in [GROUPING] Section 3.7, refers to an SVC implementation that transmits all layers within a single transport, using a single Real-time Transport Protocol (RTP) stream and synchronization source (SSRC). The term Multiple RTP stream Single Transport (***MRST***), also defined in [GROUPING] Section 3.7, refers to an implementation that transmits all layers within a single transport, using multiple RTP streams with a distinct SSRC for each layer.

2. The RTCIceGatherer Object

The ***RTCIceGatherer*** gathers local host, server reflexive and relay candidates, as well as enabling the retrieval of local Interactive Connectivity Establishment (ICE) parameters which can be exchanged in signaling. By enabling an endpoint to use a set of local candidates to construct multiple **`RTCIceTransport`** objects, the **`RTCIceGatherer`** enables support for scenarios such as parallel forking.

2.1 Overview

An **`RTCIceGatherer`** instance can be associated to multiple **`RTCIceTransport`** objects. The **`RTCIceGatherer`** does not prune local candidates until at least one **`RTCIceTransport`** object has become associated and all associated **`RTCIceTransport`** objects are in the "completed" or "failed" state.

As noted in [RFC5245] Section 7.1.2.2, an incoming connectivity check contains an `ICE-CONTROLLING` or `ICE-CONTROLLED` attribute, depending on the role of the ICE agent initiating the check. Since an **`RTCIceGatherer`** object does not have a role, it cannot determine whether to respond to an incoming connectivity check with a 487 (Role Conflict) error; however, it can

validate that an incoming connectivity check utilizes the correct local username fragment and password, and if not, can respond with an 401 (Unauthorized) error, as described in [RFC5389] Section 10.1.2.

For incoming connectivity checks that pass validation, the **RTCIceGatherer** MUST buffer the incoming connectivity checks so as to be able to provide them to associated **RTCIceTransport** objects so that they can respond.

2.2 Operation

An **RTCIceGatherer** instance is constructed from an **RTCIceGatherOptions** object.

2.3 Interface Definition

WebIDL

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

2.3.1 Attributes

component of type *RTCIceComponent*, readonly
: The component-id of the **RTCIceGatherer**.

onerror of type EventHandler, nullable
: This event handler, of event handler type `error`, MUST be supported by all objects implementing the **RTCIceGatherer** interface. If TURN credentials are invalid, then this event MUST be fired.

`ongathererstatechange` of type EventHandler, nullable

This event handler, of event handler type `gathererstatechange`, uses the **`RTCIceGathererStateChangedEvent`** interface. It MUST be supported by all objects implementing the **`RTCIceGatherer`** interface. It is called any time the **`RTCIceGathererState`** changes.

`onlocalcandidate` of type EventHandler, nullable

This event handler, of event handler event type `icecandidate`, uses the **`RTCIceGathererEvent`** interface. It MUST be supported by all objects implementing the **`RTCIceGatherer`** interface. It receives events when a new local ICE candidate is available. Since ICE candidate gathering begins once an **`RTCIceGatherer`** object is created, `candidate` events are queued until an `onlocalcandidate` event handler is assigned. When the final candidate is gathered, a `candidate` event occurs with an `RTCIceCandidateComplete` emitted.

`state` of type *RTCIceGathererState*, readonly

The current state of the ICE gatherer.

2.3.2 Methods

`close`

Prunes all local candidates, and closes the port. Associated **`RTCIceTransport`** objects transition to the "disconnected" state (unless they were in the "failed" state). Calling `close()` when *state* is "closed" has no effect.

No parameters.
Return type: `void`

`createAssociatedGatherer`

Create an associated **`RTCIceGatherer`** for RTCP. If *state* is "closed", throw an `InvalidStateError` exception. If called more than once for the same component or if *component* is "RTCP", throw an `InvalidStateError` exception.

No parameters.
Return type: **`RTCIceGatherer`**

`getLocalCandidates`

Retrieve the sequence of valid local candidates associated with the **`RTCIceGatherer`**. This retrieves all unpruned local candidates currently known (except for peer reflexive candidates), even if an `onlocalcandidate` event hasn't been processed yet.

No parameters.

Return type: sequence<**RTCIceCandidate**>

getLocalParameters

Obtain the ICE parameters of the **RTCIceGatherer**.

No parameters.

Return type: **RTCIceParameters**

2.4 The RTCIceParameters Object

The ***RTCIceParameters*** object includes the ICE username fragment and password.

WebIDL

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

2.4.1 Dictionary RTCIceParameters Members

password of type DOMString

ICE password.

usernameFragment of type DOMString

ICE username.

2.5 The RTCIceCandidate Object

The ***RTCIceCandidate*** object includes information relating to an ICE candidate.

EXAMPLE 1

```
{
  foundation: "abcd1234",
  priority: 1694498815,
  ip: "192.0.2.33",
  protocol: "udp",
  port: 10000,
  type: "host"
};
```

WebIDL

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

WebIDL

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

2.5.1 Dictionary RTCIceCandidate Members

foundation of type DOMString

A unique identifier that allows ICE to correlate candidates that appear on multiple RTCIceTransports.

ip of type DOMString

The IP address of the candidate.

port of type unsigned short

The port for the candidate.

`priority` of type unsigned long
: The assigned priority of the candidate. This is automatically populated by the browser.

`protocol` of type *RTCIceProtocol*
: The protocol of the candidate (UDP/TCP).

`relatedAddress` of type DOMString, defaulting to `""`
: For candidates that are derived from others, such as relay or reflexive candidates, the ***relatedAddress*** refers to the candidate that these are derived from. For host candidates, the *relatedAddress* is set to the empty string.

`relatedPort` of type unsigned short
: For candidates that are derived from others, such as relay or reflexive candidates, the ***relatedPort*** refers to the host candidate that these are derived from. For host candidates, the *relatedPort* is null.

`tcpType` of type *RTCIceTcpCandidateType*
: The type of TCP candidate.

`type` of type *RTCIceCandidateType*
: The type of candidate.

2.5.2 The RTCIceProtocol

The ***RTCIceProtocol*** includes the protocol of the ICE candidate.

WebIDL

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

Enumeration description	
`udp`	A UDP candidate, as described in [RFC5245].
`tcp`	A TCP candidate, as described in [RFC6544].

2.5.3 The RTCIceTcpCandidateType

The ***RTCIceTcpCandidateType*** includes the type of the ICE TCP candidate, as described in [RFC6544]. Browsers MUST gather active TCP candidates and only active TCP candidates. Servers and other endpoints MAY gather active, passive or so candidates.

WebIDL

```
enum RTCIceTcpCandidateType {
    "active",
    "passive",
    "so"
};
```

Enumeration description

`active`	An active TCP candidate is one for which the transport will attempt to open an outbound connection but will not receive incoming connection requests.
`passive`	A passive TCP candidate is one for which the transport will receive incoming connection attempts but not attempt a connection.
`so`	An so candidate is one for which the transport will attempt to open a connection simultaneously with its peer.

2.5.4 The RTCIceCandidateType

The ***RTCIceCandidateType*** includes the type of the ICE candidate.

WebIDL

```
enum RTCIceCandidateType {
    "host",
    "srflx",
    "prflx",
    "relay"
};
```

Enumeration description

`host`	A host candidate.
`srflx`	A server reflexive candidate.
`prflx`	A peer reflexive candidate.
`relay`	A relay candidate.

2.6 dictionary RTCIceCandidateComplete

RTCIceCandidateComplete is a dictionary signifying that all `RTCIceCandidate`s are gathered.

WebIDL

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

2.6.1 Dictionary `RTCIceCandidateComplete` Members

`complete` of type boolean, defaulting to **`true`**

This attribute is always present and set to true, indicating that ICE candidate gathering is complete.

2.7 enum RTCIceGathererState

RTCIceGathererState represents the current state of the ICE gatherer.

WebIDL

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

Enumeration description

`new`	The object was just created, and no gathering has occurred yet. Since **`RTCIceGatherer`** objects gather upon construction, this state will only exist momentarily.
`gathering`	The **`RTCIceGatherer`** is in the process of gathering candidates (which includes adding new candidates and removing invalidated candidates).
`complete`	The **`RTCIceGatherer`** has completed gathering. Events such as adding, updating or removing an interface, or adding, changing or removing a TURN server will cause the state to go back to gathering before re-entering "complete" once all candidate changes are finalized.
`closed`	The **`RTCIceGatherer`** has been closed intentionally (by calling `close()`) or as the result of an error.

2.8 RTCIceGathererStateChangedEvent

The `icegathererstatechange` event of the **`RTCIceGatherer`** object uses the **`RTCIceGathererStateChangedEvent`** interface.

***Firing an `RTCIceGathererStateChangedEvent` event named* e** with an **`RTCIceGathererState`** *state* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the `RTCIceGathererStateChangedEvent` interface with the *state* attribute set to the new **`RTCIceGathererState`**, MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCIceGathererStateChangedEventInit : EventInit {
             RTCIceGathererState? state;
};

[Constructor(DOMString type, RTCIceGathererStateChangedEventInit
eventInitDict)]
interface RTCIceGathererStateChangedEvent : Event {
    readonly        attribute RTCIceGathererState state;
};
```

2.8.1 Attributes

`state` of type *RTCIceGathererState*, readonly
The *state* attribute is the new **`RTCIceGathererState`** that caused the event.

2.8.2 Dictionary `RTCIceGathererStateChangedEventInit` Members

`state` of type *RTCIceGathererState*, nullable
The *state* attribute is the new **`RTCIceGathererState`** that caused the event.

2.9 RTCIceGathererEvent

The `icecandidate` event of the **`RTCIceGatherer`** object uses the **`RTCIceGathererEvent`** interface.

***Firing an `RTCIceGathererEvent` event named* e** with an **`RTCIceGatherCandidate`** *candidate* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the **`RTCIceGathererEvent`** interface with the *candidate* attribute set to the new ICE candidate, MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCIceGathererEventInit : EventInit {
             RTCIceGatherCandidate candidate;
};

[Constructor(DOMString type, RTCIceGathererEventInit eventInitDict)]
interface RTCIceGathererEvent : Event {
    readonly        attribute RTCIceGatherCandidate candidate;
};
```

2.9.1 Attributes

`candidate` of type *RTCIceGatherCandidate*, readonly

The *candidate* attribute is the **`RTCIceGatherCandidate`** object with the new ICE candidate that caused the event. If *candidate* is of type `RTCIceCandidateComplete`, there are no additional candidates.

2.9.2 Dictionary `RTCIceGathererEventInit` Members

`candidate` of type *RTCIceGatherCandidate*

The ICE candidate that caused the event.

2.10 dictionary RTCIceGatherOptions

RTCIceGatherOptions provides options relating to the gathering of ICE candidates.

WebIDL

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

2.10.1 Dictionary `RTCIceGatherOptions` Members

`gatherPolicy` of type *RTCIceGatherPolicy*

The ICE gather policy.

iceservers of type sequence<*RTCIceServer*>
The ICE servers to be configured.

2.11 enum RTCIceGatherPolicy

RTCIceGatherPolicy denotes the policy relating to the gathering of ICE candidates.

WebIDL

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

Enumeration description

`all`	The ICE gatherer gathers all types of candidates when this value is specified.
`nohost`	The ICE gatherer gathers all ICE candidate types except for host candidates.
`relay`	The ICE gatherer MUST only gather media relay candidates such as candidates passing through a TURN server. This can be used to reduce leakage of IP addresses in certain use cases.

2.12 Example

EXAMPLE 2

```
// Example to demonstrate use of RTCIceCandidateComplete
// Include some helper functions
import "helper.js";
// Create ICE gather options
var gatherOptions = new RTCIceGatherOptions();
gatherOptions.gatherPolicy = RTCIceGatherPolicy.relay;
gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , { ur
// Create ICE gatherer objects
var iceGatherer = new RTCIceGatherer(gatherOptions);
// Handle state changes
iceGatherer.ongathererstatechange = function (event) {
  myIceGathererStateChange('iceGatherer', event.state);
};
// Prepare to signal local candidates
iceGatherer.onlocalcandidate = function (event) {
  mySendLocalCandidate(event.candidate);
};
// Set up response function
mySignaller.onResponse = function(responseSignaller,response) {
  // We may get N responses
  // ... deal with the N responses as shown in Example 5 of Section 3.
    }
  };
};

mySignaller.send({
   "ice": iceGatherer.getLocalParameters()
});
```

EXAMPLE 3

```
// Helper functions used in all the examples (helper.js)
function trace(text) {
  // This function is used for logging.
  if (text[text.length - 1] === '\n') {
    text = text.substring(0, text.length - 1);
  }
  if (window.performance) {
    var now = (window.performance.now() / 1000).toFixed(3);
    console.log(now + ': ' + text);
  } else {
    console.log(text);
  }
}

function errorHandler (error) {
  trace('Error encountered: ' + error.name);
}

function mySendLocalCandidate(candidate, component, kind, parameters){
  // Set default values
  kind = kind || "all";
  component = component || RTCIceComponent.RTP;
  parameters = parameters || null;
  // Signal the local candidate
  mySignaller.mySendLocalCandidate({
  "candidate": candidate,
  "component": component,
  "kind": kind,
  "parameters": parameters
  });
}

function myIceGathererStateChange(name, state){
  switch(state){
  case RTCIceGathererState.new:
     trace('IceGatherer: ' + name + ' Has been created');
     break;
  case RTCIceGathererState.gathering:
     trace('IceGatherer: ' + name + ' Is gathering candidates');
     break;
  case RTCIceGathererState.complete:
     trace('IceGatherer: ' + name + ' Has finished gathering (for now)
```

```
      break;
  case RTCIceGathererState.closed:
      trace('IceGatherer: ' + name + ' Is closed');
      break;
  default:
      trace('IceGatherer: ' + name + ' Invalid state');
  }
}

function myIceTransportStateChange(name, state){
  switch(state){
  case RTCIceTransportState.new:
      trace('IceTransport: ' + name + ' Has been created');
      break;
  case RTCIceTransportState.checking:
      trace('IceTransport: ' + name + ' Is checking');
      break;
  case RTCIceTransportState.connected:
      trace('IceTransport: ' + name + ' Is connected');
      break;
  case RTCIceTransportState.disconnected:
      trace('IceTransport: ' + name + ' Is disconnected');
      break;
  case RTCIceTransportState.completed:
      trace('IceTransport: ' + name + ' Has finished checking (for now)
      break;
  case RTCIceTransportState.failed:
      trace('IceTransport: ' + name + ' Has failed');
      break;
  case RTCIceTransportState.closed:
      trace('IceTransport: ' + name + ' Is closed');
      break;
  default:
      trace('IceTransport: ' + name + ' Invalid state');
  }
}
```

3. The RTCIceTransport Object

The ***RTCIceTransport*** includes information relating to Interactive Connectivity Establishment (ICE).

3.1 Overview

An **RTCIceTransport** instance is associated to a transport object (such as **RTCDtlsTransport**), and provides RTC related methods to it.

3.2 Operation

An **RTCIceTransport** instance is constructed (optionally) from an **RTCIceGatherer**. If *gatherer.state* is "closed" or *gatherer.component* is "RTCP", then throw an `InvalidStateError` exception.

3.3 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?      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;
};
```

3.3.1 Attributes

`component` of type *RTCIceComponent*, readonly
The component-id of the **`RTCIceTransport`**.

`iceGatherer` of type *RTCIceGatherer*, readonly , nullable
The *iceGatherer* attribute is set to the value of *gatherer* if passed in the constructor or in the latest call to `start()`.

`oncandidatepairchange` of type EventHandler, nullable
This event handler, of event handler type `icecandidatepairchange`, uses the `RTCIceCandidatePairChangedEvent` interface. It MUST be supported by all objects implementing the **`RTCIceTransport`** interface. It is called any time the nominated **`RTCIceCandidatePair`** changes.

`onicestatechange` of type EventHandler, nullable
This event handler, of event handler type `icestatechange`, uses the `RTCIceTransportStateChangedEvent` interface. It MUST be supported by all objects implementing the **`RTCIceTransport`** interface. It is called any time the **`RTCIceTransportState`** changes.

`role` of type *RTCIceRole*, readonly
The current role of the ICE transport.

`state` of type *RTCIceTransportState*, readonly
The current state of the ICE transport.

3.3.2 Methods

`addRemoteCandidate`
Add a remote candidate associated with the remote **`RTCIceTransport`**. If *state* is "closed", throw an `InvalidStateError` exception. When the remote **`RTCIceGatherer`** emits its final candidate, `addRemoteCandidate()` should be called with an `RTCIceCandidateComplete` dictionary as an argument, so that the local **`RTCIceTransport`** can know there are no more remote candidates expected, and can enter the "completed" state.

Parameter	Type	Nullable	Optional	Description
remoteCandidate	**`RTCIceGatherCandidate`**	✘	✘	

Return type: `void`

`createAssociatedTransport`

Create an associated **`RTCIceTransport`** for RTCP. If called more than once for the same component, or if *state* is "closed", throw an `InvalidStateError` exception. If called when *component* is "RTCP", throw an `InvalidStateError` exception.

No parameters.
Return type: **`RTCIceTransport`**

`getNominatedCandidatePair`

Retrieves the nominated candidate pair on which media is flowing. If there is no nominated pair yet, or consent is lost on the nominated pair, NULL is returned.

No parameters.
Return type: **`RTCIceCandidatePair`**, nullable

`getRemoteCandidates`

Retrieve the sequence of candidates associated with the remote **`RTCIceTransport`**. Only returns the candidates previously added using `setRemoteCandidates()` or `addRemoteCandidate()`.

No parameters.
Return type: `sequence<`**`RTCIceCandidate`**`>`

`getRemoteParameters`

Obtain the current ICE parameters of the remote **`RTCIceTransport`**.

No parameters.
Return type: **`RTCIceParameters`**, nullable

`setRemoteCandidates`

Set the sequence of candidates associated with the remote **`RTCIceTransport`**. If *state* is "closed", throw an `InvalidStateError` exception.

Parameter	Type	Nullable	Optional	Description
remoteCandidates	`sequence<`**`RTCIceCandidate`**`>`	✘	✘	

Return type: `void`

`start`

The first time `start()` is called, candidate connectivity checks are started and the ICE transport attempts to connect to the remote **`RTCIceTransport`**. If `start()` is called with invalid parameters, throw an `InvalidParameters` exception. For example, if *gatherer.component* has a value different from *iceTransport.component*, throw an `InvalidParameters` exception. If *state* or *gatherer.state* is "closed", throw an `InvalidStateError` exception. When `start()` is called again, **`RTCIceTransportState`** transitions to the "connected" state, all remote candidates are flushed, and `addRemoteCandidate()` or `setRemoteCandidates()` must be called to add the remote candidates back or replace them.

If a newly constructed **`RTCIceGatherer`** object is passed as an argument when `start()` is called again, an ICE restart as defined in [RFC5245] Section 9.2.1.1 occurs. Since `start()` does not change the username fragment and password of *gatherer*, if `start()` is called again with the same value of *gatherer*, the existing local candidates are reused and the ICE username fragment and password remains unchanged. However, other aspects of the behavior are not currently defined.

As noted in [RFC5245] Section 7.1.2.3, an incoming connectivity check utilizes the local/remote username fragment and the local password, whereas an outgoing connectivity check utilizes the local/remote username fragment and the remote password. Since `start()` provides role information, as well as the remote username fragment and password, once `start()` is called an **`RTCIceTransport`** object can respond to incoming connectivity checks based on its configured role, as well as initiating connectivity checks.

Parameter	Type	Nullable	Optional	Description
gatherer	**`RTCIceGatherer`**	✘	✘	
remoteParameters	**`RTCIceParameters`**	✘	✘	
role	**`RTCIceRole`**	✘	✔	

Return type: `void`

`stop`

Stops and closes the current object. Also removes the object from the **`RTCIceTransportController`**. Calling `stop()` when *state* is "closed" has no effect.

No parameters.
Return type: `void`

3.4 enum RTCIceComponent

RTCIceComponent contains the component-id of the **`RTCIceTransport`**, which will be "RTP" unless RTP and RTCP are not multiplexed and the **`RTCIceTransport`** object was returned by `createAssociatedTransport()`.

WebIDL

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

Enumeration description

`RTP`	The RTP component ID, defined (as '1') in [RFC5245] Section 4.1.1.1. Protocols multiplexed with RTP (e.g. SCTP data channel) share its component ID.
`RTCP`	The RTCP component ID, defined (as '2') in [RFC5245] Section 4.1.1.1.

3.5 enum RTCIceRole

RTCIceRole contains the current role of the ICE transport.

WebIDL

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

Enumeration description

`controlling`	controlling state
`controlled`	controlled state

3.6 enum RTCIceTransportState

RTCIceTransportState represents the current state of the ICE transport.

WebIDL

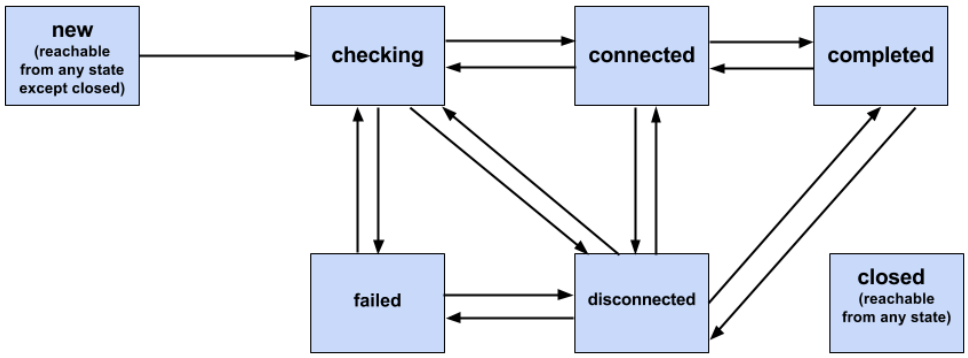
```
enum RTCIceTransportState {
    "new",
    "checking",
    "connected",
    "completed",
    "disconnected",
    "failed",
    "closed"
};
```

Enumeration description

new	The **RTCIceTransport** object is waiting for remote candidates to be supplied. In this state the object can respond to incoming connectivity checks.
checking	The **RTCIceTransport** has received at least one remote candidate, and a local and remote **RTCIceCandidateComplete** dictionary was not added as the last candidate. In this state the **RTCIceTransport** is checking candidate pairs but has not yet found a successful candidate pair, or liveness checks have failed (such as those in [CONSENT]) on a previously successful candidate pair.
connected	The **RTCIceTransport** has received a response to an outgoing connectivity check, or has received incoming DTLS/media after a successful response to an incoming connectivity check, but is still checking other candidate pairs to see if there is a better connection. In this state outgoing media is permitted.
completed	A local and remote **RTCIceCandidateComplete** dictionary was added as the last candidate to the **RTCIceTransport** and all appropriate candidate pairs have been tested and at least one functioning candidate pair has been found.

`disconnected`	The **`RTCIceTransport`** has received at least one local and remote candidate, and a local and remote **`RTCIceCandidateComplete`** dictionary was not added as the last candidate, but all appropriate candidate pairs thus far have been tested and failed (or consent checks [CONSENT], once successful, have now failed). Other candidate pairs may become available for testing as new candidates are trickled, and therefore the "failed" state has not been reached.
`failed`	A local and remote `RTCIceCandidateComplete` dictionary was added as the last candidate to the **`RTCIceTransport`** and all appropriate candidate pairs have been tested and failed.
`closed`	The **`RTCIceTransport`** has shut down and is no longer responding to STUN requests.

The non-normative ICE state transitions are:



3.7 RTCIceTransportStateChangedEvent

The `icestatechange` event of the **`RTCIceTransport`** object uses the **`RTCIceTransportStateChangedEvent`** interface.

***Firing an `RTCIceTransportStateChangedEvent` event named* e** with an **`RTCIceTransportState`** *state* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which

uses the RTCIceTransportStateChangedEvent interface with the *state* attribute set to the new **RTCIceTransportState**, MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCIceTransportStateChangedEventInit : EventInit {
            RTCIceTransportState? state;
};

[Constructor(DOMString type, RTCIceTransportStateChangedEventInit
eventInitDict)]
interface RTCIceTransportStateChangedEvent : Event {
    readonly        attribute RTCIceTransportState state;
};
```

3.7.1 Attributes

state of type *RTCIceTransportState*, readonly
The *state* attribute is the new **RTCIceTransportState** that caused the event.

3.7.2 Dictionary RTCIceTransportStateChangedEventInit Members

state of type *RTCIceTransportState*, nullable
The *state* attribute is the new **RTCIceTransportState** that caused the event.

3.8 RTCIceCandidatePairChangedEvent

The icecandidatepairchange event of the **RTCIceTransport** object uses the **RTCIceCandidatePairChangedEvent** interface.

***Firing an RTCIceCandidatePairChangedEvent event named* e** with an **RTCIceCandidatePair** *pair* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the **RTCIceCandidatePairChangedEvent** interface with the *pair* attribute set to the nominated RTCIceCandidatePair, MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCIceCandidatePairChangedEventInit : EventInit {
            RTCIceCandidatePair pair;
};

[Constructor(DOMString type, RTCIceCandidatePairChangedEventInit
eventInitDict)]
interface RTCIceCandidatePairChangedEvent : Event {
    readonly        attribute RTCIceCandidatePair pair;
};
```

3.8.1 Attributes

pair of type *RTCIceCandidatePair*, readonly
The *pair* attribute is the nominated **RTCIceCandidatePair** that caused the event.

3.8.2 Dictionary RTCIceCandidatePairChangedEventInit Members

pair of type *RTCIceCandidatePair*
The *pair* attribute is the nominated **RTCIceCandidatePair** that caused the event.

3.9 The RTCIceServer Object

The ***RTCIceServer*** is used to provide STUN or TURN server configuration. In network topologies with multiple layers of NATs, it is desirable to have a STUN server between every layer of NATs in addition to the TURN servers to minimize the peer to peer network latency.

An example of an array of RTCIceServer objects:

EXAMPLE 4

```
[ { urls: "stun:stun1.example.net" } , { urls:"turn:turn.example
```

WebIDL

```
dictionary RTCIceServer {
             (DOMString or sequence<DOMString>) urls;
             DOMString                          username;
             DOMString                          credential;
};
```

Dictionary `RTCIceServer` Members

`credential` of type DOMString
: If the uri element is a TURN URI, then this is the credential to use with that TURN server.

`urls` of type (DOMString or sequence<DOMString>)
: STUN or TURN URI(s) as defined in [RFC7064] and [RFC7065]

`username` of type DOMString
: If this RTCIceServer object represents a TURN server, then this attribute specifies the username to use with that TURN server.

3.10 dictionary RTCIceCandidatePair

The ***RTCIceCandidatePair*** contains the currently selected ICE candidate pair.

WebIDL

```
dictionary RTCIceCandidatePair {
             RTCIceCandidate local;
             RTCIceCandidate remote;
};
```

3.10.1 Dictionary `RTCIceCandidatePair` Members

`local` of type *RTCIceCandidate*
: The local ICE candidate.

`remote` of type *RTCIceCandidate*
: The remote ICE candidate.

3.11 Example

EXAMPLE 5

```
// Example to demonstrate forking when RTP and RTCP are not multiplexe
// so that both RTP and RTCP IceGatherer and IceTransport objects are
// Include some helper functions
import "helper.js";
// Create ICE gather options
var gatherOptions = new RTCIceGatherOptions();
gatherOptions.gatherPolicy = RTCIceGatherPolicy.relay;
gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , { ur
// Create ICE gatherer objects
var iceRtpGatherer = new RTCIceGatherer(gatherOptions);
var iceRtcpGatherer = iceRtpGatherer.createAssociatedGatherer();
// Prepare to signal local candidates
iceRtpGatherer.onlocalcandidate = function (event) {
  mySendLocalCandidate(event.candidate, RTCIceComponent.RTP);
};
iceRtcpGatherer.onlocalcandidate = function (event) {
  mySendLocalCandidate(event.candidate, RTCIceComponent.RTCP);
};
// Initialize the ICE transport arrays
var iceRtpTransport = [];
var iceRtcpTransport = [];
var i = 0 ;
// Set up response function
mySignaller.onResponse = function(responseSignaller,response) {
  // We may get N responses
  //
  // Create the ICE RTP and RTCP transports
  i = iceRtpTransport.push(new RTCIceTransport(iceRtpGatherer)) - 1;
  iceRtcpTransport.push(iceRtpTransport.createAssociatedTransport());
  // Start the RTP and RTCP ICE transports so that outgoing ICE connec
  iceRtpTransport[i].start(iceRtpGatherer, response.icertp, RTCIceRole
  iceRtcpTransport[i].start(iceRtcpGatherer, response.icertcp, RTCIceR
  // Prepare to add ICE candidates signalled by the remote peer
  responseSignaller.onRemoteCandidate = function(remote) {
    // Locate the ICE transport that the signaled candidate relates to
    var j = 0;
    for (j=0; j < iceTransport.length; j++){
      if (getRemoteParameters(iceTransport(j)).userNameFragment === re
        if (remote.component === RTCIceComponent.RTP){
          iceRtpTransport[j].addRemoteCandidate(remote.candidate);
        } else {
          iceRtcpTransport[j].addRemoteCandidate(remote.candidate);
```

```
        }
      }
    }
  };
};
mySignaller.send({
   "icertp": iceRtpGatherer.getLocalParameters(),
   "icertcp": iceRtcpGatherer.getLocalParameters()
});
```

4. The RTCDtlsTransport Object

The ***RTCDtlsTransport*** object includes information relating to Datagram Transport Layer Security (DTLS) transport.

4.1 Overview

An **RTCDtlsTransport** instance is associated to an **RTCRtpSender**, an **RTCRtpReceiver**, or an **RTCSctpTransport** instance.

4.2 Operation

A **RTCDtlsTransport** instance is constructed with an **RTCIceTransport** object. An RTP **RTCDtlsTransport** MUST share its certificate and fingerprint(s) with the associated RTCP **RTCDtlsTransport**. Therefore when an **RTCDtlsTransport** is constructed from an **RTCIceTransport**, the implementation MUST check whether an associated **RTCDtlsTransport** has already been constructed and if so, MUST reuse the certificate and fingerprint(s) of the associated **RTCDtlsTransport**.

A newly constructed **RTCDtlsTransport** MUST listen and respond to incoming DTLS packets before `start()` is called. However, to complete the negotiation it is necessary to verify the remote fingerprint, which is dependent on *remoteParameters*, passed to `start()`. After the DTLS handshake exchange completes (but before the remote fingerprint is verified) incoming media packets may be received. A modest buffer MUST be provided to avoid loss of media prior to remote fingerprint validation (which can begin after `start()` is called).

If an attempt is made to construct a **RTCDtlsTransport** instance from an **RTCIceTransport** in the "closed" state, an InvalidStateError exception is thrown. Since the Datagram Transport Layer Security (DTLS) negotiation occurs between transport endpoints determined via ICE, implementations of this specification MUST support multiplexing of STUN, TURN, DTLS and RTP and/or RTCP. This multiplexing, originally described in [RFC5764] Section 5.1.2, is being revised in [MUX-FIXES].

4.3 Interface Definition

WebIDL

```
typedef (octet[]) ArrayBuffer;
```

WebIDL

```
[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;
};
```

4.3.1 Attributes

ondtlsstatechange of type EventHandler, nullable

This event handler, of event handler type dtlsstatechange, uses the **RTCDtlsTransportStateChangedEvent** interface. It MUST be supported by all objects implementing the **RTCDtlsTransport** interface. It is called any time the RTCDtlsTransportState changes.

onerror of type EventHandler, nullable

This event handler, of event handler type error, MUST be supported by all objects implementing the **RTCDtlsTransport** interface. This event MUST be fired on reception

of a DTLS alert; an implementation SHOULD include DTLS alert information in *error.message*.

`state` of type *RTCDtlsTransportState*, readonly
The current state of the DTLS transport.

`transport` of type *RTCIceTransport*, readonly
The associated **`RTCIceTransport`** instance.

4.3.2 Methods

`getLocalParameters`
Obtain the DTLS parameters of the local **`RTCDtlsTransport`**.

No parameters.
Return type: **`RTCDtlsParameters`**

`getRemoteCertificates`
Obtain the certificates used by the remote peer.

No parameters.
Return type: `sequence<`**`ArrayBuffer`**`>`

`getRemoteParameters`
Obtain the current DTLS parameters of the remote **`RTCDtlsTransport`**.

No parameters.
Return type: **`RTCDtlsParameters`**, nullable

`start`
Start DTLS transport negotiation with the parameters of the remote DTLS transport, including verification of the remote fingerprint, then once the DTLS transport session is established, negotiate a ***DTLS-SRTP*** [RFC5764] session to establish keys so as protect media using SRTP [RFC3711]. Since symmetric RTP [RFC4961] is utilized, the DTLS-SRTP session is bi-directional.

If *remoteParameters* is invalid, throw an `InvalidParameters` exception. If `start()` is called after a previous `start()` call, or if *state* is "closed", throw an `InvalidStateError` exception. Only a single DTLS transport can be multiplexed over an ICE transport. Therefore if a **`RTCDtlsTransport`** object *dtlsTransportB* is constructed with an **`RTCIceTransport`** object *iceTransport* previously used to

construct another **RTCDtlsTransport** object *dtlsTransportA*, then if `dtlsTransportB.start()` is called prior to having called `dtlsTransportA.stop()`, then throw an `InvalidStateError` exception.

Parameter	Type	Nullable	Optional	Description
remoteParameters	**RTCDtlsParameters**	✘	✘	

Return type: `void`

stop

Stops and closes the DTLS transport object. Calling `stop()` when *state* is "closed" has no effect.

No parameters.
Return type: `void`

4.4 The RTCDtlsParameters Object

The ***RTCDtlsParameters*** object includes information relating to DTLS configuration.

WebIDL

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

4.4.1 Dictionary RTCDtlsParameters Members

fingerprints of type sequence<*RTCDtlsFingerprint*>
Sequence of fingerprints.

role of type *RTCDtlsRole*, defaulting to **"auto"**
The DTLS role, with a default of auto.

4.5 The RTCDtlsFingerprint Object

The ***RTCDtlsFingerprint*** object includes the hash function algorithm and certificate fingerprint as described in [RFC4572].

WebIDL

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

4.5.1 Dictionary `RTCDtlsFingerprint` Members

`algorithm` of type DOMString
: One of the the hash function algorithms defined in the 'Hash function Textual Names' registry, initially specified in [RFC4572] Section 8.

`value` of type DOMString
: The value of the certificate fingerprint in lowercase hex string as expressed utilizing the syntax of 'fingerprint' in [RFC4572] Section 5.

4.6 enum RTCDtlsRole

RTCDtlsRole indicates the role of the DTLS transport.

WebIDL

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

Enumeration description

`auto`	The DTLS role is determined based on the resolved ICE role: the "controlled" role acts as the DTLS client, the "controlling" role acts as the DTLS server. Since **`RTCDtlsRole`** is initialized to "auto" on construction of an **`RTCDtlsTransport`** object, `transport.getLocalParameters().RTCDtlsRole` will have an initial value of "auto".

`client`	The DTLS client role. A transition to "client" will occur if `start(remoteParameters)` is called with *remoteParameters.RTCDtlsRole* having a value of "server". If **`RTCDtlsRole`** had previously had a value of "server" (e.g. due to the **`RTCDtlsTransport`** having previously received packets from a DTLS client), then the DTLS session is reset prior to transitioning to the "client" role.
`server`	The DTLS server role. If **`RTCDtlsRole`** has a value of "auto" and the **`RTCDtlsTransport`** receives a DTLS client_helo packet, **`RTCDtlsRole`** will transition to "server", even before `start()` is called. A transition from "auto" to "server" will also occur if `start(remoteParameters)` is called with *remoteParameters.RTCDtlsRole* having a value of "client".

4.7 enum RTCDtlsTransportState

RTCDtlsTransportState indicates the state of the DTLS transport.

WebIDL

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

Enumeration description

`new`	The **`RTCDtlsTransport`** object has been created and has not started negotiating yet.
`connecting`	DTLS is in the process of negotiating a secure connection. Once a secure connection is negotiated and DTLS-SRTP has derived keys (but prior to verification of the remote fingerprint, enabled by calling `start()`), incoming media can flow through.

`connected`	DTLS has completed negotiation of a secure connection (including DTLS-SRTP and remote fingerprint verification). Outgoing media can now flow through.
`closed`	The DTLS connection has been closed intentionally via a call to `stop()` or as the result of an error (such as a failure to validate the remote fingerprint). Calling `transport.stop()` will also result in a transition to the "closed" state.

4.8 RTCDtlsTransportStateChangedEvent

The ***dtlsstatechange*** event of the **`RTCDtlsTransport`** object uses the **`RTCDtlsTransportStateChangedEvent`** interface.

Firing an `RTCDtlsTransportStateChangedEvent` event named **e** with an **`RTCDtlsTransportState`** *state* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the **`RTCDtlsTransportStateChangedEvent`** interface with the *state* attribute set to the new **`RTCDtlsTransportState`**, MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCDtlsTransportStateChangedEventInit : EventInit {
             RTCDtlsTransportState? state;
};

[Constructor(DOMString type, RTCDtlsTransportStateChangedEventInit
eventInitDict)]
interface RTCDtlsTransportStateChangedEvent : Event {
    readonly        attribute RTCDtlsTransportState state;
};
```

4.8.1 Attributes

`state` of type *RTCDtlsTransportState*, readonly

The *state* attribute is the new **`RTCDtlsTransportState`** that caused the event.

4.8.2 Dictionary `RTCDtlsTransportStateChangedEventInit` Members

`state` of type *RTCDtlsTransportState*, nullable

The *state* attribute is the new **`RTCDtlsTransportState`** that caused the event.

4.9 Examples

EXAMPLE 6

```
// This is an example of  how to offer ICE and DTLS parameters and
// ICE candidates and get back ICE and DTLS parameters and ICE candida
// and start both ICE and DTLS, assuming that RTP and RTCP are multipl
// Assume that we have a way to signal (mySignaller).
// Include some helper functions
import "helper.js";
function initiate(mySignaller) {
  // Prepare the ICE gatherer
  var gatherOptions = new RTCIceGatherOptions();
  gatherOptions.gatherPolicy = RTCIceGatherPolicy.all;
  gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , {
  var iceGatherer = new RTCIceGatherer(gatherOptions);
  iceGatherer.onlocalcandidate = function(event) {
    mySignaller.mySendLocalCandidate(event.candidate);
  };
  // Initialize the ICE and DTLS transport arrays
  var iceTransport = [];
  var dtlsTransport = [];
  var i = 0;
  // Create a DTLS transport so we can signal the parameters
  var ice = new RTCIceTransport(iceGatherer);
  var dtls = new RTCDtlsTransport(ice);
  // Prepare to handle remote ICE candidates
  mySignaller.onRemoteCandidate = function(remote){
  // Figure out which IceTranport a remote candidate relates to by mat
    var j = 0;
    for (j=0; j < iceTransport.length; j++){
      if (getRemoteParameters(iceTransport(j)).userNameFragment === re
        iceTransport[j].addRemoteCandidate(remote.candidate);
      }
    }
  };
  // ... create RtpSender/RtpReceiver objects as illustrated in Sectio

  mySignaller.mySendInitiate({
    "ice": iceGatherer.getLocalParameters(),
    "dtls": dtls.getLocalParameters(),
    // ... marshall RtpSender/RtpReceiver capabilities as illustrated
  }, function(remote) {
    // Create the ICE and DTLS transports
    i = iceTransport.push(new RTCIceTransport(iceGatherer));
    // Issue 211: See: https://github.com/openpeer/ortc/issues/211
```

```
    // The statement below constructs a RTCDtlsTransport object with a
    dtlsTransport.push(new RTCDtlsTransport(iceTransport[i]));
    // Start the ICE transport
    iceTransport[i].start(iceGatherer, remote.ice, RTCIceRole.controll
    // Since the RTCDtlsTransport fingerprint will not match what was
    dtlsTransport[i].start(remote.dtls);
    // ... configure RtpSender/RtpReceiver objects as illustrated in S
  });
}
```

EXAMPLE 7

```
// This is an example of how to answer with ICE and DTLS
// and DTLS parameters and ICE candidates and start both ICE and DTLS,
// assuming that RTP and RTCP are multiplexed.
// Include some helper functions
import "helper.js";
// Assume that remote info is signalled to us.
function accept(mySignaller, remote) {
  var gatherOptions = new RTCIceGatherOptions();
  gatherOptions.gatherPolicy = RTCIceGatherPolicy.all;
  gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , {
  var iceGatherer = new RTCIceGatherer(gatherOptions);
  iceGatherer.onlocalcandidate = function(event) {
    mySignaller.mySendLocalCandidate(event.candidate);
  };
  // Create ICE and DTLS transports
  var ice = new RTCIceTransport(iceGatherer);
  var dtls = new RTCDtlsTransport(ice);
  // Prepare to handle remote candidates
  mySignaller.onRemoteCandidate = function(remote) {
    ice.addRemoteCandidate(remote.candidate);
  };
  // ... create RtpSender/RtpReceiver objects as illustrated in Sectio

  mySignaller.mySendAccept({
    "ice": iceGatherer.getLocalParameters(),
    "dtls": dtls.getLocalParameters()
    // ... marshall RtpSender/RtpReceiver capabilities as illustrated
  });
  // Start the ICE transport with an implicit gather policy of "all"
  ice.start(iceGatherer, remote.ice, RTCIceRole.controlled);
  // Start the DTLS transport and send data into the bit bucket if the
  // since the remote DTLS transport will not have the same certificat
  dtls.start(remote.dtls);
  // ... configure RtpSender/RtpReceiver objects as illustrated in Sec
}
```

5. The RTCRtpSender Object

The ***RTCRtpSender*** includes information relating to the RTP sender.

5.1 Overview

An **RTCRtpSender** instance is associated to a sending MediaStreamTrack and provides RTC related methods to it.

5.2 Operation

A **RTCRtpSender** instance is constructed from an MediaStreamTrack object and associated to an **RTCDtlsTransport**. If an attempt is made to construct an **RTCRtpSender** object with *transport.state* or *rtcpTransport.state* "closed", throw an `InvalidStateError` exception.

5.3 Interface Definition

WebIDL

```
[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                   setTrack (MediaStreamTrack track);
    static RTCRtpCapabilities getCapabilities (optional DOMString
kind);
    void                      send (RTCRtpParameters parameters);
    void                      stop ();
                    attribute EventHandler?    onerror;
                    attribute EventHandler?    onssrcconflict;
};
```

5.3.1 Attributes

onerror of type EventHandler, nullable

This event handler, of event handler type `error`, MUST be supported by all objects implementing the **RTCRtpSender** interface. This event MUST be fired if an issue is

found with the **RTCRtpParameters** object passed to `send()`, that is not immediately detected.

onssrcconflict of type EventHandler, nullable
: This event handler, of event handler type `RTCSsrcConflictEvent`, MUST be supported by all objects implementing the **RTCRtpSender** interface. This event MUST be fired if an SSRC conflict is detected. On an SSRC conflict, the `RTCRtpSender` automatically sends an RTCP BYE on the conflicted SSRC.

rtcpTransport of type *RTCDtlsTransport*, readonly
: The associated RTCP **RTCDtlsTransport** instance.

track of type MediaStreamTrack, readonly
: The associated `MediaStreamTrack` instance.

transport of type *RTCDtlsTransport*, readonly
: The associated RTP **RTCDtlsTransport** instance.

5.3.2 Methods

getCapabilities, static
: Obtain the sender capabilities, based on *kind*. If *kind* is omitted or is set to "", then all capabilities are returned. Capabilities such as retransmission [RFC4588], redundancy [RFC2198], and Forward Error Correction that do not have an associated value of *kind* are always included, regardless of the value of *kind* passed to `getCapabilities()`.

Parameter	Type	Nullable	Optional	Description
kind	`DOMString`	✘	✔	

Return type: **RTCRtpCapabilities**

send
: Media to be sent is controlled by parameters. If `send()` is called with invalid *parameters*, throw an `InvalidParameters` exception. If *rtcpTransport* is not set and `send(parameters)` is called with *parameters.rtcp.mux* set to `false`, throw an `InvalidParameters` exception. The `send()` method does not update *parameters* based on what is currently being sent, so that the value of *parameters* remains that last passed to the `send()` method. The `RTCRtpSender` object starts sending when `send()` is called for the first time, and changes the sending `parameters` when `send()` is called again. The `RTCRtpSender` object stops sending when `stop()` is called.

Parameter	Type	Nullable	Optional	Description
parameters	**RTCRtpParameters**	✘	✘	

Return type: `void`

`setTrack`

Attempts to replace the track being sent with another track provided.

When the `setTrack()` method is invoked, the user agent MUST run the following steps:

1. Let *p* be a new promise.
2. Let *withTrack* be the argument to this method.
3. Run the following steps asynchronously:
    1. If `withTrack.kind` differs from `RTCRtpSender.track.kind` or if *withTrack* has different `peerIdentity` constraints, then reject *p* with `IncompatibleMediaStreamTrackError` and abort these steps.
    2. Set the `RTCRtpSender.track` attribute to *withTrack*, and have the sender seamlessly switch to transmitting *withTrack* in place of what it is sending. On the remote receiving end, the track maintains its existing grouping and id until the connection ends.

Parameter	Type	Nullable	Optional	Description
track	`MediaStreamTrack`	✘	✘	

Return type: `Promise`

`setTransport`

Set the RTP **`RTCDtlsTransport`** (and if used) RTCP **`RTCDtlsTransport`**. If `setTransport()` is called with a single argument or if *rtcpTransport* is not set, and the last call to `sender.send(parameters)` had `parameters.rtcp.mux` set to `false`, throw an `InvalidParameters` exception. If `setTransport()` is called when *transport.state* or *rtcpTransport.state* is "closed", throw an `InvalidStateError` exception.

Parameter	Type	Nullable	Optional	Description
transport	**RTCDtlsTransport**	✘	✘	

rtcpTransport	**RTCDtlsTransport**	✘	✔

Return type: `void`

stop

Stops sending the track on the wire, and sends an RTCP BYE. Stop is final as in `MediaStreamTrack.stop()`

No parameters.
Return type: `void`

5.4 RTCSsrcConflictEvent

The `ssrcconflict` event of the **RTCRtpSender** object uses the **RTCSsrcConflictEvent** interface.

Firing an ***RTCSsrcConflictEvent*** event named *e* with an *ssrc* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the **RTCSsrcConflictEvent** interface with the `ssrc` attribute set to the conflicting SSRC MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCSsrcConflictEventInit : EventInit {
             unsigned long? ssrc;
};

[Constructor(DOMString type, RTCSsrcConflictEventInit eventInitDict)]
interface RTCSsrcConflictEvent : Event {
    readonly        attribute unsigned long ssrc;
};
```

5.4.1 Attributes

ssrc of type unsigned long, readonly
The *ssrc* attribute represents the conflicting SSRC that caused the event.

5.4.2 Dictionary RTCSsrcConflictEventInit Members

ssrc of type unsigned long, nullable

The *ssrc* attribute represents the conflicting SSRC that caused the event.

5.5 Example

> EXAMPLE 8

6. The RTCRtpReceiver Object

The ***RTCRtpReceiver*** includes information relating to the RTP receiver.

6.1 Overview

An **`RTCRtpReceiver`** instance is associated to a receiving MediaStreamTrack and provides RTC related methods to it.

6.2 Operation

A **`RTCRtpReceiver`** instance is constructed from an **`RTCDtlsTransport`** object. If an attempt is made to construct an **`RTCRtpReceiver`** object with *transport.state* or *rtcpTransport.state* "closed", throw an `InvalidStateError` exception.

6.3 Interface Definition

WebIDL

```
[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;
};
```

6.3.1 Attributes

onerror of type EventHandler, nullable

This event handler, of event handler type `error`, MUST be supported by all objects implementing the **RTCRtpReceiver** interface. This event MUST be fired if an issue is found with the **RTCRtpParameters** object passed to `receive()`, that is not immediately detected.

rtcpTransport of type *RTCDtlsTransport*, readonly

The associated RTCP **RTCDtlsTransport** instance.

track of type MediaStreamTrack, readonly , nullable

The associated `MediaStreamTrack` instance.

transport of type *RTCDtlsTransport*, readonly

The associated RTP **RTCDtlsTransport** instance.

6.3.2 Methods

getCapabilities, static

Obtain the receiver capabilities, based on *kind*. If *kind* is omitted or set to "", then all capabilities are returned. Capabilities such as retransmission [RFC4588], redundancy [RFC2198], and Forward Error Correction that do not have an associated value of *kind* are always included, regardless of the value of *kind* passed to `getCapabilities()`. To avoid confusion, `getCapabilities(kind)` should return codecs with a matching

intrinsic *kind* value, as well as codecs with no intrinsic *kind* (such as redundancy [RFC2198]). For codecs with no intrinsic *kind*, *RTCRtpCapabilities.RTCRtpCodecCapability[i].kind* returned by `getCapabilities(kind)` should be set to the value of *kind* if *kind* is equal to "audio" or "video". If the *kind* argument was omitted or set to "", then the value of *RTCRtpCapabilities.RTCRtpCodecCapability[i].kind* is set to "".

Parameter	Type	Nullable	Optional	Description
kind	`DOMString`	✘	✔	

Return type: **`RTCRtpCapabilities`**

`receive`

Media to be received is controlled by *parameters*. If `receive(parameters)` is called with invalid *parameters*, throw an `InvalidParameters` exception. If *rtcpTransport* is not set and `receive(parameters)` is called with *parameters.rtcp.mux* set to `false`, throw an `InvalidParameters` exception. The `receive()` method does not update *parameters* based on what is currently being received, so that the value of *parameters* remains that last passed to the `receive()` method. The `RTCRtpReceiver` object starts receiving when `receive()` is called for the first time, and changes the receiving `parameters` when `receive()` is called again. The `RTCRtpReceiver` object stops receiving when `stop()` is called. After `receive()` returns, *track* is set. The value of *track.kind* is determined based on the *kind* of the codecs provided in *parameters.codecs*. If *parameters.codecs* are all of a single *kind* then *track.kind* is set to that *kind*. If *parameters.codecs* are of mixed *kind*, throw an `InvalidParameters` exception. For this purpose a *kind* of "" is not considered mixed.

Parameter	Type	Nullable	Optional	Description
parameters	**`RTCRtpParameters`**	✘	✘	

Return type: `void`

`setTransport`

Set the RTP **`RTCDtlsTransport`** (and if used) RTCP **`RTCDtlsTransport`**. If `setTransport()` is called with a single argument or *rtcpTransport* is not set, and the last call to `receive(parameters)` had `parameters.rtcp.mux` set to `false`, throw an `InvalidParameters` exception. If `setTransport()` is called and *transport.state* or *rtcpTransport.state* is "closed", throw an `InvalidParameters` exception.

Parameter	Type	Nullable	Optional	Description
transport	**RTCDtlsTransport**	✘	✘	
rtcpTransport	**RTCDtlsTransport**	✘	✔	

Return type: `void`

stop

Stops receiving the track on the wire. Stop is final like MediaStreamTrack.

No parameters.

Return type: `void`

6.4 Examples

EXAMPLE 9

```
// Assume we already have a way to signal, a transport
// (RTCDtlsTransport), and audio and video tracks. This is an example
// of  how to offer them  and get back an answer with audio and
// video tracks, and begin sending and receiving them.
// The example assumes that RTP and RTCP are multiplexed.
function myInitiate(mySignaller, transport, audioTrack, videoTrack) {
  var audioSender = new RTCRtpSender(audioTrack, transport);
  var videoSender = new RTCRtpSender(videoTrack, transport);
  var audioReceiver = new RTCRtpReceiver(transport);
  var videoReceiver = new RTCRtpReceiver(transport);

// Retrieve the audio and video receiver capabilities
  var recvAudioCaps = RTCRtpReceiver.getCapabilities('audio');
  var recvVideoCaps = RTCRtpReceiver.getCapabilities('video');
// Retrieve the audio and video sender capabilities
  var sendAudioCaps = RTCRtpSender.getCapabilities('audio');
  var sendVideoCaps = RTCRtpSender.getCapabilities('video');

  mySignaller.myOfferTracks({
    // The initiator offers its receiver and sender capabilities.
    "recvAudioCaps": recvAudioCaps,
    "recvVideoCaps": recvVideoCaps,
    "sendAudioCaps": sendAudioCaps,
    "sendVideoCaps": sendVideoCaps
  }, function(answer) {
    // The responder answers with its receiver capabilities

    // Derive the send and receive parameters
    var audioSendParams = myCapsToSendParams(sendAudioCaps, answer.rec
    var videoSendParams = myCapsToSendParams(sendVideoCaps, answer.rec
    var audioRecvParams = myCapsToRecvParams(recvAudioCaps, answer.sen
    var videoRecvParams = myCapsToRecvParams(recvVideoCaps, answer.sen
    audioSender.send(audioSendParams);
    videoSender.send(videoSendParams);
    audioReceiver.receive(audioRecvParams);
    videoReceiver.receive(videoRecvParams);

    // Now we can render/play
    // audioReceiver.track and videoReceiver.track.
  });
}
```

EXAMPLE 10

```
// Assume we already have a way to signal, a transport (RTCDtlsTranspo
// and audio and video tracks. This is an example of how to answer an
// offer with audio and video tracks, and begin sending and receiving
// The example assumes that RTP and RTCP are multiplexed.
function myAccept(mySignaller, remote, transport, audioTrack, videoTra
  var audioSender = new RTCRtpSender(audioTrack, transport);
  var videoSender = new RTCRtpSender(videoTrack, transport);
  var audioReceiver = new RTCRtpReceiver(transport);
  var videoReceiver = new RTCRtpReceiver(transport);

// Retrieve the send and receive capabilities
  var recvAudioCaps = RTCRtpReceiver.getCapabilities('audio');
  var recvVideoCaps = RTCRtpReceiver.getCapabilities('video');
  var sendAudioCaps = RTCRtpSender.getCapabilities('audio');
  var sendVideoCaps = RTCRtpSender.getCapabilities('video');

  mySignaller.myAnswerTracks({
    "recvAudioCaps": recvAudioCaps,
    "recvVideoCaps": recvVideoCaps,
    "sendAudioCaps": sendAudioCaps,
    "sendVideoCaps": sendVideoCaps
  });

    // Derive the send and receive parameters using Javascript functio
    var audioSendParams = myCapsToSendParams(sendAudioCaps, remote.rec
    var videoSendParams = myCapsToSendParams(sendVideoCaps, remote.rec
    var audioRecvParams = myCapsToRecvParams(recvAudioCaps, remote.sen
    var videoRecvParams = myCapsToRecvParams(recvVideoCaps, remote.sen
    audioSender.send(audioSendParams);
    videoSender.send(videoSendParams);
    audioReceiver.receive(audioRecvParams);
    videoReceiver.receive(videoRecvParams);

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

7. The RTCIceTransportController Object

The ***RTCIceTransportController*** object assists in the managing of ICE freezing and bandwidth estimation.

7.1 Overview

An **RTCIceTransportController** object provides methods to add and retrieve **RTCIceTransport** objects with a *component* of "RTP" (associated **RTCIceTransport** objects with a *component* of "RTCP" are included implicitly).

7.2 Operation

An **RTCIceTransportController** instance is automatically constructed.

7.3 Interface Definition

WebIDL

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

7.3.1 Methods

addTransport

Adds *transport* to the **RTCIceTransportController** object for the purposes of managing ICE freezing and sharing bandwidth estimation. Since `addTransport` manages ICE freezing, candidate pairs that are not in the frozen state maintain their state when `addTransport(transport)` is called. **RTCIceTransport** objects will be unfrozen according to their *index*. *transport* is inserted at *index*, or at the end if *index* is not specified. If *index* is greater than the current number of **RTCIceTransport**s with a *component* of "RTP", throw an `InvalidParameters` exception. If

`transport.state` is "closed", throw an `InvalidStateError` exception. If *transport* has already been added to another **`RTCIceTransportController`** object, or if `transport.component` is "RTCP", throw an `InvalidStateError` exception.

Parameter	Type	Nullable	Optional	Description
transport	**`RTCIceTransport`**	✘	✘	
index	`unsigned long`	✘	✔	

Return type: `void`

`getTransports`

Returns the **`RTCIceTransport`** objects with a *component* of "RTP".

No parameters.

Return type: `sequence<`**`RTCIceTransport`**`>`

7.4 Examples

EXAMPLE 11

```
// This is an example of how to utilize distinct ICE transports for Au
// as well as for RTP and RTCP.  If both sides can multiplex audio/vid
// and RTP/RTCP then the multiplexing will occur.
//
// Assume we have an audioTrack and a videoTrack to send.
//
// Include some helper functions
import "helper.js";
// Create the ICE gather options
var gatherOptions = new RTCIceGatherOptions();
gatherOptions.gatherPolicy = RTCIceGatherPolicy.all;
gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , { ur

// Create the RTP and RTCP ICE gatherers for audio and video
var audioRtpIceGatherer = new RTCIceGatherer(gatherOptions);
var audioRtcpIceGatherer = audioRtpIceGatherer.createAssociatedGathere
var videoRtpIceGatherer = new RTCIceGatherer(gatherOptions);
var videoRtcpIceGatherer = videoRtpIceGatherer.createAssociatedGathere

// Set up the ICE gatherer error handlers
audioRtpIceGatherer.onerror = errorHandler;
audioRtcpIceGatherer.onerror = errorHandler;
videoRtpIceGatherer.onerror = errorHandler;
videoRtcpIceGatherer.onerror = errorHandler;

// Create the RTP and RTCP ICE transports for audio and video
var audioRtpIceTransport = new RTCIceTransport(audioRtpIceGatherer);
var audioRtcpIceTransport = audioRtpIceTransport.createAssociatedTrans
var videoRtpIceTransport = new RTCIceTransport(videoRtpIceGatherer);
var videoRtcpIceTransport = audioRtpIceTransport.createAssociatedTrans

// Enable local ICE candidates to be signaled to the remote peer.
audioRtpIceGatherer.onlocalcandidate = function (event)  {
  mySendLocalCandidate(event.candidate, RTCIceComponent.RTP, "audio");
};
audioRtcpIceGatherer.onlocalcandidate = function (event) {
  mySendLocalCandidate(event.candidate, RTCIceComponent.RTCP, "audio")
};
videoRtpIceGatherer.onlocalcandidate = function (event)  {
  mySendLocalCandidate(event.candidate, RTCIceComponent.RTP, "video");
};
videoRtcpIceGatherer.onlocalcandidate = function (event) {
```

```
  mySendLocalCandidate(event.candidate, RTCIceComponent.RTCP, "video")
};

// Set up the ICE state change event handlers
audioRtpIceTransport.onicestatechange =  function(event){
  myIceTransportStateChange('audioRtpIceTransport', event.state);
};
audioRtcpIceTransport.onicestatechange = function(event){
  myIceTransportStateChange('audioRtcpIceTransport', event.state);
};
videoRtpIceTransport.onicestatechange =  function(event){
  myIceTransportStateChange('videoRtpIceTransport', event.state);
};
videoRtcpIceTransport.onicestatechange = function(event){
  myIceTransportStateChange('videoRtcpIceTransport', event.state);
};

// Prepare to add ICE candidates signaled by the remote peer on any of
mySignaller.onRemoteCandidate = function(remote) {
  switch (remote.kind) {
     case "audio":
       if (remote.component === RTCIceComponent.RTP){
          audioRtpIceTransport.addRemoteCandidate(remote.candidate);
       } else {
          audioRtcpIceTransport.addRemoteCandidate(remote.candidate);
       }
       break;
     case "video":
       if (remote.component === RTCIceComponent.RTP){
          videoRtpIceTransport.addRemoteCandidate(remote.candidate);
       } else {
          videoRtcpIceTransport.addRemoteCandidate(remote.candidate);
       }
       break;
     default:
       trace('Invalid media type received: ' + remote.kind);
  }
};

// Create the DTLS transports
var audioRtpDtlsTransport = new RTCDtlsTransport(audioRtpIceTransport)
var audioRtcpDtlsTransport = new RTCDtlsTransport(audioRtcpIceTranspor
var videoRtpDtlsTransport = new RTCDtlsTransport(videoRtpIceTransport)
```

```
var videoRtcpDtlsTransport = new RTCDtlsTransport(videoRtcpIceTranspor

// Create the sender and receiver objects
var audioSender = new RtpSender(audioTrack, audioRtpDtlsTransport, aud
var videoSender = new RtpSender(videoTrack, videoRtpDtlsTransport, vid
var audioReceiver = new RtpReceiver(audioRtpDtlsTransport, audioRtcpDt
var videoReceiver = new RtpReceiver(videoRtpDtlsTransport, videoRtcpDt

// Retrieve the receiver and sender capabilities
var recvAudioCaps = RTCRtpReceiver.getCapabilities('audio');
var recvVideoCaps = RTCRtpReceiver.getCapabilities('video');
var sendAudioCaps = RTCRtpSender.getCapabilities('audio');
var sendVideoCaps = RTCRtpSender.getCapabilities('video');

// Exchange ICE/DTLS parameters and Send/Receive capabilities

mySignaller.myOfferTracks({
  // Indicate that the initiator would prefer to multiplex both A/V an
  "bundle": true,
  // Indicate that the initiator is willing to multiplex RTP/RTCP with
  "rtcpMux": true,
  // Offer the ICE parameters
  "audioRtpIce": audioRtpIceGatherer.getLocalParameters(),
  "audioRtcpIce": audioRtcpIceGatherer.getLocalParameters(),
  "videoRtpIce": videoRtpIceGatherer.getLocalParameters(),
  "videoRtcpIce": videoRtcpIceGatherer.getLocalParameters(),
  // Offer the DTLS parameters
  "audioRtpDtls": audioRtpDtlsTransport.getLocalParameters(),
  // Assume implementation does not yet support dtlsTransport.getLocal
  "audioRtcpDtls": audioRtcpDtlsTransport.getLocalParameters(),
  "videoRtpDtls": videoRtpDtlsTransport.getLocalParameters(),
  "videoRtcpDtls": videoRtcpDtlsTransport.getLocalParameters(),
  // Offer the receiver and sender audio and video capabilities.
  "recvAudioCaps": recvAudioCaps,
  "recvVideoCaps": recvVideoCaps,
  "sendAudioCaps": sendAudioCaps,
  "sendVideoCaps": sendVideoCaps
}, function(answer) {
   // The responder answers with its preferences, parameters and capab
   // Since we didn't create transport arrays, we are assuming that th
   //
   // Derive the send and receive parameters, assuming that RTP/RTCP m
   var audioSendParams = myCapsToSendParams(sendAudioCaps, answer.recv
```

```
var videoSendParams = myCapsToSendParams(sendVideoCaps, answer.recv
var audioRecvParams = myCapsToRecvParams(recvAudioCaps, answer.send
var videoRecvParams = myCapsToRecvParams(recvVideoCaps, answer.send
//
// If the responder wishes to enable bundle, we will enable it
if (answer.bundle) {
  // Since bundle implies RTP/RTCP multiplexing, we only need a sin
  // ICE transport and DTLS transport.  No need for the ICE transpo
  audioRtpIceTransport.start(audioRtpIceGatherer,answer.audioRtpIce
  audioRtpDtlsTransport.start(remote.audioRtpDtls);
  //
  // Replace the transport on the Sender and Receiver objects
  //
  audioSender.setTransport(audioRtpDtlsTransport);
  videoSender.setTransport(audioRtpDtlsTransport);
  audioReceiver.setTransport(audioRtpDtlsTransport);
  videoReceiver.setTransport(audioRtpDtlsTransport);
  // If BUNDLE was requested, then also assume RTP/RTCP mux
  answer.rtcpMux = true;
} else {
  var controller = new RTCIceTransportController();
  if (answer.rtcpMux){
    // The peer doesn't want BUNDLE, but it does want to multiplex
    // Now we need audio and video ICE transports
    // as well as an ICE Transport Controller object
    controller.addTransport(audioRtpIceTransport);
    controller.addTransport(videoRtpIceTransport);
    // Start the audio and video ICE transports
    audioRtpIceTransport.start(audioRtpIceGatherer,answer.audioRtpI
    videoRtpIceTransport.start(videoRtpIceGatherer,answer.videoRtpI
    // Start the audio and video DTLS transports
    audioRtpDtlsTransport.onerror = errorHandler;
    audioRtpDtlsTransport.start(answer.audioRtpDtls);
    videoRtpDtlsTransport.onerror = errorHandler;
    videoRtpDtlsTransport.start(answer.videoRtpDtls);
    // Replace the transport on the Sender and Receiver objects
    //
    audioSender.setTransport(audioRtpDtlsTransport);
    videoSender.setTransport(videoRtpDtlsTransport);
    audioReceiver.setTransport(audioRtpDtlsTransport);
    videoReceiver.setTransport(videoRtpDtlsTransport);
  } else {
    // We arrive here if the responder does not want BUNDLE
```

```
        // or RTP/RTCP multiplexing
        //
        // Now we need all the audio and video RTP and RTCP ICE transpo
        // as well as an ICE Transport Controller object
        controller.addTransport(audioRtpIceTransport);
        controller.addTransport(videoRtpIceTransport);
        // Start the ICE transports
        audioRtpIceTransport.start(audioRtpIceGatherer,answer.audioRtpI
        audioRtcpIceTransport.start(audioRtcpIceGatherer,answer.audioRt
        videoRtpIceTransport.start(videoRtpIceGatherer,answer.videoRtpI
        videoRtcpIceTransport.start(videoRtcpIceGatherer,answer.videoRt
        // Start the DTLS transports that are needed
        audioRtpDtlsTransport.start(answer.audioRtpDtls);
        audioRtcpDtlsTransport.start(answer.audioRtcpDtls);
        videoRtpDtlsTransport.start(answer.videoRtpDtls);
        videoRtcpDtlsTransport.start(answer.videoRtcpDtls);
        // Disable RTP/RTCP multiplexing
        audioSendParams.rtcp.mux = false;
        videoSendParams.rtcp.mux = false;
        audioRecvParams.rtcp.mux = false;
        videoRecvParams.rtcp.mux = false;
      }
    }
    // 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
```

8. The RTCRtpListener Object

The ***RTCRtpListener*** listens to RTP packets received from the **`RTCDtlsTransport`**, determining whether an incoming RTP stream is configured to be processed by an existing **`RTCRtpReceiver`** object. If no match is found, the `unhandledrtp` event is fired. This can be due to packets having an unknown SSRC, payload type or any other error that makes it impossible to attribute an RTP

packet to a specific **RTCRtpReceiver** object. The event is not fired once for each arriving packet; multiple discarded packets for the same SSRC SHOULD result in a single event.

Note that application handling of the `unhandledrtp` event may not be sufficient to enable the unhandled RTP stream to be rendered. The amount of buffering to be provided for unhandled RTP streams is not mandated by this specification and is recommended to be strictly limited to protect against denial of service attacks. Therefore an application attempting to create additional **RTCRtpReceiver** objects to handle the incoming RTP stream may find that portions of the incoming RTP stream were lost due to insufficient buffers, and therefore could not be rendered.

8.1 Overview

An **RTCRtpListener** instance is associated to an **RTCDtlsTransport**.

8.2 Operation

An **RTCRtpListener** instance is constructed from an **RTCDtlsTransport** object.

8.3 RTP matching rules

When the **RTCRtpListener** object receives an RTP packet over an **RTCDtlsTransport**, the **RTCRtpListener** attempts to determine which **RTCRtpReceiver** object to deliver the packet to, based on the values of the SSRC and payload type fields in the RTP header, as well as the value of the MID RTP header extension, if present.

The **RTCRtpListener** maintains three tables in order to facilitate matching: the *ssrc_table* which maps SSRC values to **RTCRtpReceiver** objects; the *muxId_table* which maps values of the MID header extension to **RTCRtpReceiver**objects and the *pt_table* which maps payload type values to **RTCRtpReceiver** objects.

For an **RTCRtpReceiver** object *receiver*, table entries are added when `receiver.receive()` is called, and are removed when `receiver.stop()` is called. If `receiver.receive()` is called again, all entries referencing *receiver* are removed prior to adding new entries.

SSRC table: *ssrc_table*[`parameters.encodings[i].ssrc`] is set to *receiver* for each entry where `parameters.encodings[i].ssrc` is set, for values of *i* from 0 to the number of

encodings. If *ssrc_table*[*ssrc*] is already set to a value other than *receiver*, then `receiver.receive()` will throw an `InvalidParameters` exception.

muxId table: If `parameters.muxId` is set, *muxId_table*[*parameters.muxId*] is set to *receiver*. If *muxId_table*[*muxId*] is already set to a value other than *receiver*, then `receiver.receive()` will throw an `InvalidParameters` exception.

payload type table: If *parameters.muxId* is unset and `parameters.encodings[i].ssrc` is unset for all values of *i* from 0 to the number of encodings, then add entries to *pt_table* by setting *pt_table*[`parameters.codecs[j].payloadType`] to *receiver*, for values of *j* from 0 to the number of codecs. If *pt_table*[*pt*] is already set to a value other than *receiver*, or `parameters.codecs[j].payloadType` is unset for any value of *j* from 0 to the number of codecs, then `receiver.receive()` will throw an `InvalidParameters` exception.

When an RTP packet arrives, if *ssrc_table*[*packet.ssrc*] is set: set *packet_receiver* to *ssrc_table*[*packet.ssrc*] and check whether the value of *packet.pt* is equal to one of the values of `parameters.codecs[j].payloadtype` for *packet_receiver*, where *j* varies from 0 to the number of codecs. If so, route the packet to *packet_receiver*. If *packet.pt* does not match, fire the `unhandledrtp` event.

Else if *packet.muxId* is set: If *muxId_table*[*packet.muxId*] is unset, fire the `unhandledrtp` event, else set *packet_receiver* to *muxId_table*[*packet.muxId*] and check whether the value of *packet.pt* is equal to one of the values of `parameters.codecs[j].payloadtype` for *packet_receiver*, where *j* varies from 0 to the number of codecs. If so, set *ssrc_table*[*packet.ssrc*] to *packet_receiver* and route the packet to *packet_receiver*. If *packet.pt* does not match, fire the `unhandledrtp` event.

Else if *pt_table*[*packet.pt*] is set: set *packet_receiver* to *pt_table*[*packet.pt*], set *ssrc_table*[*packet.ssrc*] to *packet_receiver*, set *pt_table*[*packet.pt*] to null and route the packet to *packet_receiver*. Question: Do we remove all *pt_table*[*packet.pt*] entries set to *packet_receiver*?

Else if no matches are found in the *ssrc_table*, *muxId_table* or *pt_table*, fire the `unhandledrtp` event.

TODO: Revise this paragraph based on the outcome of the discussion on FEC/RTX/RED.

8.4 Interface Definition

WebIDL

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

8.4.1 Attributes

onunhandledrtp of type EventHandler, nullable
: The event handler which handles the RTCRtpUnhandledRtpEvent, which is fired when the **RTCRtpListener** detects an RTP stream that is not configured to be processed by an existing **RTCRtpReceiver** object.

transport of type *RTCDtlsTransport*, readonly
: The RTP **RTCDtlsTransport** instance.

8.5 RTCRtpUnhandledEvent

The unhandledrtp event of the **RTCRtpListener** object uses the **RTCRtpUnhandledEvent** interface.

Firing an RTCRtpUnhandledEvent event named **e** means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the **RTCRtpUnhandledEvent** interface MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCRtpUnhandledEventInit : EventInit {
            DOMString     muxId;
            payloadtype   payloadType;
            unsigned long ssrc;
};

[Constructor(DOMString type, RTCRtpUnhandledEventInit eventInitDict)]
interface RTCRtpUnhandledEvent : Event {
    readonly        attribute DOMString     muxId;
    readonly        attribute payloadtype   payloadType;
    readonly        attribute unsigned long ssrc;
};
```

8.5.1 Attributes

`muxId` of type DOMString, readonly
: The value of the MID RTP header extension [BUNDLE] in the RTP stream triggering the `unhandledrtp` event.

`payloadType` of type *payloadtype*, readonly
: The Payload Type value in the RTP stream triggering the `unhandledrtp` event.

`ssrc` of type unsigned long, readonly
: The SSRC in the RTP stream triggering the `unhandledrtp` event.

8.5.2 Dictionary `RTCRtpUnhandledEventInit` Members

`muxId` of type DOMString
: If present, the value of the MID RTP header extension [BUNDLE] in the RTP stream triggering the `unhandledrtp` event.

`payloadType` of type *payloadtype*
: The Payload Type value in the RTP stream triggering the `unhandledrtp` event.

`ssrc` of type unsigned long
: The SSRC in the RTP stream triggering the `unhandledrtp` event.

8.6 Example

> EXAMPLE 12

9. Dictionaries related to Rtp

9.1 dictionary RTCRtpCapabilities

The ***RTCRtpCapabilities*** object expresses the capabilities of **RTCRtpSender** and **RTCRtpReceiver** objects. Features which are mandatory to implement in [RTP-USAGE], such as RTP/RTCP multiplexing [RFC5761], audio/video multiplexing [RTP-MULTI-STREAM] and reduced size RTCP [RFC5506] are assumed to be available and are therefore not included in **RTCRtpCapabilities**, although these features can be set via **RTCRtpParameters**.

WebIDL

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

9.1.1 Dictionary RTCRtpCapabilities Members

codecs of type sequence<*RTCRtpCodecCapability*>
: Supported codecs.

fecMechanisms of type sequence<DOMString>
: Supported Forward Error Correction (FEC) mechanisms. [FEC] summarizes requirements relating to FEC mechanisms.

headerExtensions of type sequence<*RTCRtpHeaderExtension*>
: Supported RTP header extensions.

9.2 dictionary RTCRtcpFeedback

RTCRtcpFeedback provides information on RTCP feedback messages.

WebIDL

```
dictionary RTCRtcpFeedback {
             DOMString type;
             DOMString parameter;
};
```

9.2.1 Dictionary RTCRtcpFeedback Members

`parameter` of type DOMString

For a *type* of "ack" or "nack", valid values for *parameters* are the "ack" and "nack" Attribute Values enumerated in [IANA-SDP-15] ("sli", "rpsi", etc.). For a *type* of "ccm", valid values for *parameters* are the "Codec Control Messages" enumerated in [IANA-SDP-19] ("fir", "tmmbr" (includes "tmmbn"), etc.).

`type` of type DOMString

Valid values for *type* are the "RTCP Feedback" Attribute Values enumerated in [IANA-SDP-14] ("ack", "ccm", "nack", etc.).

9.3 dictionary RTCRtpCodecCapability

RTCRtpCodecCapability provides information on the capabilities of a codec.

WebIDL

```
typedef object Dictionary;
```

WebIDL

```
typedef octet payloadtype;
```

WebIDL

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

9.3.1 Dictionary `RTCRtpCodecCapability` Members

`clockRate` of type unsigned long

Codec clock rate expressed in Hertz, null if unset.

`kind` of type DOMString
The media supported by the codec: "audio", "video" or "" for both.

`maxSpatialLayers` of type unsigned short, defaulting to **`0`**
Maximum number of spatial layer extensions supported by this codec (e.g. a value of 1 indicates support for up to 2 spatial layers). A value of 0 indicates no support for spatial scalability.

`maxTemporalLayers` of type unsigned short, defaulting to **`0`**
Maximum number of temporal layer extensions supported by this codec (e.g. a value of 1 indicates support for up to 2 temporal layers). A value of 0 indicates no support for temporal scalability.

`maxptime` of type unsigned long
The maximum packetization time supported by the **`RTCRtpReceiver`**.

`name` of type DOMString
The MIME media type. Valid types are listed in [IANA-RTP-2].

`numChannels` of type unsigned long
The number of channels supported (e.g. stereo). For video, this will be unset.

`options` of type *Dictionary*
Codec-specific parameters available for signaling.

`parameters` of type *Dictionary*
Codec-specific parameters that must be signaled to the remote party.

`preferredPayloadType` of type *payloadtype*
The preferred RTP payload type for the codec denoted by *RTCRtpCodecCapability.name*. This attribute was added to make it possible for the sender and receiver to pick a matching payload type when creating sender and receiver parameters. When returned by `RTCRtpSender.getCapabilities()`, *RTCRtpCapabilities.codecs.preferredPayloadtype* represents the preferred RTP payload type for sending. When returned by `RTCRtpReceiver.getCapabilities()`, *RTCRtpCapabilities.codecs.preferredPayloadtype* represents the preferred RTP payload type for receiving. To avoid payload type conflicts, each value of *RTCRtpCodecCapability.name* should have a unique value of *RTCRtpCodecCapability.preferredPayloadtype*.

`rtcpFeedback` of type sequence<*RTCRtcpFeedback*>

Transport layer and codec-specific feedback messages for this codec.

`svcMultiStreamSupport` of type boolean
Whether the implementation can send SVC layers utilizing distinct SSRCs. Unset for audio codecs. For video codecs, only set if the codec supports scalable video coding with multiple streams.

9.3.2 Codec capability parameters

The capability parameters for commonly implemented codecs are provided below.

9.3.2.1 Opus

The following capabilities are defined for Opus, as noted in [OPUS-RTP] Section 6.1:

Property Name	Values	Notes
maxplaybackrate	`unsigned long`	A hint about the maximum output sampling rate that the receiver is capable of rendering in Hz.
stereo	`boolean`	Specifies whether the decoder prefers receiving stereo (if true) or mono signals (if false).

9.3.2.2 VP8

The following receiver capabilities are defined for VP8, as noted in [VP8-RTP] Section 6.1:

Property Name	Values	Notes
max-fr	`unsigned long`	This capability indicates the maximum frame rate in frames per second that the decoder is capable of decoding.
max-fs	`unsigned long long`	This capability indicates the maximum frame size in macroblocks that the decoder is capable of decoding.

9.3.2.3 H.264

The following capabilities are defined for H.264, as noted in [RFC6184] Section 8.1, and [RTCWEB-VIDEO] Section 6.2.

Property Name	Values	Notes
profile-level-id	`unsigned long`	This parameter, defined in [RFC6184] Section 8.1, is mandatory to support, as noted in [RTCWEB-VIDEO] Section 6.2.
packetization-mode	`sequence<unsigned short>`	A sequence of unsigned shorts, each ranging from 0 to 2, indicating supported *packetization-mode* values. As noted in [RTCWEB-VIDEO] Section 6.2, support for *packetization-mode* 1 is mandatory.
max-mbps, max-smbps, max-fs, max-cpb, max-dpb, max-br	`unsigned long long`	As noted in [RTCWEB-VIDEO] Section 6.2, these optional parameters allow the implementation to specify that they can support certain features of H.264 at higher rates and values than those signalled with *profile-level-id*.

9.4 dictionary RTCRtpParameters

RTCRtpParameters contains the RTP stack settings.

WebIDL

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

9.4.1 Dictionary `RTCRtpParameters` Members

`codecs` of type sequence<*RTCRtpCodecParameters*>

The codecs to send or receive (could include RED [RFC2198], RTX [RFC4588] and CN [RFC3389]).

`encodings` of type sequence<*RTCRtpEncodingParameters*>
The "encodings" or "layers" to be used for things like simulcast, Scalable Video Coding, RTX, FEC, etc.

`headerExtensions` of type sequence<*RTCRtpHeaderExtensionParameters*>
Configured header extensions.

`muxId` of type DOMString, defaulting to `""`
The *muxId* assigned to the RTP stream, if any, empty string if unset. In an **`RTCRtpReceiver`** or **`RTCRtpSender`** object, this corresponds to MID RTP header extension defined in [BUNDLE]. This is a stable identifier that permits the track corresponding to an RTP stream to be identified, rather than relying on an SSRC. An SSRC is randomly generated and can change arbitrarily due to conflicts with other SSRCs, whereas the *muxId* has a value whose meaning can be defined in advance between RTP sender and receiver, assisting in RTP demultiplexing. Note that including *muxId* in **`RTCRtpParameters`** rather than in **`RTCRtpEncodingParameters`** implies that if it is desired to send simulcast streams with different *muxId* values for each stream, then multiple **`RTCRtpSender`** objects are needed.

`rtcp` of type *RTCRtcpParameters*
Parameters to configure RTCP.

9.5 dictionary RTCRtcpParameters

RTCRtcpParameters provides information on RTCP settings.

WebIDL

```
dictionary RTCRtcpParameters {
             unsigned long ssrc;
             DOMString     cname;
             boolean       reducedSize = false;
             boolean       mux = true;
};
```

9.5.1 Dictionary `RTCRtcpParameters` Members

cname of type DOMString

The Canonical Name (CNAME) used by RTCP (e.g. in SDES messages). Guidelines for CNAME generation are provided in [RTP-USAGE] Section 4.9 and [RFC7022]. By default, ORTC implementations SHOULD set the CNAME to be the same within all `RTCRtcpParameter` objects created within the same Javascript sandbox. For backward compatibility with WebRTC 1.0, applications MAY set the CNAME; if unset, the CNAME is chosen by the browser.

mux of type boolean, defaulting to **true**

Whether RTP and RTCP are multiplexed, as specified in [RFC5761]. The default is `true`. If set to `false`, the **RTCIceTransport** MUST have an associated **RTCIceTransport** object with a *component* of "RTCP", in which case RTCP will be sent on the associated **RTCIceTransport**.

reducedSize of type boolean, defaulting to **false**

Whether reduced size RTCP [RFC5506] is configured (if true) or compound RTCP as specified in [RFC3550] (if false). The default is `false`.

ssrc of type unsigned long

The SSRC to be used in the "SSRC of packet sender" field defined in [RFC3550] Section 6.4.2 (Receiver Report) and [RFC4585] Section 6.1 (Feedback Messages), as well as the "SSRC" field defined in [RFC3611] Section 2 (Extended Reports). This is only set for an **RTCRtpReceiver**. If unset, *ssrc* is chosen by the browser, and the chosen value is not reflected in `RTCRtcpParameters.ssrc`. If the browser chooses the *ssrc* it may change it in event of a collision, as described in [RFC3550].

9.6 dictionary RTCRtpCodecParameters

RTCRtpCodecParameters provides information on codec settings.

WebIDL

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

9.6.1 Dictionary `RTCRtpCodecParameters` Members

`clockRate` of type unsigned long
: Codec clock rate expressed in Hertz, null if unset.

`maxptime` of type unsigned long
: The maximum packetization time set on the **`RTCRtpSender`**. Not specified if unset.

`name` of type DOMString
: The MIME media type. Valid types are listed in [IANA-RTP-2]. The *name* MUST always be provided.

`numChannels` of type unsigned long
: The number of channels supported (e.g. stereo). If unset for audio, use the codec default. For video, this can be left unset.

`parameters` of type *Dictionary*
: Codec-specific parameters available for signaling.

`payloadType` of type *payloadtype*
: The value that goes in the RTP Payload Type Field [RFC3550]. The *payloadType* MUST always be provided.

`rtcpFeedback` of type sequence<*RTCRtcpFeedback*>
: Transport layer and codec-specific feedback messages for this codec.

9.6.2 Codec parameters

The capabilities for commonly implemented codecs described in Section 9.4.2, are also used as codec parameters, with `RTCRtpCodecCapability.parameters` of the receiver used as `RTCRtpCodecParameters.parameters` of the sender, and `RTCRtpCodecCapability.parameters` of the sender used as `RTCRtpCodecParameters.parameters` of the receiver, with the Property Name and Values unchanged.

9.7 dictionary RTCRtpEncodingParameters

WebIDL

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

9.7.1 Dictionary `RTCRtpEncodingParameters` Members

`active` of type boolean, defaulting to **`true`**

Whether the sender or receiver is active. If false, don't send any media right now. Disable is different than omitting the encoding; it can keep resources available to re-enable more quickly than re-adding. As noted in [RFC3264] Section 5.1, RTCP is still sent, regardless of the value of the active attribute. If unset, the default is assumed.

`codecPayloadType` of type *payloadtype*

For per-encoding codec specifications, give the codec Payload Type here. If unset, the browser will choose.

`dependencyEncodingIds` of type sequence<DOMString>

The `encodingId`s on which this layer depends. Within this specification `encodingId`s are permitted only within the same **`RTCRtpEncodingParameters`** sequence. In the future if MST were to be supported, then if searching within an **`RTCRtpEncodingParameters`** sequence did not produce a match, then a global search would be carried out.

`encodingId` of type DOMString

An identifier for the encoding object. This identifier should be unique within the scope of the localized sequence of **`RTCRtpEncodingParameters`** for any given **`RTCRtpParameters`** object. For a codec (such as VP8) where a compliant decoder is required to be able to decode anything that an encoder can send, it is not necessary to

specify the expected scalable video coding configuration on the receiver via use of *encodingId* (or *dependencyEncodingIds*). Where specified for a receiver, the expected layering is ignored. A sender MAY send fewer layers than what is specified in **RTCRtpEncodingParameters**, but MUST NOT send more.

fec of type *RTCRtpFecParameters*

Specifies the FEC mechanism if set.

framerateBias of type double, defaulting to **0.5**

What to give more bits to, if available. 0.0 = strongly favor resolution or 1.0 = strongly favor framerate. 0.5 = neither (default). For scalable video coding, this parameter is only relevant for the base layer. This parameter is ignored in an **RTCRtpReceiver** object. If unset, the default is assumed.

framerateScale of type double

Inverse of the input framerate fraction to be encoded. Example: 1.0 = full framerate, 2.0 = one half of the full framerate. For scalable video coding, *framerateScale* refers to the inverse of the aggregate fraction of input framerate achieved by this layer when combined with all dependent layers.

maxBitrate of type unsigned long long

Ramp up resolution/quality/framerate until this bitrate, if set. Summed when using dependent layers. This parameter is ignored in scalable video coding, or in an **RTCRtpReceiver** object. If unset, there is no maximum bitrate.

minQuality of type double, defaulting to **0**

Never send less than this quality. 1.0 = maximum attainable quality. For scalable video coding, this parameter is only relevant for the base layer. This parameter is ignored in an **RTCRtpReceiver** object.

priority of type double, defaulting to **1.0**

The higher the value, the more the bits will be given to each as available bandwidth goes up. Default is 1.0. For scalable video coding, this parameter is only relevant for the base layer. This parameter is ignored in an **RTCRtpReceiver** object. If unset, the default is assumed.

resolutionScale of type double

Inverse of the input resolution fraction to be encoded, or die trying. Example: 1.0 = full resolution, 2.0 = one half of the full resolution. For scalable video coding, *resolutionScale* refers to the inverse aggregate fraction of the input resolution achieved by this layer when combined with all dependent layers.

`rtx` of type *RTCRtpRtxParameters*

Specifies the RTX [RFC4588] parameters if set.

`ssrc` of type unsigned long

The SSRC for this layering/encoding. If *ssrc* is unset in a `RTCRtpEncodingParameters` object passed to the `RTCRtpReceiver.receive` method, the next unhandled SSRC will match, and an `RTCRtpUnhandledEvent` will not be fired. If *ssrc* is unset in a `RTCRtpEncodingParameters` object passed to the `RTCRtpSender.send` method, the browser will choose, and the chosen value is not reflected in `RTCRtpEncodingParameters.ssrc`. If the browser chooses the *ssrc*, it may change it due to a collision without firing an `RTCSsrcConflictEvent`. If *ssrc* is set in a `RTCRtpEncodingParameters` object passed to the `RTCRtpSender.send` method and an SSRC conflict is detected, then an `RTCSsrcConflictEvent` is fired (see Section 6.4).

9.8 Examples

9.8.1 Basic Example

EXAMPLE 13

```
// Send a thumbnail along with regular size, prioritizing the thumbnai
var encodings = [{ ssrc: 1,  priority: 1.0 }];
var encodings = [{ ssrc: 2, priority: 10.0 }];

// Sign Language (need high framerate, but don't get too bad quality)
var encodings = [{ minQuality: 0.2,  framerateBias: 1.0 }];

// Screencast (High quality, framerate can be low)
var encodings = [{ framerateBias: 0.0 }];

//Remote Desktop (High framerate, must not downscale)
var encodings = [{ framerateBias: 1.0 }];

// Audio more important than video
var audioEncodings = [{ priority: 10.0 }];
var videoEncodings = [{ priority: 0.1 }];

//Video more important than audio
var audioEncodings = [{ priority: 0.1 }];
var videoEncodings = [{ priority: 10.0 }];

//Crank up the quality
var encodings = [{ maxBitrate: 10000000 }];

//Keep the bandwidth low
var encodings = [{ maxBitrate: 100000 }];
```

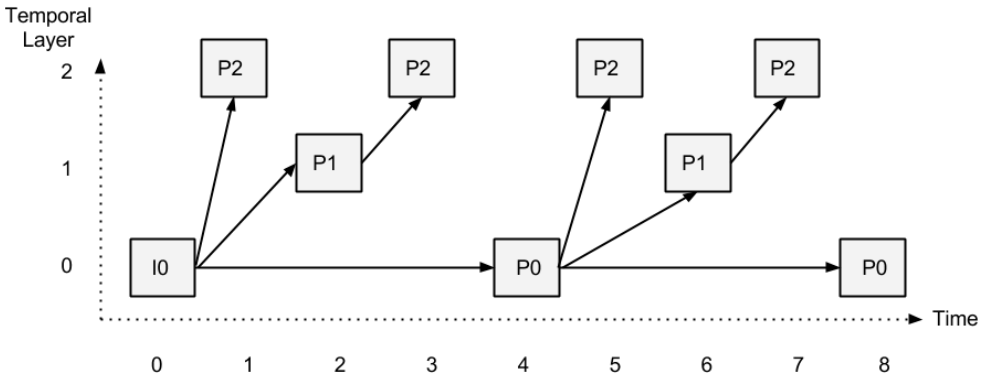
9.8.2 Temporal Scalability

EXAMPLE 14

```
// Example of 3-layer temporal scalability encoding
var encodings =[{
// Base framerate is one quarter of the input framerate
  encodingId: "0",
  framerateScale: 4.0
}, {
// Temporal enhancement (half the input framerate when combined with t
  encodingId: "1",
  dependencyEncodingIds: ["0"],
  framerateScale: 2.0
}, {
// Another temporal enhancement layer (full input framerate when all l
  encodingId: "2",
  dependencyEncodingIds: ["0", "1"],
  framerateScale: 1.0
}]

// Example of 3-layer temporal scalability with all but the base layer
var encodings =[{
  encodingId: "0",
  framerateScale: 4.0
}, {
  encodingId: "1",
  dependencyEncodingIds: ["0"],
  framerateScale: 2.0,
  active: false
}, {
  encodingId: "2",
  dependencyEncodingIds: ["0", "1"],
  framerateScale: 1.0,
  active: false
}];
```

Below is a representation of a 3-layer temporal scalability encoding. In the diagram, I0 is the base layer I-frame, and P0 represents base-layer P-frames. P1 represents the first temporal enhancement layer, and P2 represents the second temporal enhancement layer.



9.8.3 Spatial Simulcast

EXAMPLE 15

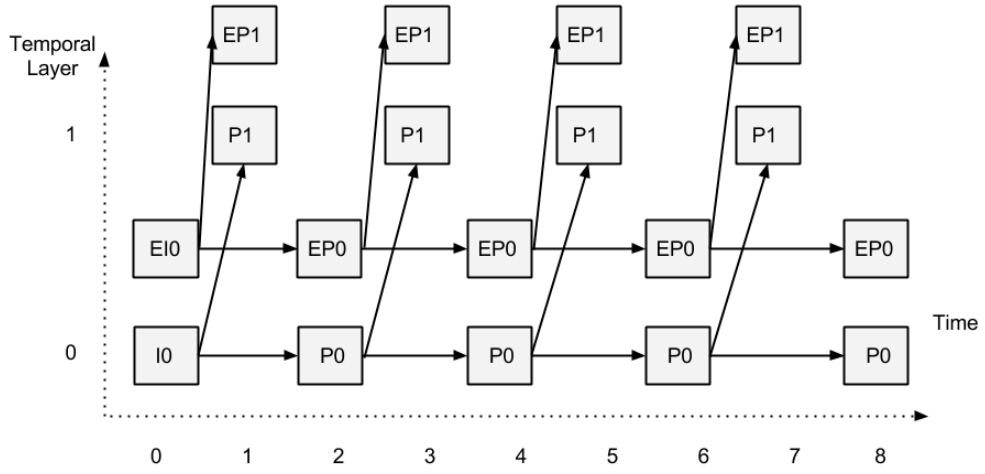
```
// Example of 3-layer spatial simulcast
var encodings =[{
// Simulcast layer at one quarter scale
  encodingId: "0",
  resolutionScale: 4.0
}, {
// Simulcast layer at one half scale
  encodingId: "1",
  resolutionScale: 2.0
}, {
// Simulcast layer at full scale
  encodingId: "2",
  resolutionScale: 1.0
}]

// Example of 3-layer spatial simulcast with all but the lowest resolu
var encodings =[{
  encodingId: "0",
  resolutionScale: 4.0
}, {
  encodingId: "1",
  resolutionScale: 2.0,
  active: false
}, {
  encodingId: "2",
  resolutionScale: 1.0,
  active: false
}];

// Example of 2-layer spatial simulcast combined with 2-layer temporal
var encodings =[{
// Low resolution base layer (half the input framerate, half the input
  encodingId: "0",
  resolutionScale: 2.0,
  framerateScale: 2.0
}, {
// Enhanced resolution Base layer (half the input framerate, full inpu
  encodingId: "E0",
  resolutionScale: 1.0,
  framerateScale: 2.0
}, {
// Temporal enhancement to the low resolution base layer (full input f
```

```
  encodingId: "1",
  dependencyEncodingIds: ["0"],
  resolutionScale: 2.0,
  framerateScale: 1.0
}, {
// Temporal enhancement to the enhanced resolution base layer (full in
  encodingId: "E1",
  dependencyEncodingIds: ["E0"],
  resolutionScale: 1.0,
  framerateScale: 1.0
}]
```

Below is a representation of 2-layer temporal scalability combined with 2-layer spatial simulcast. Solid arrows represent temporal prediction. In the diagram, I0 is the base-layer I-frame, and P0 represents base-layer P-frames. EI0 is an enhanced resolution base-layer I-frame, and EP0 represents P-frames within the enhanced resolution base layer. P1 represents the first temporal enhancement layer, and EP1 represents a temporal enhancement to the enhanced resolution simulcast base-layer.



9.8.4 Spatial Scalability

EXAMPLE 16

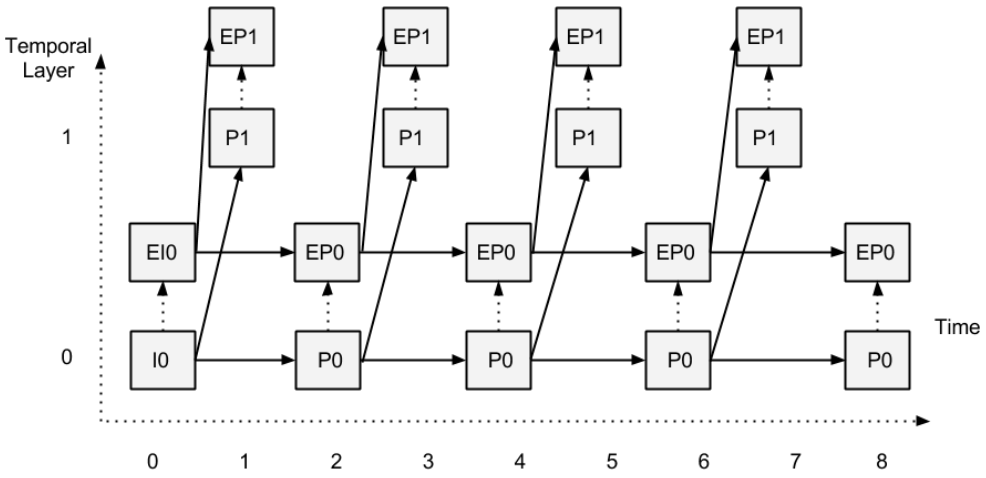
```
// Example of 3-layer spatial scalability encoding
var encodings =[{
// Base layer with one quarter input resolution
  encodingId: "0",
  resolutionScale: 4.0
}, {
// Spatial enhancement layer providing half input resolution when comb
  encodingId: "1",
  dependencyEncodingIds: ["0"],
  resolutionScale: 2.0
}, {
// Additional spatial enhancement layer providing full input resolutio
  encodingId: "2",
  dependencyEncodingIds: ["0", "1"],
  resolutionScale: 1.0
}]

// Example of 3-layer spatial scalability with all but the base layer
var encodings =[{
  encodingId: "0",
  resolutionScale: 4.0
}, {
  encodingId: "1",
  dependencyEncodingIds: ["0"],
  resolutionScale: 2.0,
  active: false
}, {
  encodingId: "2",
  dependencyEncodingIds: ["0", "1"],
  resolutionScale: 1.0,
  active: false
}];

// Example of 2-layer spatial scalability combined with 2-layer tempor
var encodings =[{
// Base layer (half input framerate, half resolution)
  encodingId: "0",
  resolutionScale: 2.0,
  framerateScale: 2.0
}, {
// Temporal enhancement to the base layer (full input framerate, half
  encodingId: "1",
```

```
  dependencyEncodingIds: ["0"],
  resolutionScale: 2.0,
  framerateScale: 1.0
}, {
// Spatial enhancement to the base layer (half input framerate, full r
  encodingId: "E0",
  dependencyEncodingIds: ["0"],
  resolutionScale: 1.0,
  framerateScale: 2.0
}, {
// Temporal enhancement to the spatial enhancement layer (full input f
  encodingId: "E1",
  dependencyEncodingIds: ["E0", "1"],
  resolutionScale: 1.0,
  framerateScale: 1.0
}]
```

Below is a representation of 2-layer temporal scalability combined with 2-layer spatial scalability. Solid arrows represent temporal prediction and dashed arrows represent inter-layer prediction. In the diagram, I0 is the base-layer I-frame, and EI0 is an intra spatial enhancement. P0 represents base-layer P-frames, and P1 represents the first temporal enhancement layer. EP0 represents a resolution enhancement to the base-layer P frames, and EP1 represents a resolution enhancement to the second temporal layer P-frames.



9.9 dictionary RTCRtpFecParameters

WebIDL

```
dictionary RTCRtpFecParameters {
             unsigned long ssrc;
             DOMString     mechanism;
};
```

9.9.1 Dictionary `RTCRtpFecParameters` Members

`mechanism` of type DOMString
: The Forward Error Correction (FEC) mechanism to use.

`ssrc` of type unsigned long
: The SSRC to use for FEC. If unset in an **`RTCRtpSender`** object, the browser will choose.

9.10 dictionary RTCRtpRtxParameters

WebIDL

```
dictionary RTCRtpRtxParameters {
             unsigned long ssrc;
};
```

9.10.1 Dictionary `RTCRtpRtxParameters` Members

`ssrc` of type unsigned long
: The SSRC to use for retransmission, as specified in [RFC4588]. If unset when passed to `RTCRtpSender.send()`, the browser will choose.

9.11 dictionary RTCRtpHeaderExtension

WebIDL

```
dictionary RTCRtpHeaderExtension {
             DOMString      kind;
             DOMString      uri;
             unsigned short preferredId;
             boolean        preferredEncrypt = false;
};
```

9.11.1 Dictionary `RTCRtpHeaderExtension` Members

`kind` of type DOMString
: The media supported by the header extension: "audio" for an audio codec, "video" for a video codec, and "" for both.

`preferredEncrypt` of type boolean, defaulting to **`false`**
: If true, it is preferred that the value in the header be encrypted as per [RFC6904]. Default is to prefer unencrypted.

`preferredId` of type unsigned short
: The preferred ID value that goes in the packet.

`uri` of type DOMString
: The URI of the RTP header extension, as defined in [RFC5285].

9.12 dictionary RTCRtpHeaderExtensionParameters

WebIDL

```
dictionary RTCRtpHeaderExtensionParameters {
             DOMString      uri;
             unsigned short id;
             boolean        encrypt = false;
};
```

9.12.1 Dictionary `RTCRtpHeaderExtensionParameters` Members

`encrypt` of type boolean, defaulting to **`false`**
: If true, the value in the header is encrypted as per [RFC6904]. Default is unencrypted.

`id` of type unsigned short
: The value that goes in the packet.

`uri` of type DOMString
: The URI of the RTP header extension, as defined in [RFC5285].

9.13 RTP header extensions

Registered RTP header extensions are listed in [IANA-RTP-10]. Header extensions mentioned in [RTP-USAGE] include:

Header Extension	Reference	Notes
Rapid Synchronization	[RFC6051]	This extension enables carriage of an NTP-format timestamp, as defined in [RFC6051] Section 3.3.
Client-to-Mixer Audio Level	[RFC6464]	This extension indicates the audio level of the audio sample carried in an RTP packet.
Mixer-to-Client Audio Level	[RFC6465]	This extension indicates the audio level of individual conference participants.
MID	[BUNDLE]	This extension defines a track identifier which can be used to identify the track corresponding to an RTP stream.

10. The RTCDtmfSender Object

> NOTE
> This section of the ORTC API specification depends on the WebRTC 1.0 DtmfSender API, and needs to be synchronized once it is updated.

10.1 Overview

An ***RTCDtmfSender*** instance allows sending DTMF tones to/from the remote peer, as per [RFC4733].

10.2 Operation

An **RTCDtmfSender** object is constructed from an **RTCRtpSender** object.

10.3 Interface Definition

WebIDL

```
[Constructor(RTCRtpSender sender)]
interface RTCDtmfSender {
    readonly        attribute boolean      canInsertDTMF;
    void insertDTMF (DOMString tones, optional long duration,
optional long interToneGap);
    readonly        attribute RTCRtpSender sender;
                    attribute EventHandler ontonechange;
    readonly        attribute DOMString    toneBuffer;
    readonly        attribute long         duration;
    readonly        attribute long         interToneGap;
};
```

10.3.1 Attributes

canInsertDTMF of type boolean, readonly

Whether the **RTCDtmfSender** is capable of sending DTMF.

duration of type long, readonly

The ***duration*** attribute returns the current tone duration value in milliseconds. This value will be the value last set via the `insertDTMF()` method, or the default value of 100 ms if `insertDTMF()` was called without specifying the duration.

interToneGap of type long, readonly

The ***interToneGap*** attribute returns the current value of the between-tone gap. This value will be the value last set via the `insertDTMF()` method, or the default value of 70 ms if `insertDTMF()` was called without specifying the `interToneGap.`

ontonechange of type EventHandler

The ***ontonechange*** event handler uses the **RTCDTMFToneChangeEvent** interface to return the character for each tone as it is played out.

sender of type *RTCRtpSender*, readonly

The **RTCRtpSender** instance

`toneBuffer` of type DOMString, readonly

The ***toneBuffer*** attribute returns a list of the tones remaining to be played out.

10.3.2 Methods

`insertDTMF`

The `insertDTMF()` method is used to send DTMF tones. Since DTMF tones cannot be sent without configuring the DTMF codec, if `insertDTMF()` is called prior to `sender.send(parameters)`, or if `sender.send(parameters)` was called but *parameters* did not include the DTMF codec, throw an `InvalidStateError` exception.

The tones parameter is treated as a series of characters. The characters 0 through 9, A through D, #, and * generate the associated DTMF tones. The characters a to d are equivalent to A to D. The character ',' indicates a delay of 2 seconds before processing the next character in the tones parameter. All other characters MUST be considered ***unrecognized***.

The duration parameter indicates the duration in ms to use for each character passed in the tones parameters. The duration cannot be more than 6000 ms or less than 40 ms. The default duration is 100 ms for each tone.

The interToneGap parameter indicates the gap between tones. It MUST be at least 30 ms. The default value is 70 ms.

The browser MAY increase the duration and interToneGap times to cause the times that DTMF start and stop to align with the boundaries of RTP packets but it MUST not increase either of them by more than the duration of a single RTP audio packet.

When the `insertDTMF()` method is invoked, the user agent MUST run the following steps:

1. Set the object's `toneBuffer` attribute to the value of the first argument, the `duration` attribute to the value of the second argument, and the `interToneGap` attribute to the value of the third argument.
2. If `toneBuffer` contains any unrecognized characters, throw an `InvalidCharacterError` exception and abort these steps.
3. If `toneBuffer` is an empty string, return.

4. If the value of the `duration` attribute is less than 40, set it to 40. If, on the other hand, the value is greater than 6000, set it to 6000.
5. If the value of the `interToneGap` attribute is less than 30, set it to 30.
6. If a *Playout task* is scheduled to be run; abort these steps; otherwise queue a task that runs the following steps (*Playout task*):
    1. If `toneBuffer` is an empty string, fire an event named `tonechange` with an empty string at the **`RTCDtmfSender`** object and abort these steps.
    2. Remove the first character from `toneBuffer` and let that character be *tone*.
    3. Start playout of *tone* for `duration` ms on the associated RTP media stream, using the appropriate codec.
    4. Queue a task to be executed in `duration` + `interToneGap` ms from now that runs the steps labelled *Playout task*.
    5. Fire an event named `tonechange` with a string consisting of *tone* at the **`RTCDtmfSender`** object.

Calling `insertDTMF()` with an empty tones parameter can be used to cancel all tones queued to play after the currently playing tone.

Parameter	Type	Nullable	Optional	Description
tones	`DOMString`	✘	✘	
duration	`long`	✘	✔	
interToneGap	`long`	✘	✔	

Return type: `void`

10.4 RTCDTMFToneChangeEvent

The tonechange event uses the **`RTCDTMFToneChangeEvent`** interface.

Firing an tonechange event named *e* with a DOMString *tone* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the **`RTCDTMFToneChangeEvent`** interface with the tone attribute set to *tone*, MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCDTMFToneChangeEventInit : EventInit {
             DOMString tone = "";
};

[Constructor(DOMString type, RTCDTMFToneChangeEventInit
eventInitDict)]
interface RTCDTMFToneChangeEvent : Event {
    readonly        attribute DOMString tone;
};
```

10.4.1 Attributes

tone of type DOMString, readonly

The ***tone*** attribute contains the character for the tone that has just begun playout (see `insertDTMF()`). If the value is the empty string, it indicates that the previous tone has completed playback.

10.4.2 Dictionary `RTCDTMFToneChangeEventInit` Members

tone of type DOMString, defaulting to `""`

The tone parameter is treated as a series of characters. The characters 0 through 9, A through D, #, and * generate the associated DTMF tones. The characters a to d are equivalent to A to D. The character ',' indicates a delay of 2 seconds before processing the next character in the tones parameter. Unrecognized characters are ignored.

10.5 DTMF Example

Examples assume that *sendObject* is an **`RTCRtpSender`** object.

Sending the DTMF signal "1234" with 500 ms duration per tone:

EXAMPLE 17

```
var sender = new RTCDtmfSender(sendObject);
if (sender.canInsertDTMF) {
    var duration = 500;
    sender.insertDTMF("1234", duration);
} else {
    trace("DTMF function not available");
}
```

Send the DTMF signal "1234", and light up the active key using `lightKey(key)` while the tone is playing (assuming that `lightKey("")` will darken all the keys):

EXAMPLE 18

```
var sender = new RTCDtmfSender(sendObject);
sender.ontonechange = function (e) {
    if (!e.tone)
        return;
    // light up the key when playout starts
    lightKey(e.tone);
    // turn off the light after tone duration
    setTimeout(lightKey, sender.duration, "");
};
sender.insertDTMF("1234");
```

Send a 1-second "1" tone followed by a 2-second "2" tone:

EXAMPLE 19

```
var sender = new RTCDtmfSender(sendObject);
sender.ontonechange = function (e) {
    if (e.tone == "1")
        sender.insertDTMF("2", 2000);
};
sender.insertDTMF("1", 1000);
```

It is always safe to append to the tone buffer. This example appends before any tone playout has started as well as during playout.

EXAMPLE 20

```
var sender = new RTCDtmfSender(sendObject);
sender.insertDTMF("123");
// append more tones to the tone buffer before playout has begun
sender.insertDTMF(sender.toneBuffer + "456");

sender.ontonechange = function (e) {
    if (e.tone == "1")
        // append more tones when playout has begun
        sender.insertDTMF(sender.toneBuffer + "789");
};
```

Send the DTMF signal "123" and abort after sending "2".

EXAMPLE 21

```
var sender = new RTCDtmfSender(sendObject);
sender.ontonechange = function (e) {
    if (e.tone == "2")
        // empty the buffer to not play any tone after "2"
        sender.insertDTMF("");
};
sender.insertDTMF("123");
```

11. The RTCDataChannel Object

11.1 Overview

An ***RTCDataChannel*** class instance allows sending data messages to/from the remote peer.

11.2 Operation

An **RTCDataChannel** object is constructed from an **RTCDataTransport** object and an **RTCDataChannelParameters** object.

11.3 Interface Definition

The **RTCDataChannel** interface represents a bi-directional data channel between two peers. There are two ways to establish a connection with **RTCDataChannel**. The first way is to construct an **RTCDataChannel** at one of the peers with the **RTCDataChannelParameters**.negotiated attribute unset or set to its default value false. This will announce the new channel in-band and trigger an ondatachannel event with the corresponding **RTCDataChannel** object at the other peer. The second way is to let the application negotiate the **RTCDataChannel**. To do this, create an **RTCDataChannel** object with the **RTCDataChannelParameters**.negotiated dictionary member set to true, and signal out-of-band (e.g. via a web server) to the other side that it should create a corresponding **RTCDataChannel** with the **RTCDataChannelParameters**.negotiated dictionary member set to true and the same id. This will connect the two separately created **RTCDataChannel** objects. The second way makes it possible to create channels with asymmetric properties and to create channels in a declarative way by specifying matching ids. Each **RTCDataChannel** has an associated ***underlying data transport*** that is used to transport actual data to the other peer. The transport properties of the underlying data transport, such as in order delivery settings and reliability mode, are configured by the peer as the channel is created. The properties of a channel cannot change after the channel has been created.

WebIDL

```
[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);
};
```

11.3.1 Attributes

`binaryType` of type DOMString

The ***`binaryType`*** attribute MUST, on getting, return the value to which it was last set. On setting, the user agent MUST set the IDL attribute to the new value. When an **`RTCDataChannel`** object is constructed, the `binaryType` attribute MUST be initialized to the string 'blob'. This attribute controls how binary data is exposed to scripts. See the [WEBSOCKETS-API] for more information.

`bufferedAmount` of type unsigned long, readonly

The ***`bufferedAmount`*** attribute MUST return the number of bytes of application data (UTF-8 text and binary data) that have been queued using send() but that, as of the last time the event loop started executing a task, had not yet been transmitted to the network. This includes any text sent during the execution of the current task, regardless of whether the user agent is able to transmit text asynchronously with script execution. This does not include framing overhead incurred by the protocol, or buffering done by the operating system or network hardware. If the channel is closed, this attribute's value will only increase with each call to the send() method (the attribute does not reset to zero once the channel closes).

`onclose` of type EventHandler

This event handler, of event handler type `close`, MUST be supported by all objects implementing the RTCDataChannel interface.

`onerror` of type EventHandler

This event handler, of event handler type `error`, MUST be supported by all objects implementing the RTCDataChannel interface.

`onmessage` of type EventHandler

This event handler, of event handler event type `message`, MUST be fired to allow a developer's JavaScript to receive data from a remote peer.

Event Argument	*Description*
Object data	The received remote data.

`onopen` of type EventHandler

This event handler, of event handler type `open`, MUST be supported by all objects implementing the RTCDataChannel interface.

`parameters` of type *RTCDataChannelParameters*, readonly

The parameters applying to this data channel.

`readyState` of type *RTCDataChannelState*, readonly

The ***`readyState`*** attribute represents the state of the **`RTCDataChannel`** object. It MUST return the value to which the user agent last set it (as defined by the processing model algorithms).

`transport` of type *RTCDataTransport*, readonly

The readonly attribute referring to the related transport object.

11.3.2 Methods

`close`

Closes the **`RTCDataChannel`**. It may be called regardless of whether the **`RTCDataChannel`** object was created by this peer or the remote peer. When the `close()` method is called, the user agent MUST run the following steps:

1. Let channel be the **`RTCDataChannel`** object which is about to be closed.

2. If channel's `readyState` is closing or closed, then abort these steps.

3. Set channel's `readyState` attribute to closing.

4. If the closing procedure has not started yet, start it.

No parameters.

Return type: `void`

`send`

Run the steps described by the `send()` algorithm with argument type `string` object.

Parameter	Type	Nullable	Optional	Description
data	`DOMString`	✘	✘	

Return type: `void`

`send`

Run the steps described by the `send()` algorithm with argument type `Blob` object.

Parameter	Type	Nullable	Optional	Description
data	`Blob`	✘	✘	

Return type: `void`

send

Run the steps described by the `send()` algorithm with argument type `ArrayBuffer` object.

Parameter	Type	Nullable	Optional	Description
data	**ArrayBuffer**	✘	✘	

Return type: `void`

send

Run the steps described by the `send()` algorithm with argument type `ArrayBufferView` object.

Parameter	Type	Nullable	Optional	Description
data	ArrayBufferView	✘	✘	

Return type: `void`

11.4 Interface Definition

WebIDL

```
interface RTCDataTransport : RTCStatsProvider {
};
```

11.5 enum RTCDataChannelState

WebIDL

```
enum RTCDataChannelState {
    "connecting",
    "open",
    "closing",
    "closed"
};
```

Enumeration description

`connecting`	The user agent is attempting to establish the underlying data transport. This is the initial state of an **`RTCDataChannel`** object.
`open`	The underlying data transport is established and communication is possible. This is the initial state of an **`RTCDataChannel`** object dispatched as a part of an RTCDataChannelEvent.
`closing`	The procedure to close down the underlying data transport has started.
`closed`	The underlying data transport has been closed or could not be established.

11.6 dictionary RTCDataChannelParameters

An **`RTCDataChannel`** can be configured to operate in different reliability modes. A reliable channel ensures that the data is delivered at the other peer through retransmissions. An unreliable channel is configured to either limit the number of retransmissions (maxRetransmits) or set a time during which transmissions (including retransmissions) are allowed (maxPacketLifeTime). These properties can not be used simultaneously and an attempt to do so will result in an error. Not setting any of these properties results in a reliable channel.

WebIDL

```
dictionary RTCDataChannelParameters {
             DOMString      label = "";
             boolean        ordered = true;
             unsigned short maxPacketLifetime;
             unsigned short maxRetransmits;
             DOMString      protocol = "";
             boolean        negotiated = false;
             unsigned short id;
};
```

11.6.1 Dictionary `RTCDataChannelParameters` Members

`id` of type unsigned short

The id attribute returns the id for this **RTCDataChannel**, or null if unset. The id was either assigned by the user agent at channel creation time or was selected by the script. For SCTP, the id represents a stream identifier, as discussed in [DATA] Section 6.5. The attribute MUST return the value to which it was set when the **RTCDataChannel** was constructed.

label of type DOMString, defaulting to `""`

The ***label*** attribute represents a label that can be used to distinguish this **RTCDataChannel** object from other **RTCDataChannel** objects. The attribute MUST return the value to which it was set when the **RTCDataChannel** object was constructed. For an SCTP data channel, the label is carried in the DATA_CHANNEL_OPEN message defined in [DATA-PROT] Section 5.1.

maxPacketLifetime of type unsigned short

The ***maxPacketLifetime*** attribute represents the length of the time window (in milliseconds) during which retransmissions may occur in unreliable mode, or null if unset. The attribute MUST return the value to which it was set when the **RTCDataChannel** was constructed.

maxRetransmits of type unsigned short

The ***maxRetransmits*** attribute returns the maximum number of retransmissions that are attempted in unreliable mode, or null if unset. The attribute MUST be initialized to null by default and MUST return the value to which it was set when the **RTCDataChannel** was constructed.

negotiated of type boolean, defaulting to **false**

The ***negotiated*** attribute returns true if this **RTCDataChannel** was negotiated by the application, or false otherwise. The attribute MUST be initialized to `false` by default and MUST return the value to which it was set when the **RTCDataChannel** was constructed. If set to true, the application developer MUST signal to the remote peer to construct an **RTCDataChannel** object with the same id for the data channel to be open. If set to false, the remote party will receive an ondatachannel event with a system constructed **RTCDataChannel** object.

ordered of type boolean, defaulting to **true**

The ***ordered*** attribute returns true if the **RTCDataChannel** is ordered, and false if out of order delivery is allowed. Default is true. The attribute MUST return the value to which it was set when the **RTCDataChannel** was constructed.

protocol of type DOMString, defaulting to `""`

The name of the sub-protocol used with this **RTCDataChannel** if any, or the empty string otherwise (in which case the protocol is unspecified). The attribute MUST return the value to which it was set when the **RTCDataChannel** was constucted. Sub-protocols are registered in the 'Websocket Subprotocol Name Registry' created in [RFC6455] Section 11.5.

12. The RTCSctpTransport Object

The ***RTCSctpTransport*** includes information relating to Stream Control Transmission Protocol (SCTP) transport.

12.1 Overview

An **RTCSctpTransport** inherits from an **RTCDataTransport** object, which is associated to an **RTCDataChannel** object.

12.2 Operation

An **RTCSctpTransport** is constructed from an **RTCDtlsTransport** object.

12.3 Interface Definition

WebIDL

```
[Constructor(RTCDtlsTransport transport)]
interface RTCSctpTransport : RTCDataTransport {
    readonly        attribute RTCDtlsTransport transport;
    static RTCSctpCapabilities getCapabilities ();
    void                       start (RTCSctpCapabilities
remoteCaps);
    void                       stop ();
                    attribute EventHandler     ondatachannel;
};
```

12.3.1 Attributes

ondatachannel of type EventHandler

The ***ondatachannel*** event handler, of type `datachannel`, MUST be supported by all objects implementing the **RTCSctpTransport** interface. If the remote peers sets **RTCDataChannelParameters**.`negotiated` to false, then the event will fire indicating a new **RTCDataChannel** object has been constructed to connect with the **RTCDataChannel** constructed by the remote peer.

transport of type *RTCDtlsTransport*, readonly

The **RTCDtlsTransport** instance the **RTCSctpTransport** object is sending over.

12.3.2 Methods

getCapabilities, static

Retrieves the **RTCSctpCapabilities** of the **RTCSctpTransport** instance.

No parameters.

Return type: **RTCSctpCapabilities**

start

Parameter	Type	Nullable	Optional	Description
remoteCaps	**RTCSctpCapabilities**	✘	✘	

Return type: `void`

stop

Stops the **RTCSctpTransport** instance.

No parameters.

Return type: `void`

12.3.3 dictionary RTCSctpCapabilities

WebIDL

```
dictionary RTCSctpCapabilities {
             unsigned short maxMessageSize;
};
```

*12.3.3.1 Dictionary **RTCSctpCapabilities** Members*

maxMessageSize of type unsigned short
: Maximum message size.

12.4 RTCDataChannelEvent

The datachannel event uses the **RTCDataChannelEvent** interface.

Firing a datachannel event named **e** with a **RTCDataChannel** *channel* means that an event with the name *e*, which does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated), and which uses the **RTCDataChannelEvent** interface with the channel attribute set to *channel*, MUST be created and dispatched at the given target.

WebIDL

```
dictionary RTCDataChannelEventInit : EventInit {
             RTCDataChannel channel;
};

[Constructor(DOMString type, RTCDataChannelEventInit eventInitDict)]
interface RTCDataChannelEvent : Event {
    readonly        attribute RTCDataChannel channel;
};
```

12.4.1 Attributes

channel of type *RTCDataChannel*, readonly
: The ***channel*** attribute represents the **RTCDataChannel** object associated with the event.

12.4.2 Dictionary **RTCDataChannelEventInit** Members

channel of type *RTCDataChannel*
: TODO

12.5 Example

EXAMPLE 22

```
function initiate(signaller) {
  // Prepare the ICE gatherer
  var gatherOptions = new RTCIceGatherOptions();
  gatherOptions.gatherPolicy = RTCIceGatherPolicy.all;
  gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , {
  var iceGatherer = new RTCIceGatherer(gatherOptions);
  iceGatherer.onlocalcandidate = function(event) {
    mySignaller.mySendLocalCandidate(event.candidate);
  };
  // Create ICE and DTLS transports
  var ice = new RTCIceTransport(iceGatherer);
  var dtls = new RTCDtlsTransport(ice);
  // Prepare to handle remote ICE candidates
  mySignaller.onRemoteCandidate = function(remote){
    ice.addRemoteCandidate(remote.candidate);
  };
  var sctp = new RTCSctpTransport(dtls);
  // Construct RTCDataChannelParameters object
  var parameters = new RTCDataChannelParameters();

  signaller.sendInitiate({
    // ... include ICE/DTLS info from other example.
  "sctpCapabilities": RTCSctpTransport.getCapabilities()
  }, function(remote) {
    sctp.start(remote.sctpCapabilities);
  });

  var channel = new RTCDataChannel (sctp, parameters);
  channel.send("foo");
}

function accept(signaller, remote) {
  var gatherOptions = new RTCIceGatherOptions();
  gatherOptions.gatherPolicy = RTCIceGatherPolicy.all;
  gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , {
  var iceGatherer = new RTCIceGatherer(gatherOptions);
  iceGatherer.onlocalcandidate = function(event) {
    mySignaller.mySendLocalCandidate(event.candidate);
  };
  // Create ICE and DTLS transports
  var ice = new RTCIceTransport(iceGatherer);
  var dtls = new RTCDtlsTransport(ice);
```

```
  // Prepare to handle remote candidates
  mySignaller.onRemoteCandidate = function(remote) {
    ice.addRemoteCandidate(remote.candidate);
  };
  signaller.sendAccept({
    // ... include ICE/DTLS info from other example.
    "sctpCapabilities": RTCSctpTransport.getCapabilities()
  });

  var sctp = new RTCSctpTransport(dtls);
  sctp.start(remote.sctpCapabilties);

  // Assume in-band signalling.  We could also easily add
  // RTCDataChannelParameters into the out-of-band signalling
  // And construct the data channel with with negotiated: true.

  sctp.ondatachannel = function(channel) {
    channel.onmessage = function(message) {
    if (message == "foo") {
      channel.send("bar");
     }
    };
  };
}
```

13. Statistics API

The Statistics API enables retrieval of statistics relating to **RTCRtpSender**, **RTCRtpReceiver**, **RTCDtlsTransport**, **RTCIceGatherer**, **RTCIceTransport** and **RTCSctpTransport** objects. For detailed information on the Statistics API, consult [WEBRTC-STATS].

WebIDL

```
interface RTCStatsProvider {
    Promise<RTCStatsReport> getStats ();
};
```

13.1 Methods

getStats

Gathers stats for the given object and reports the result asynchronously. If the object has not yet begun to send or receive data, the returned stats will reflect this. If the object is in the closed state, the returned stats will reflect the stats at the time the object transitioned to the closed state.

When the ***getStats()*** method is invoked, the user agent MUST queue a task to run the following steps:

1. Let *p* be a new promise.
2. Return, but continue the following steps in the background.
3. Start gathering the stats.
4. When the relevant stats have been gathered, return a new **RTCStatsReport** object, representing the gathered stats.

No parameters.
Return type: Promise<**RTCStatsReport**>

13.2 RTCStatsReport Object

The getStats() method delivers a successful result in the form of a **RTCStatsReport** object. A **RTCStatsReport** object represents a map between strings, identifying the inspected objects (RTCStats.id), and their corresponding **RTCStats** objects.

An **RTCStatsReport** may be composed of several **RTCStats** objects, each reporting stats for one underlying object. One achieves the total for the object by summing over all stats of a certain type; for instance, if an **RTCRtpSender** object is sending RTP streams involving multiple SSRCs over the network, the **RTCStatsReport** may contain one RTCStats object per SSRC (which can be distinguished by the value of the *ssrc* stats attribute).

WebIDL

```
interface RTCStatsReport {
    getter RTCStats (DOMString id);
};
```

13.2.1 Methods

`RTCStats`

Getter to retrieve the **`RTCStats`** objects that this stats report is composed of.

The set of supported property names [WEBIDL] is defined as the ids of all the **`RTCStats`** objects that has been generated for this stats report. The order of the property names is left to the user agent.

Parameter	Type	Nullable	Optional	Description
id	`DOMString`	✘	✘	

Return type: `getter`

13.3 RTCStats Dictionary

An **`RTCStats`** dictionary represents the stats gathered by inspecting a specific object. The **`RTCStats`** dictionary is a base type that specifies as set of default attributes, such as `timestamp` and `type`. Specific stats are added by extending the **`RTCStats`** dictionary.

Note that while stats names are standardized, any given implementation may be using experimental values or values not yet known to the Web application. Thus, applications MUST be prepared to deal with unknown stats.

Statistics need to be synchronized with each other in order to yield reasonable values in computation; for instance, if "bytesSent" and "packetsSent" are both reported, they both need to be reported over the same interval, so that "average packet size" can be computed as "bytes / packets" - if the intervals are different, this will yield errors. Thus implementations MUST return synchronized values for all stats in a **`RTCStats`** object.

WebIDL

```
dictionary RTCStats {
             DOMHiResTimeStamp timestamp;
             RTCStatsType      type;
             DOMString         id;
};
```

13.3.1 Dictionary `RTCStats` Members

id of type DOMString

A unique `id` that is associated with the object that was inspected to produce this `RTCStats` object. Two `RTCStats` objects, extracted from two different `RTCStatsReport` objects, MUST have the same `id` if they were produced by inspecting the same underlying object. User agents are free to pick any format for the `id` as long as it meets the requirements above.

timestamp of type DOMHiResTimeStamp

The ***timestamp***, of type `DOMHiResTimeStamp` [HIGHRES-TIME], associated with this object. The time is relative to the UNIX epoch (Jan 1, 1970, UTC). The timestamp for local measurements corresponds to the to the local clock and for remote measurements corresponds to the timestamp indicated in the incoming RTCP Sender Report (SR), Receiver Report (RR) or Extended Report (XR).

type of type RTCStatsType

The type of this object.

The ***type*** attribute MUST be initialized to the name of the most specific type this **RTCStats** dictionary represents.

13.3.2 RTCStatsType DOMString

RTCStatsType is equal to one of the following strings defined in [IANA-TOBE]:

"inboundrtp"

Statistics for the inbound RTP stream. It is accessed via the `RTCInboundRTPStreamStats` defined in [WEBRTC-STATS] Section 4.2.3. Local inbound RTP statistics can be obtained from the **RTCRtpReceiver** object; remote inbound RTP statistics can be obtained from the **RTCRtpSender** object.

"outboundrtp"

Statistics for the outbound RTP stream. It is accessed via the `RTCOutboundRTPStreamStats` defined in [WEBRTC-STATS] Section 4.2.4. Local outbound RTP statistics can be obtained from the **RTCRtpSender** object; remote outbound RTP statistics can be obtained from the **RTCRtpReceiver** object.

"session"

Statistics relating to **RTCDataChannel** objects. It is accessed via the `RTCPeerConnectionStats` defined in [WEBRTC-STATS] Section 4.3.

"datachannel"
: Statistics relating to each `RTCDataChannel` id. It is accessed via the `RTCDataChannelStats` defined in [WEBRTC-STATS] Section 4.5.

"track"
: Statistics relating to the `MediaStreamTrack` object. It is accessed via the `RTCMediaStreamTrackStats` defined in [WEBRTC-STATS] Section 4.4.2.

"transport"
: Transport statistics related to the **`RTCDtlsTransport`** object. It is accessed via the `RTCTransportStats` and `RTCCertificateStats` defined in [WEBRTC-STATS] Sections 4.6 and 4.9.

"candidatepair"
: ICE candidate pair statistics related to **`RTCIceTransport`** objects. It is accessed via the `RTCIceCandidatePairStats` defined in [WEBRTC-STATS] Section 4.8.

"localcandidate"
: ICE local candidates, related to **`RTCIceGatherer`** objects. It is accessed via the `RTCIceCandidateAttributes` defined in [WEBRTC-STATS] Section 4.7.

"remotecandidate"
: ICE remote candidate, related to **`RTCIceTransport`** objects. It is accessed via the `RTCIceCandidateAttributes` defined in [WEBRTC-STATS] Section 4.7.

13.4 RTCP matching rules

Since statistics are retrieved from objects within the ORTC API, and information within RTCP packets is used to maintain some of the statistics, the handling of RTCP packets is important to the operation of the statistics API.

RTCP packets arriving on an **`RTCDtlsTransport`** are decrypted and a notification is sent to all **`RTCRtpSender`** and **`RTCRtpReceiver`** objects utilizing that transport. **`RTCRtpSender`** and **`RTCRtpReceiver`** objects then examine the RTCP packets to determine the information relevant to their operation and the statistics maintained by them.

RTCP packets should be queued for 30 seconds and all **`RTCRtpSender`** and **`RTCRtpReceiver`** objects on the related `RTCDTlsTransport` have access to those packets until the packet is removed from the queue, should the **`RTCRtpSender`** or **`RTCRtpReceiver`** objects need to examine them.

Relevant SSRC fields within selected RTCP packets are summarized within [RFC3550] Section 6.4.1 (Sender Report), Section 6.4.2 (Receiver Report), Section 6.5 (SDES), Section 6.6 (BYE), [RFC4585] Section 6.1 (Feedback Messages), and [RFC3611] Section 2 (Extended Reports).

13.5 Example

Consider the case where the user is experiencing bad sound and the application wants to determine if the cause of it is packet loss. The following example code might be used:

EXAMPLE 23

```
var mySender = new RTCRtpSender(myTrack);
var myPreviousReport = null;

// ... wait a bit
setTimeout(function () {
        mySender.getStats().then(function (report) {
        processStats(report);
        myPreviousReport = report;
    });
}, aBit);

function processStats(currentReport) {
    if (myPreviousReport === null) return;
    // currentReport + myPreviousReport are an RTCStatsReport interfac
    // compare the elements from the current report with the baseline
    for (var i in currentReport) {
        var now = currentReport[i];
        if (now.type != "outbound-rtp")
            continue;
        // get the corresponding stats from the previous report
        base = myPreviousReport[now.id];
        // base + now will be of RTCRtpStreamStats dictionary type
        if (base) {
            remoteNow = currentReport[now.associateStatsId];
            remoteBase = myPreviousReport[base.associateStatsId];
            var packetsSent = now.packetsSent - base.packetsSent;
            var packetsReceived = remoteNow.packetsReceived - remoteBa
            // if fractionLost is > 0.3, we have probably found the cu
            var fractionLost = (packetsSent - packetsReceived) / packe
        }
    }
}
```

14. Identity

> NOTE
> This section of the ORTC API specification depends on the WebRTC 1.0 Identity API, and needs to be synchronized once it is updated.

14.1 Overview

An **`RTCIdentity`** instance enables authentication of a DTLS transport using a web-based identity provider (IdP). The idea is that the initiator acts as the Authenticating Party (AP) and obtains an identity assertion from the IdP which is then conveyed in signaling. The responder acts as the Relying Party (RP) and verifies the assertion.

The interaction with the IdP is designed to decouple the browser from any particular identity provider, so that the browser need only know how to load the IdP's Javascript (which is deterministic from the IdP's identity), and the generic protocol for requesting and verifying assertions. The IdP provides whatever logic is necessary to bridge the generic protocol to the IdP's specific requirements. Thus, a single browser can support any number of identity protocols, including being forward compatible with IdPs which did not exist at the time the Identity Provider API was implemented. The generic protocol details are described in [RTCWEB-SECURITY-ARCH]. This section specifies the procedures required to instantiate the IdP proxy, request identity assertions, and consume the results.

14.2 Operation

A **`RTCIdentity`** instance is constructed from an **`RTCDtlsTransport`** object.

14.3 Identity Provider Selection

In order to communicate with the IdP, the browser instantiates an isolated interpreted context, effectively an invisible IFRAME. The initial contents of the context are loaded from a URI derived from the IdP's domain name, as described in [RTCWEB-SECURITY-ARCH].

For purposes of generating assertions, the IdP shall be chosen as follows:

1. If the `getIdentityAssertion()` method has been called, the IdP provided shall be used.

2. If the `getIdentityAssertion()` method has not been called, then the browser can use an IdP configured into the browser.

In order to verify assertions, the IdP domain name and protocol are taken from the `domain` and `protocol` fields of the identity assertion.

14.4 Instantiating an IdP Proxy

The browser creates an IdP proxy by loading an isolated, invisible IFRAME with HTML content from the IdP URI. The URI for the IdP is a well-known URI formed from the “domain” and “protocol” fields, as specified in [RTCWEB-SECURITY-ARCH].

When an IdP proxy is required, the browser performs the following steps:

1. An invisible, sandboxed IFRAME is created within the browser context. The IFRAME `sandbox` attribute is set to "allow-forms allow-scripts allow-same-origin" to limit the capabilities available to the IdP. The browser MUST prevent the IdP proxy from navigating the browsing context to a different location. The browser MUST prevent the IdP proxy from interacting with the user (this includes, in particular, popup windows and user dialogs).
2. Once the IdP proxy is created, the browser creates a `MessageChannel` [webmessaging] within the context of the IdP proxy and assigns one port from the channel to a variable named *rtcwebIdentityPort* on the *window*. This message channel forms the basis of communication between the browser and the IdP proxy. Since it is an essential security property of the web sandbox that a page is unable to insert objects into content from another origin, this ensures that the IdP proxy can trust that messages originating from *window.rtcwebIdentityPort* are from `RTCIdentity` and not some other page. This protection ensures that pages from other origins are unable to instantiate IdP proxies and obtain identity assertions.
3. The IdP proxy completes loading and informs the `RTCIdentity` object that it is ready by sending a "READY" message to the message channel port [RTCWEB-SECURITY-ARCH]. Once this message is received by the `RTCIdentity` object, the IdP is considered ready to receive requests to generate or verify identity assertions.

[TODO: This is not sufficient unless we expect the IdP to protect this information. Otherwise, the identity information can be copied from a session with "good" properties to any other session with the same fingerprint information. Since we want to reuse credentials, that would be bad.] The identity mechanism MUST provide an indication to the remote side of whether it requires the stream

contents to be protected. Implementations MUST have an user interface that indicates the different cases and identity for these.

14.5 Requesting Identity Assertions

The identity assertion request process involves the following steps:

1. The `RTCIdentity` instantiates an IdP proxy as described in Identity Provider Selection section and waits for the IdP to signal that it is ready.
2. The IdP sends a "SIGN" message to the IdP proxy. This message includes the material the `RTCIdentity` object desires to be bound to the user's identity.
3. If the user has been authenticated by the IdP, and the IdP is willing to generate an identity assertion, the IdP generates an identity assertion. This step depends entirely on the IdP. The methods by which an IdP authenticates users or generates assertions is not specified, though this could involve interacting with the IdP server or other servers.
4. The IdP proxy sends a response containing the identity assertion to the `RTCIdentity` object over the message channel.
5. The `RTCIdentity` object MAY store the identity assertion.

The format and contents of the messages that are exchanged are described in detail in [RTCWEB-SECURITY-ARCH].

The IdP proxy can return an "ERROR" response. If an error is encountered, the `getIdentityAssertion` Promise MUST be rejected.

The browser SHOULD limit the time that it will allow for this process. This includes both the loading of the IdP proxy and the identity assertion generation. Failure to do so potentially causes the corresponding operation to take an indefinite amount of time. This timer can be cancelled when the IdP produces a response. The timer running to completion can be treated as equivalent to an error from the IdP.

NOTE: Where RTP and RTCP are not multiplexed, distinct `RTCRtpIceTransport`, `RTCRtpDtlsTransport` and `RTCIdentity` objects can be constructed for RTP and RTCP. However, while it is possible for `getIdentityAssertion()` to be called with different values of *provider*, *protocol* and *username* for the RTP and RTCP `RTCIdentity` objects, application developers desiring backward compatibility with WebRTC 1.0 are strongly discouraged from doing so, since this is likely to result in an error.

14.5.1 User Login Procedure

An IdP could respond to a request to generate an identity assertion with a "LOGINNEEDED" error. This indicates that the site does not have the necessary information available to it (such as cookies) to authorize the creation of an identity assertion.

The "LOGINNEEDED" response includes a URL for a page where the authorization process can be completed. This URL is exposed to the application through the `loginUrl` attribute of the `RTCIdentityError` object. This URL might be to a page where a user is able to enter their (IdP) username and password, or otherwise provide any information the IdP needs to authorize a assertion request.

An application can load the login URL in an IFRAME or popup; the resulting page then provides the user with an opportunity to provide information necessary to complete the authorization process.

Once the authorization process is complete, the page loaded in the IFRAME or popup sends a message using *postMessage* [webmessaging] to the page that loaded it (through the *window.opener* attribute for popups, or through *window.parent* for pages loaded in an IFRAME). The message MUST be the *DOMString* "LOGINDONE". This message informs the application that another attempt at generating an identity assertion is likely to be successful.

14.6 Verifying Identity Assertions

Identity assertion validation happens when `setIdentityAssertion()` is invoked. The process runs asynchronously.

The identity assertion validation process involves the following steps:

1. The `RTCIdentity` instantiates an IdP proxy as described in Identity Provider Selection section and waits for the IdP to signal that it is ready.
2. The IdP sends a "VERIFY" message to the IdP proxy. This message includes the assertion which is to be verified.
3. The IdP proxy verifies the identity assertion (depending on the authentication protocol this could involve interacting with the IDP server).
4. Once the assertion is verified, the IdP proxy sends a response containing the verified assertion results to the `RTCIdentity` object over the message channel.

5. The `RTCIdentity` object validates that the fingerprint provided by the IdP in the validation response matches the certificate fingerprint that is, or will be, used for communications. This is done by waiting for the DTLS connection to be established and checking that the certificate fingerprint on the connection matches the one provided by the IdP.
6. The `RTCIdentity` validates that the domain portion of the identity matches the domain of the IdP as described in [RTCWEB-SECURITY-ARCH].
7. The `RTCIdentity` stores the assertion in the `peerIdentity`, and returns an `RTCIdentityAssertion` object when the Promise from `setIdentityAssertion()` is fulfilled. The assertion information to be displayed MUST contain the domain name of the IdP as provided in the assertion.
8. The browser MAY display identity information to a user in browser UI. Any user identity information that is displayed in this fashion MUST use a mechanism that cannot be spoofed by content.

The IdP might fail to validate the identity assertion by providing an "ERROR" response to the validation request. Validation can also fail due to the additional checks performed by the browser. In both cases, the process terminates and no identity information is exposed to the application or the user.

The browser MUST cause the Promise of `setIdentityAssertion()` to be rejected if validation of an identity assertion fails for any reason.

The browser SHOULD limit the time that it will allow for this process. This includes both the loading of the IdP proxy and the identity assertion validation. Failure to do so potentially causes the corresponding operation to take an indefinite amount of time. This timer can be cancelled when the IdP produces a response. The timer running to completion can be treated as equivalent to an error from the IdP.

The format and contents of the messages that are exchanged are described in detail in [RTCWEB-SECURITY-ARCH].

NOTE: Where RTP and RTCP are not multiplexed, it is possible that the assertions for both the RTP and RTCP will be validated, but that the identities will not be equivalent. For applications requiring backward compatibility with WebRTC 1.0, this MUST be considered an error. However, if backward compatibility with WebRTC 1.0 is not required the application MAY consider an alternative, such as ignoring the RTCP identity assertion.

14.7 RTCIdentity Interface

The Identity API is described below.

WebIDL

```
[Constructor(RTCDtlsTransport transport)]
interface RTCIdentity {
    readonly        attribute RTCIdentityAssertion? peerIdentity;
    readonly        attribute RTCDtlsTransport      transport;
    Promise<DOMString>            getIdentityAssertion (DOMString
provider, optional DOMString protocol = "default", optional DOMString
username);
    Promise<RTCIdentityAssertion> setIdentityAssertion (DOMString
assertion);
};
```

14.7.1 Attributes

`peerIdentity` of type *RTCIdentityAssertion*, readonly , nullable

peerIdentity contains the peer identity assertion information if an identity assertion was provided and verified. Once this value is set to a non-*null* value, it cannot change.

`transport` of type *RTCDtlsTransport*, readonly

The `RTCDtlsTransport` to be authenticated.

14.7.2 Methods

`getIdentityAssertion`

Sets the identity provider to be used for a given **`RTCIdentity`** object, and initiates the process of obtaining an identity assertion.

When ***getIdentityAssertion()*** is invoked, the user agent MUST run the following steps:

1. If `transport.state` is `closed`, throw an `InvalidStateError` exception and abort these steps.
2. Set the current identity provider values to the triplet (`provider`, `protocol`, `username`).
3. If any identity provider value has changed, discard any stored identity assertion.

4. Request an identity assertion from the IdP.

5. If the IdP proxy provides an assertion over the message channel, the Promise is fulfilled, and the assertion is returned (equivalent to `onidentityresult` in the WebRTC 1.0 API). If the IdP proxy returns an "ERROR" response, the Promise is rejected, and an `RTCIdentityError` object is returned, (equivalent to `onidpassertionerror` in the WebRTC 1.0 API).

Parameter	Type	Nullable	Optional	Description
provider	`DOMString`	✘	✘	
protocol	`DOMString = "default"`	✘	✔	
username	`DOMString`	✘	✔	

Return type: `Promise<DOMString>`

`setIdentityAssertion`

Validates the identity assertion. If the Promise is fulfilled, an `RTCIdentityAssertion` is returned. If the Promise is rejected, an `RTCIdentityError` object is returned, (equivalent to `onidpvalidationerror` in the WebRTC 1.0 API).

Parameter	Type	Nullable	Optional	Description
assertion	`DOMString`	✘	✘	

Return type: `Promise<RTCIdentityAssertion>`

14.8 dictionary RTCIdentityError

WebIDL

```
dictionary RTCIdentityError {
             DOMString  idp;
             DOMString? loginUrl;
             DOMString  protocol;
};
```

14.8.1 Dictionary `RTCIdentityError` Members

idp of type DOMString
The domain name of the identity provider that is providing the error response.

loginUrl of type DOMString, nullable
An IdP that is unable to generate an identity assertion due to a lack of sufficient user authentication information can provide a URL to a page where the user can complete authentication. If the IdP provides this URL, this attribute includes the value provided by the IdP.

protocol of type DOMString
The IdP protocol that is in use.

14.9 dictionary RTCIdentityAssertion

WebIDL

```
dictionary RTCIdentityAssertion {
             DOMString idp;
             DOMString name;
};
```

14.9.1 Dictionary RTCIdentityAssertion Members

idp of type DOMString
A domain name representing the identity provider.

name of type DOMString
A representation of the verified peer identity conforming to [RFC5322]. This identity will have been verified via the procedures described in [RTCWEB-SECURITY-ARCH].

14.10 Example

The identity system is designed so that applications need not take any special action in order for users to generate and verify identity assertions; if a user has configured an IdP into their browser, then the browser will automatically request/generate assertions and the other side will automatically verify them and display the results. However, applications may wish to exercise tighter control over the identity system as shown by the following examples.

This example shows how to configure the identity provider and protocol, and consume identity assertions.

EXAMPLE 24

```
// Set ICE gather options and construct the RTCIceGatherer object, ass
// we are using RTP/RTCP mux and A/V mux so that only one RTCIceTransp
// Include some helper functions
import "helper.js";
var gatherOptions = new RTCIceGatherOptions();
gatherOptions.gatherPolicy = RTCIceGatherPolicy.all;
gatherOptions.iceservers = [ { urls: "stun:stun1.example.net" } , { ur
var iceGatherer = new RTCIceGatherer(gatherOptions);
iceGatherer.onlocalcandidate = function (event) {mySendLocalCandidate(
var ice = new RTCIceTransport(iceGatherer);
// Create the RTCDtlsTransport object.
var dtls = new RTCDtlsTransport(ice);
var identity = new RTCIdentity(dtls);
identity.getIdentityAssertion("example.com", "default", "alice@example
 ,function (e) {
    trace('Could not obtain an Identity Assertion. idp: ' + e.idp + '
});

function signalAssertion(assertion){
  mySignalInitiate(
    { "myAssertion": assertion,
      "ice": iceGatherer.getLocalParameters(),
      "dtls": dtls.getLocalParameters()
    }, function (response) {
          ice.start(iceGatherer, response.ice, RTCIceRole.controlling)
          // Need to call dtls.start() before setIdentityAssertion so
          dtls.start(response.dtls);
          identity.setIdentityAssertion(response.myAssertion).then(fun
               trace('Peer identity assertion validated. idp: ' + peer
              }, function (e) {
               trace('Could not validate peer assertion. idp: ' + e.id
          });
    });
}
```

15. Event summary

This section is non-normative.

The following events fire on **RTCDtlsTransport** objects:

Event name	Interface	Fired when...
error	Event	The **RTCDtlsTransport** object has received a DTLS Alert.
dtlsstatechange	**RTCDtlsTransportStateChangedEvent**	The *RTCDtlsTransportState* changed.

The following events fire on **RTCIceTransport** objects:

Event name	Interface	Fired when...
icestatechange	**RTCIceTransportStateChangedEvent**	The *RTCIceTransportState* changed.
icecandidatepairchange	**RTCIceCandidatePairChangedEvent**	The nominated *RTCIceCandidatePair* changed.

The following events fire on **RTCIceGatherer** objects:

Event name	Interface	Fired when...
error	Event	The **RTCIceGatherer** object has experienced an ICE gathering failure (such as an authentication failure with TURN credentials).
icegatherstatechange	**RTCIceGathererStateChangedEvent**	The *RTCIceGathererState* changed.
icecandidate	**RTCIceGatherer**	A new **RTCIceGatherCandidate**

		is made available to the script.

The following events fire on **RTCRtpSender** objects:

Event name	Interface	Fired when...
error	Event	An error has been detected within the **RTCRtpSender** object. This is not used for programmatic exceptions.
ssrcconflict	**RTCSsrcConflictEvent**	An SSRC conflict has been detected.

The following event fires on **RTCRtpReceiver** objects:

Event name	Interface	Fired when...
error	Event	An error has been detected within the **RTCRtpReceiver** object, such as an issue with **RTCRtpParameters** that could not be detected until media arrival. This is not used for programmatic exceptions.

The following events fire on **RTCRtpListener** objects:

Event name	Interface	Fired when...
		The **RTCRtpListener** object has received an RTP packet that it

unhandledrtp	RTCRtpUnhandledEvent	cannot deliver to an RTCRtpReceiver object.

The following events fire on RTCDTMFSender objects:

Event name	Interface	Fired when...
tonechange	Event	The RTCDTMFSender object has either just begun playout of a tone (returned as the tone attribute) or just ended playout of a tone (returned as an empty value in the tone attribute).

The following events fire on RTCDataChannel objects:

Event name	Interface	Fired when...
open	Event	The RTCDataChannel object's underlying data transport has been established (or re-established).
MessageEvent	Event	A message was successfully received. TODO: Ref where MessageEvent is defined?
error	Event	TODO.
close	Event	The RTCDataChannel object's underlying data transport has been closed.

The following events fire on RTCSctpTransport objects:

Event name	Interface	Fired when...
		A new

datachannel	RTCDataChannelEvent	RTCDataChannel is dispatched to the script in response to the other peer creating a channel.

16. WebRTC 1.0 Compatibility

This section is non-normative.

It is a goal of the ORTC API to provide the functionality of the WebRTC 1.0 API [WEBRTC10], as well as to enable the WebRTC 1.0 API to be implemented on top of the ORTC API, utilizing a Javascript "shim" library. This section discusses WebRTC 1.0 compatibility issues that have been encountered by ORTC API implementers.

16.1 BUNDLE

Via the use of [BUNDLE] it is possible for WebRTC 1.0 implementations to multiplex audio and video on the same RTP session. Within ORTC API, equivalent behavior can be obtained by constructing multiple RTCRtpReceiver and RTCRtpSender objects from the same RTCDtlsTransport object. As noted in [RTP-USAGE] Section 4.4, support for audio/video multiplexing is required, as described in [RTP-MULTI-STREAM].

16.2 Voice Activity Detection

[WEBRTC10] Section 4.2.4 defines the RTCOfferOptions dictionary, which includes the *voiceActivityDetection* attribute, which determines whether Voice Activity Detection (VAD) is enabled within the Offer produced by createOffer(). The effect of setting *voiceActivityDetection* to *true* is to include the Comfort Noice (CN) codec defined in [RFC3389] within the Offer.

Within ORTC API, equivalent behavior can be obtained by configuring the Comfort Noise (CN) codec for use within RTCRtpParameters, or configuring a codec with built-in support for

Comfort Noise (such as Opus) to enable comfort noise. As noted in [RTCWEB-AUDIO] Section 3, support for CN is required.

17. Examples

This section is non-normative.

17.1 Simple Peer-to-peer Example

This example code provides a basic audio and video session between two browsers.

EXAMPLE 25

17.2 myCapsToSendParams Example

EXAMPLE 26

```
RTCRtpParameters function myCapsToSendParams (RTCRtpCapabilities sendC
// Function returning the sender RTCRtpParameters, based on the local
// The goal is to enable a single stream audio and video call with min
//
// Steps to be followed:
// 1. Determine the RTP features that the receiver and sender have in
// 2. Determine the codecs that the sender and receiver have in common
// 3. Within each common codec, determine the common formats, header e
// 4. Determine the payloadType to be used, based on the receiver pref
// 5. Set RTCRtcpParameters such as mux to their default values.
// 6. Return RTCRtpParameters enablig the jointly supported features a
}

RTCRtpParameters function myCapsToRecvParams (RTCRtpCapabilities recvC
// Function returning the receiver RTCRtpParameters, based on the loca
return myCapsToSendParams(remoteSendCaps, recvCaps);
}
```

A. Acknowledgements

The editor wishes to thank Erik Lagerway for his support. Substantial text in this specification was provided by many people including Peter Thatcher, Martin Thomson, Iñaki Baz Castillo, Jose Luis Millan, Christoph Dorn, Roman Shpount, Emil Ivov, Shijun Sun and Jason Ausborn. Special thanks to Peter Thatcher for his design contributions relating to many of the objects in the current specification, and to Philipp Hancke for his detailed review.

B. Change Log

This section will be removed before publication.

B.1 Changes since 7 May 2015

1. Addressed Philipp Hancke's review comments, as noted in: Issue 198
2. Added the "failed" state to **`RTCIceTransportState`**, as noted in: Issue 199
3. Added text relating to handling of incoming media packets prior to remote fingerprint verification, as noted in: Issue 200
4. Added a *complete* attribute to the **`RTCIceCandidateComplete`** dictionary, as noted in: Issue 207
5. Updated the description of `RTCIceGatherer.close()` and the "closed" state, as noted in: Issue 208
6. Updated Statistics API error handling to reflect proposed changes to the WebRTC 1.0 API, as noted in: Issue 214
7. Updated Section 10 (RTCDtmfSender) to reflect changes in the WebRTC 1.0 API, as noted in: Issue 215
8. Clarified state transitions due to consent failure, as noted in: Issue 216
9. Added a reference to [FEC], as noted in: Issue 217

B.2 Changes since 25 March 2015

1. `sender.setTrack()` updated to return a Promise, as noted in: Issue 148

2. Added `RTCIceGatherer` as an optional argument to the `RTCIceTransport` constructor, as noted in: Issue 174
3. Clarified handling of contradictory RTP/RTCP multiplexing settings, as noted in: Issue 185
4. Clarified error handling relating to **`RTCIceTransport`**, **`RTCDtlsTransport`** and **`RTCIceGatherer`** objects in the "closed" state, as noted in: Issue 186
5. Added *component* attribute and `createAssociatedGatherer()` method to the **`RTCIceGatherer`** object, as noted in: Issue 188
6. Added `close()` method to the **`RTCIceGatherer`** object as noted in: Issue 189
7. Clarified behavior of TCP candidate types, as noted in: Issue 190
8. Clarified behavior of `iceGatherer.onlocalcandidate`, as noted in: Issue 191
9. Updated terminology in Section 1.1 as noted in: Issue 193
10. Updated **`RTCDtlsTransportState`** definitions, as noted in: Issue 194
11. Updated **`RTCIceTransportState`** definitions, as noted in: Issue 197

B.3 Changes since 22 January 2015

1. Updated Section 8.3 on RTP matching rules, as noted in: Issue 48
2. Further updates to the Statistics API, reflecting: Issue 85
3. Added support for *maxptime*, as noted in: Issue 160
4. Revised the text relating to `RTCDtlsTransport.start()`, as noted in: Issue 168
5. Clarified pre-requisites for `insertDTMF()`, based on: Issue 178
6. Added Section 13.4 and updated Section 9.5.1 to clarify aspects of RTCP sending and receiving, based on: Issue 180
7. Fixed miscellaneous typos, as noted in: Issue 183
8. Added informative reference to [RFC3264] Section 5.1, as noted in: Issue 184

B.4 Changes since 14 October 2014

1. Update to the Statistics API, reflecting: Issue 85
2. Update on 'automatic' use of scalable video coding, as noted in: Issue 156

3. Update to the H.264 parameters, as noted in: Issue 158
4. Update to the 'Big Picture', as noted in: Issue 159
5. Changed 'RTCIceTransportEvent' to 'RTCIceGathererEvent' as noted in: Issue 161
6. Update to `RTCRtpUnhandledEvent` as noted in: Issue 163
7. Added support for `RTCIceGatherer.state` as noted in: Issue 164
8. Revised the text relating to `RTCIceTransport.start()` as noted in: Issue 166
9. Added text relating to DTLS interoperability with WebRTC 1.0, as noted in: Issue 167
10. Added a reference to the ICE consent specification, as noted in: Issue 171

B.5 Changes since 20 August 2014

1. Address questions about `RTCDtlsTransport.start()`, as noted in: Issue 146
2. Address questions about `RTCRtpCodecCapability.preferredPayloadType`, as noted in: Issue 147
3. Address questions about `RTCRtpSender.setTrack()` error handling, as noted in: Issue 148
4. Address 'automatic' use of scalable video coding (in `RTCRtpReceiver.receive()`) as noted in: Issue 149
5. Renamed RTCIceListener to **`RTCIceGatherer`** as noted in: Issue 150
6. Added text on multiplexing of STUN, TURN, DTLS and RTP/RTCP, as noted in: Issue 151
7. Address issue with queueing of candidate events within the **`RTCIceGatherer`**, as noted in: Issue 152
8. Clarify behavior of `RTCRtpReceiver.getCapabilities(kind)`, as noted in: Issue 153

B.6 Changes since 16 July 2014

1. Clarification of the ICE restart issue, as noted in : Issue 93
2. Clarified onerror usage in sender and receiver objects, as noted in: Issue 95
3. Clarified SST-MS capability issue noted in: Issue 108
4. Clarification of `send()` and `receive()` usage as noted in: Issue 119

5. Changed ICE state diagram as noted in: Issue 122
6. Removed getParameters methods and changed send() method as noted in: Issue 136
7. Changed definition of framerateScale and resolutionScale as noted in: Issue 137
8. Substituted 'muxId' for the 'receiverId' as noted in: Issue 138 and Issue 140
9. Clarified the setting of *track.kind* as described in: Issue 141
10. Added SSRC conflict event to the **`RTCRtpSender`**, as described in: Issue 143
11. Addressed the "end of candidates" issues noted in: Issue 142 and Issue 144

B.7 Changes since 16 June 2014

1. Added section on WebRTC 1.0 compatibility issues, responding to Issue 66
2. Added Identity support, as described in Issue 78
3. Reworked `getStats()` method, as described in Issue 85
4. Removed ICE restart method described in Issue 93
5. Addressed CNAME and synchronization context issues described in Issue 94
6. Fixed WebIDL issues noted in Issue 97
7. Addressed NITs described in Issue 99
8. DTLS transport issues fixed as described in Issue 100
9. ICE transport issues fixed as described in Issue 101
10. ICE transport controller fixes made as described in Issue 102
11. Sender and Receiver object fixes made as described in Issue 103
12. Fixed **`RTCRtpEncodingParameters`** default issues described in Issue 104
13. Fixed 'Big Picture' issues descibed in Issue 105
14. Fixed **`RTCRtpParameters`** default issues described in Issue 106
15. Added a multi-stream capability, as noted in Issue 108
16. Removed quality scalability capabilities and parameters, as described in Issue 109
17. Added scalability examples as requested in Issue 110
18. Addressed WebRTC 1.0 Data Channel compatibility issue described in Issue 111

19. Removed header extensions from **`RTCRtpCodecParameters`** as described in Issue 113
20. Addressed RTP/RTCP non-mux issues with IdP as described in Issue 114
21. Added getParameter methods to **`RTCRtpSender`** and **`RTCRtpReceiver`** objects, as described in Issue 116
22. Added layering diagrams as requested in Issue 117
23. Added a typedef for *payloadtype*, as described in Issue 118
24. Moved `onerror` from the **`RTCIceTransport`** object to the `RTCIceListener` object as described in Issue 121
25. Added explanation of Voice Activity Detection (VAD), responding to Issue 129
26. Clarified the meaning of maxTemporalLayers and maxSpatialLayers, as noted in Issue 130
27. Added [RFC6051] to the list of header extensions and removed RFC 5450, as noted in Issue 131
28. Addressed ICE terminology issues, as described in Issue 132
29. Separated references into Normative and Informative, as noted in Issue 133

B.8 Changes since 14 May 2014

1. Added support for non-multiplexed RTP/RTCP and ICE freezing, as described in Issue 57
2. Added support for getRemoteCertificates(), as described in Issue 67
3. Removed filterParameters() and createParameters() methods, as described in Issue 80
4. Partially addressed capabilities issues, as described in Issue 84
5. Addressed WebIDL type issues described in Issue 88
6. Addressed Overview section issues described in Issue 91
7. Addressed readonly attribute issues described in Issue 92
8. Added ICE restart method to address the issue described in Issue 93
9. Added onerror eventhandler to sender and receiver objects as described in Issue 95

B.9 Changes since 29 April 2014

1. ICE restart explanation added, as described in Issue 59

2. Fixes for error handling, as described in Issue 75
3. Fixes for miscellaneous NITs, as described in Issue 76
4. Enable retrieval of the SSRC to be used by RTCP, as described in Issue 77
5. Support for retrieval of audio and video capabilities, as described in Issue 81
6. getStats interface updated, as described in Issue 82
7. Partially addressed SVC issues described in Issue 83
8. Partially addressed statistics update issues described in Issue 85

B.10 Changes since 12 April 2014

1. Fixes for error handling, as described in Issue 26
2. Support for contributing sources removed (re-classified as a 1.2 feature), as described in Issue 27
3. Cleanup of DataChannel construction, as described in Issue 60
4. Separate proposal on simulcast/layering, as described in Issue 61
5. Separate proposal on quality, as described in Issue 62
6. Fix for TCP candidate type, as described in Issue 63
7. Fix to the fingerprint attribute, as described in Issue 64
8. Fix to RTCRtpFeatures, as described in Issue 65
9. Support for retrieval of remote certificates, as described in Issue 67
10. Support for ICE error handling, described in Issue 68
11. Support for Data Channel send rate control, as described in Issue 69
12. Support for capabilities and settings, as described in Issue 70
13. Removal of duplicate RTCIceListener functionality, as described in Issue 71
14. ICE gathering state added, as described in Issue 72
15. Removed ICE role from the ICE transport constructor, as described in Issue 73

B.11 Changes since 13 February 2014

1. Support for contributing source information added, as described in Issue 27
2. Support for control of quality, resolution, framerate and layering added, as described in Issue 31
3. RTCRtpListener object added and figure in Section 1 updated, as described in Issue 32
4. More complete support for RTP and Codec Parameters added, as described in Issue 33
5. Data Channel transport problem fixed, as described in Issue 34
6. Various NITs fixed, as described in Issue 37
7. **`RTCDtlsTransport`** operation and interface definition updates, as described in: Issue 38
8. Default values of some dictionary attributes added, to partially address the issue described in: Issue 39
9. Support for ICE TCP added, as described in Issue 41
10. Fixed issue with sequences as attributes, as described in Issue 43
11. Fix for issues with onlocalcandidate, as described in Issue 44
12. Initial stab at a Stats API, as requested in Issue 46
13. Added support for ICE gather policy, as described in Issue 47

B.12 Changes since 07 November 2013

1. RTCTrack split into **`RTCRtpSender`** and **`RTCRtpReceiver`** objects, as proposed on 06 January 2014.
2. RTCConnection split into **`RTCIceTransport`** and **`RTCDtlsTransport`** objects, as proposed on 09 January 2014.
3. **`RTCSctpTransport`** object added, as described in Issue 25
4. RTCRtpHeaderExtensionParameters added, as described in Issue 28
5. RTCIceListener added, in order to support parallel forking, as described in Issue 29
6. DTMF support added, as described in Issue 30

C. References

C.1 Normative references

[BUNDLE]

C. Holmberg; H. Alvestrand; C. Jennings. IETF. *Negotiating Media Multiplexing Using the Session Description Protocol (SDP)*. 16 June 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-mmusic-sdp-bundle-negotiation

[CONSENT]

M. Perumal; D. Wing; T. Reddy; M. Thomson. IETF. *STUN Usage for Consent Freshness*. 8 June 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-stun-consent-freshness

[DATA]

R. Jesup; S. Loreto; M. Tuexen. IETF. *WebRTC Data Channels*. 04 January 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel

[DATA-PROT]

R. Jesup; S. Loreto; M. Tuexen. IETF. *WebRTC Data Channel Establishment Protocol*. 04 January 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-protocol

[GETUSERMEDIA]

Daniel Burnett; Adam Bergkvist; Cullen Jennings; Anant Narayanan. W3C. *Media Capture and Streams*. 14 April 2015. W3C Last Call Working Draft. URL: http://www.w3.org/TR/mediacapture-streams/

[HTML5]

Ian Hickson; Robin Berjon; Steve Faulkner; Travis Leithead; Erika Doyle Navara; Edward O'Connor; Silvia Pfeiffer. W3C. *HTML5*. 28 October 2014. W3C Recommendation. URL: http://www.w3.org/TR/html5/

[IANA-RTP-10]

IANA. *RTP Compact Header Extensions*. URL: http://www.iana.org/assignments/rtp-parameters/rtp-parameters.xhtml#rtp-parameters-10

[IANA-RTP-2]

IANA. *RTP Payload Format media types*. URL: http://www.iana.org/assignments/rtp-parameters/rtp-parameters.xhtml#rtp-parameters-2

[IANA-SDP-14]

IANA. *'rtcp-fb' Attribute Values*. URL: http://www.iana.org/assignments/sdp-parameters/sdp-parameters.xhtml#sdp-parameters-14

[IANA-SDP-15]

IANA. *'ack' and 'nack' Attribute Values*. URL: http://www.iana.org/assignments/sdp-parameters/sdp-parameters.xhtml#sdp-parameters-15

[IANA-SDP-19]

IANA. *Codec Control Messages*. URL: http://www.iana.org/assignments/sdp-parameters/sdp-parameters.xhtml#sdp-parameters-19

[OPUS-RTP]

J. Spittka; K. Vos; JM. Valin. IETF. *RTP Payload Format for Opus Speech and Audio Codec*. 14 April 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-payload-rtp-opus

[RFC3389]

R. Zopf. IETF. *Real-time Transport Protocol (RTP) Payload for Comfort Noise (CN)*. September 2002. Proposed Standard. URL: https://tools.ietf.org/html/rfc3389

[RFC3550]

H. Schulzrinne; S. Casner; R. Frederick; V. Jacobson. IETF. *RTP: A Transport Protocol for Real-Time Applications*. July 2003. Internet Standard. URL: https://tools.ietf.org/html/rfc3550

[RFC3611]

T. Friedman, Ed.; R. Caceres, Ed.; A. Clark, Ed.. IETF. *RTP Control Protocol Extended Reports (RTCP XR)*. November 2003. Proposed Standard. URL: https://tools.ietf.org/html/rfc3611

[RFC3711]

M. Baugher; D. McGrew; M. Naslund; E. Carrara; K. Norrman. IETF. *The Secure Real-time Transport Protocol (SRTP)*. March 2004. Proposed Standard. URL: https://tools.ietf.org/html/rfc3711

[RFC4572]

J. Lennox. IETF. *Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP)*. July 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4572

[RFC4585]

J. Ott; S. Wenger; N. Sato; C. Burmeister; J. Rey. IETF. *Extended RTP Profile for RTCP-Based Feedback (RTP/AVPF)*. July 2006. RFC. URL: https://tools.ietf.org/html/rfc4585

[RFC4588]

J. Rey; D. Leon; A. Miyazaki; V. Varsa; R. Hakenberg. IETF. *RTP Retransmission Payload Format*. July 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4588

[RFC4733]

H. Schulzrinne; T. Taylor. IETF. *RTP Payload for DTMF Digits, Telephony Tones, and Telephony Signals*. December 2006. RFC. URL: https://tools.ietf.org/html/rfc4733

[RFC4961]

D. Wing. IETF. *Symmetric RTP/RTC Control Protocol (RTCP)*. July 2007. RFC. URL: https://tools.ietf.org/html/rfc4961

[RFC5245]

J. Rosenberg. IETF. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. April 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5245

[RFC5285]

D. Singer; H. Desineni. IETF. *A General Mechanism for RTP Header Extensions*. July 2008. RFC. URL: https://tools.ietf.org/html/rfc5285

[RFC5389]

J. Rosenberg; R. Mahy; P. Matthews; D. Wing. IETF. *Session Traversal Utilities for NAT (STUN)*. October 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5389

[RFC5506]

I. Johansson; M. Westerlund. IETF. *Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences*. April 2009. RFC. URL: https://tools.ietf.org/html/rfc5506

[RFC5761]

C. Perkins; M. Westerlund. IETF. *Multiplexing RTP Data and Control Packets on a Single Port*. April 2010. RFC. URL: https://tools.ietf.org/html/rfc5761

[RFC5764]

D. McGrew; E. Rescorla. IETF. *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)*. May 2010. RFC. URL: https://tools.ietf.org/html/rfc5764

[RFC6051]

C. Perkins; T. Schierl. IETF. *Rapid Synchronisation of RTP Flows*. November 2010. RFC. URL: https://tools.ietf.org/html/rfc6051

[RFC6464]

J. Lennox, Ed.; E. Ivov; E. Marocco. IETF. *A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication*. December 2011. RFC. URL: https://tools.ietf.org/html/rfc6464

[RFC6544]

J. Rosenberg; A. Keranen; B. B. Lowekamp; A. B. Roach. IETF. *TCP Candidates with Interactive Connectivity Establishment (ICE)*. March 2012. RFC. URL:

https://tools.ietf.org/html/rfc6544

[RFC6904]

J. Lennox. IETF. *Encryption of Header Extensions in the SRTP*. April 2013. RFC. URL: https://tools.ietf.org/html/rfc6904

[RFC7022]

A. Begen; C. Perkins; D. Wing; E. Rescorla. IETF. *Guidelines for Choosing RTP Control Protocol (RTCP) Canonical Names (CNAMEs)*. September 2013. RFC. URL: https://tools.ietf.org/html/rfc7022

[RFC7064]

S. Nandakumar; G. Salgueiro; P. Jones; M. Petit-Huguenin. IETF. *URI Scheme for Session Traversal Utilities for NAT (STUN) Protocol*. November 2013. RFC. URL: https://tools.ietf.org/html/rfc7064

[RFC7065]

M. Petit-Huguenin; S. Nandakumar; G. Salgueiro; P. Jones. IETF. *Traversal Using Relays around NAT (TURN) Uniform Resource Identifiers*. November 2013. RFC. URL: https://tools.ietf.org/html/rfc7065

[RTCWEB-AUDIO]

JM. Valin; C. Bran. IETF. *WebRTC Audio Codec and Processing Requirements*. 30 April 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-audio

[RTCWEB-SECURITY-ARCH]

E. Rescorla. IETF. *WebRTC Security Architecture*. 07 March 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch

[RTCWEB-VIDEO]

A.B. Roach. IETF. *WebRTC Video Processing and Codec Requirements*. 12 June 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-video

[RTP-MULTI-STREAM]

M. Westerlund; C. Perkins; J. Lennox. IETF. *Sending Multiple Types of Media in a Single RTP Session*. 09 March 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-avtcore-multi-media-rtp-session

[RTP-USAGE]

C. Perkins; M. Westerlund; J. Ott. IETF. *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP*. 12 June 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage

[WEBIDL]

Cameron McCormack; Boris Zbarsky. W3C. *WebIDL Level 1*. 8 March 2016. W3C Candidate Recommendation. URL: http://www.w3.org/TR/WebIDL-1/

[WEBRTC-STATS]

Harald Alvestrand; Varun Singh. W3C. *Identifiers for WebRTC's Statistics API*. 06 February 2015 (work in progress). URL: http://www.w3.org/TR/webrtc-stats/

[webmessaging]

Ian Hickson. W3C. *HTML5 Web Messaging*. 19 May 2015. W3C Recommendation. URL: http://www.w3.org/TR/webmessaging/

C.2 Informative references

[FEC]

J. Uberti. IETF. *WebRTC Forward Error Correction Requirements*. 05 March 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-rtcweb-fec

[GROUPING]

J. Lennox; K. Gross; S. Nandakumar; G. Salgueiro; B. Burman. IETF. *RTP Grouping Taxonomy*. 05 March 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-avtext-rtp-grouping-taxonomy

[HIGHRES-TIME]

Ilya Grigorik; James Simonsen; Jatinder Mann. W3C. *High Resolution Time Level 2*. 25 February 2016. W3C Working Draft. URL: http://www.w3.org/TR/hr-time-2/

[MUX-FIXES]

M. Petit-Huguenin; G. Salgueiro. IETF. *Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)*. 24 March 2015. Internet draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-avtcore-rfc5764-mux-fixes

[RFC2198]

C. Perkins; I. Kouvelas; O. Hodson; V. Hardman; M. Handley; J.C. Bolot; A. Vega-Garcia; S. Fosse-Parisis. IETF. *RTP Payload for Redundant Audio Data*. September 1997. Proposed Standard. URL: https://tools.ietf.org/html/rfc2198

[RFC3264]

J. Rosenberg; H. Schulzrinne. IETF. *An Offer/Answer Model with the Session Description Protocol*. July 2002. RFC. URL: https://tools.ietf.org/html/rfc3264

[RFC5322]

P. Resnick, Ed.. IETF. *Internet Message Format*. October 2008. Draft Standard. URL: https://tools.ietf.org/html/rfc5322

[RFC6184]

Y.-K.. Wang; R. Even; T. Kristensen; R. Jesup. IETF. *RTP Payload Format for H.264 Video*. May 2011. RFC. URL: https://tools.ietf.org/html/rfc6184

[RFC6190]

S. Wenger; Y.-K. Wang; T. Schierl; A. Eleftheriadis. IETF. *RTP Payload Format for Scalable Video Coding*. May 2011. RFC. URL: https://tools.ietf.org/html/rfc6190

[RFC6455]

I. Fette; A. Melnikov. IETF. *The WebSocket Protocol*. December 2011. RFC. URL: https://tools.ietf.org/html/rfc6455

[RFC6465]

E. Ivov; E. Marocco; J. Lennox. IETF. *A Real-time Protocol (RTP) Header Extension for Mixer-to-Client Audio Level Indication*. December 2011. RFC. URL: https://tools.ietf.org/html/rfc6465

[VP8-RTP]

P. Westin; H. Lundin; M. Glover; J. Uberti; F. Galligan. IETF. *RTP Payload Format for VP8 Video*. 05 June 2015. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-payload-vp8

[WEBRTC10]

Adam Bergkvist; Daniel Burnett; Cullen Jennings; Anant Narayanan; Bernard Aboba. W3C. *WebRTC 1.0: Real-time Communication Between Browsers*. 28 January 2016. W3C Working Draft. URL: http://www.w3.org/TR/webrtc/

[WEBSOCKETS-API]

Ian Hickson. W3C. *The WebSocket API*. 20 September 2012. W3C Candidate Recommendation. URL: http://www.w3.org/TR/websockets/

↑