WebRTC: Real-Time Communication in Browsers

W3C Recommendation 08 October 2024

▾ **More details about this document**

This version:
https://www.w3.org/TR/2024/REC-webrtc-20241008/

Latest published version:
https://www.w3.org/TR/webrtc/

Latest editor's draft:
https://w3c.github.io/webrtc-pc/

History:
https://www.w3.org/standards/history/webrtc/
Commit history

Test suite:
https://github.com/web-platform-tests/wpt/tree/master/webrtc/

Implementation report:
https://w3c.github.io/webrtc-interop-reports/webrtc-pc-report.html

Editors:
Cullen Jennings (Cisco)
Florent Castelli (Google)
Henrik Boström (Google)
Jan-Ivar Bruaroey (Mozilla)

Former editors:
Adam Bergkvist (Ericsson) - Until 01 June 2018
Daniel C. Burnett (Invited Expert) - Until 01 June 2018
Anant Narayanan (Mozilla) - Until 01 November 2012
Bernard Aboba (Microsoft Corporation) - Until 01 March 2017
Taylor Brandstetter (Google) - Until 01 June 2018

Feedback:
GitHub w3c/webrtc-pc (pull requests, new issue, open issues)
public-webrtc@w3.org with subject line `[webrtc] … message topic …` (archives)

Participate
Mailing list

See also **translations**.

Abstract

This document defines a set of ECMAScript APIs in WebIDL to allow media and generic application data to be sent to and received from another browser or device implementing the appropriate set of real-time protocols. This specification is being developed in conjunction with a protocol specification developed by the IETF RTCWEB group and an API specification to get access to local media devices.

Status of This Document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document includes Proposed Amendments and Candidate Amendments to the current W3C Recommendation dated January 26, 2021.

Its associated test suite has been used to build an implementation report of the API at the time of its initial publication as a Recommendation. That test suite has been updated to integrate most of the amendments, and an updated implementation report focused on the implementation status of these amendments has been used to select features with double implementation as proposed amendments.

This document was published by the Web Real-Time Communications Working Group as a Recommendation using the Recommendation track. It includes proposed amendments, introducing substantive changes and new features since the previous Recommendation.

W3C recommends the wide deployment of this specification as a standard for the Web.

A W3C Recommendation is a specification that, after extensive consensus-building, is endorsed by W3C and its Members, and has commitments from Working Group members to royalty-free licensing for implementations. Future updates to this Recommendation may incorporate new features.

Candidate additions are marked in the document.

Candidate corrections are marked in the document.

Proposed additions are marked in the document.

Proposed corrections are marked in the document.

The W3C Membership and other interested parties are invited to review the proposed additions and send comments through 08 December 2024. Advisory Committee Representatives should consult their WBS questionnaires.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 03 November 2023 W3C Process Document.

Table of Contents

§ 1. Introduction

This section is non-normative.

There are a number of facets to peer-to-peer communications and video-conferencing in HTML covered by this specification:

- Connecting to remote peers using NAT-traversal technologies such as ICE, STUN, and TURN.
- Sending the locally-produced tracks to remote peers and receiving tracks from remote peers.
- Sending arbitrary data directly to remote peers.

This document defines the APIs used for these features. This specification is being developed in conjunction with a protocol specification developed by the IETF RTCWEB group and an API specification to get access to local media devices [GETUSERMEDIA] developed by the WebRTC Working Group. An overview of the system can be found in [RFC8825] and [RFC8826].

§ 2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, and *SHOULD* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification defines conformance criteria that apply to a single product: the **user agent** that implements the interfaces that it contains.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

Implementations that use ECMAScript to implement the APIs defined in this specification *MUST* implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [WEBIDL], as this specification uses that specification and terminology.

§ 3. Terminology

The `EventHandler` interface, representing a callback used for event handlers, is defined in [HTML].

The concepts queue a task and networking task source are defined in [HTML].

The concept fire an event is defined in [DOM].

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

`Performance.timeOrigin` and `Performance.now()` are defined in [hr-time].

The terms **serializable objects**, serialization steps, and deserialization steps are defined in [HTML].

The terms `MediaStream`, `MediaStreamTrack`, and `MediaStreamConstraints` are defined in [GETUSERMEDIA]. Note that `MediaStream` is extended in 9.2 MediaStream in this document while `MediaStreamTrack` is extended in 9.3 MediaStreamTrack in this document.

The term `Blob` is defined in [FILEAPI].

The term **media description** is defined in [RFC4566].

The term **media transport** is defined in [RFC7656].

The term **generation** is defined in [RFC8838] Section 2.

The terms **stats object** and **monitored object** are defined in [WEBRTC-STATS].

When referring to exceptions, the terms throw and created are defined in [WEBIDL].

The callback `VoidFunction` is defined in [WEBIDL].

The term "throw" is used as specified in [INFRA]: it terminates the current processing steps.

The terms **fulfilled**, **rejected**, **resolved**, and **settled** used in the context of Promises are defined in [ECMASCRIPT-6.0].

The **AlgorithmIdentifier** is defined in [WebCryptoAPI].

> ***NOTE***
>
> ***The general principles for Javascript APIs apply, including the principle of run-to-completion and no-data-races as defined in [API-DESIGN-PRINCIPLES]. That is, while a task is running, external events do not influence what's visible to the Javascript application. For example, the amount of data buffered on a data channel will increase due to "send" calls while Javascript is executing, and the decrease due to packets being sent will be visible after a task checkpoint.***
> ***It is the responsibility of the user agent to make sure the set of values presented to the application is consistent - for instance that getContributingSources() (which is synchronous) returns values for all sources measured at the same time.***

§ 4. Peer-to-peer connections

§ 4.1 Introduction

This section is non-normative.

An `RTCPeerConnection` instance allows an application to establish peer-to-peer communications with another `RTCPeerConnection` instance in another browser, or to another endpoint implementing the required protocols. Communications are coordinated by the exchange of control messages (called a signaling protocol) over a signaling channel which is provided by unspecified means, but generally by a script in the page via the server, e.g. using `WebSocket` or `XMLHttpRequest`.

§ 4.2 Configuration

PROPOSED CORRECTION 1: Set default values of the `RTCConfiguration` dictionary, aligning it with current implementations (PR #2691)

PROPOSED CORRECTION 2: Allow an implementation-defined limited to the number of configured ICE Servers (PR #2679)

◉ Show Current and Future ○ Show Current ○ Show Future

~~4.2.1 `RTCConfiguration` Dictionary~~

§ 4.2.1 `RTCConfiguration` Dictionary

The `RTCConfiguration` defines a set of parameters to configure how the peer-to-peer communication established via `RTCPeerConnection` is established or re-established.

```
dictionary RTCConfiguration {
  sequence<RTCIceServer> iceServersiceServers = [];
  RTCIceTransportPolicy iceTransportPolicyiceTransportPolicy = "all";
  RTCBundlePolicy bundlePolicybundlePolicy = "balanced";
  RTCRtcpMuxPolicy rtcpMuxPolicyrtcpMuxPolicy = "require";
  sequence<RTCCertificate> certificatescertificates = [];
  [EnforceRange] octet iceCandidatePoolSize = 0;
};
```

~~*Dictionary `RTCConfiguration` Members*~~

§ *Dictionary `RTCConfiguration` Members*

`iceServers` **of type sequence<`RTCIceServer`>, defaulting to `[]`.**
: An array of objects describing servers available to be used by ICE, such as STUN and TURN servers. If the number of ICE servers exceeds an implementation-defined limit, ignore the ICE servers above the threshold. This implementation defined limit *MUST* be at least 32.

`iceTransportPolicy` **of type RTCIceTransportPolicy, defaulting to `"all"`.**
: Indicates which candidates the ICE Agent is allowed to use.

`bundlePolicy` **of type RTCBundlePolicy, defaulting to `"balanced"`.**
: Indicates which media-bundling policy to use when gathering ICE candidates.

`rtcpMuxPolicy` **of type RTCRtcpMuxPolicy, defaulting to `"require"`.**

Indicates which rtcp-mux policy to use when gathering ICE candidates.

certificates **of type sequence<RTCCertificate>, defaulting to [].**
A set of certificates that the RTCPeerConnection uses to authenticate.

Valid values for this parameter are created through calls to the generateCertificate() function.

Although any given DTLS connection will use only one certificate, this attribute allows the caller to provide multiple certificates that support different algorithms. The final certificate will be selected based on the DTLS handshake, which establishes which certificates are allowed. The RTCPeerConnection implementation selects which of the certificates is used for a given connection; how certificates are selected is outside the scope of this specification.

> ***NOTE***
>
> ***Existing implementations only utilize the first certificate provided; the others are ignored.***

If this value is absent, then a default set of certificates is generated for each RTCPeerConnection instance.

This option allows applications to establish key continuity. An RTCCertificate can be persisted in [INDEXEDDB] and reused. Persistence and reuse also avoids the cost of key generation.

The value for this configuration option cannot change after its value is initially selected.

iceCandidatePoolSize **of type octet, defaulting to 0**
Size of the prefetched ICE pool as defined in [~~RFC8829~~RFC9429] (section 3.5.4. and section 4.1.1.).

PROPOSED CORRECTION 17: Remove single-value RTCIceCredentialType enum (PR #2767)

◉ Show Current and Future ○ Show Current ○ Show Future

~~4.2.2 RTCIceCredentialType Enum~~

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

~~Enumeration description~~

~~password~~	~~The credential is a long-term authentication username and password, as described in [RFC5389], Section 10.2.~~

PROPOSED CORRECTION 17: Remove single-value RTCIceCredentialType enum (PR #2767)

◉ Show Current and Future ○ Show Current ○ Show Future

~~4.2.3 RTCIceServer Dictionary~~

§ 4.2.2 RTCIceServer Dictionary

The RTCIceServer dictionary is used to describe the STUN and TURN servers that can be used by the ICE Agent to establish a connection with a peer.

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

```
  RTCIceCredentialType credentialType = "password";
};
```

~~*Dictionary RTCIceServer Members*~~

§ *Dictionary RTCIceServer Members*

urls **of type (DOMString or sequence<DOMString>), required**
STUN or TURN URI(s) as defined in [RFC7064] and [RFC7065] or other URI types.

username **of type DOMString**
If this `RTCIceServer` object represents a TURN server, ~~and credentialType is "password",~~ then this ~~attribute~~ attribute specifies the username to use ~~with~~ with that TURN server.

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

~~If credentialType is "password",~~ `credential` represents a long-term ~~authentication~~ authentication password, ~~as~~ as described in [RFC5389], Section 10.2.

> ~~***NOTE***~~
>
> ~~***To support additional values of credentialType, credential may evolve in future as a union.***~~

~~***credentialType*** **of type RTCIceCredentialType, defaulting to "password"**~~

~~If this RTCIceServer object represents a TURN server, then this attribute specifies how *credential* should be used when that TURN server requests authorization.~~

An example array of `RTCIceServer` objects is:

EXAMPLE 1

```
[
  {urls: 'stun:stun1.example.net'},
  {urls: ['turns:turn.example.org', 'turn:turn.example.net'],
    username: 'user',
    credential: 'myPassword',
    credentialType: 'password'},
];
```

§ 4.2.3 `RTCIceTransportPolicy` Enum

As described in [RFC9429] (section 4.1.1.), if the `iceTransportPolicy` member of the `RTCConfiguration` is specified, it defines the ICE candidate policy [RFC9429] (section 3.5.3.) the browser uses to surface the permitted candidates to the application; only these candidates will be used for connectivity checks.

WebIDL

```
enum RTCIceTransportPolicy {
  "relay",
  "all"
};
```

`RTCIceTransportPolicy` Enumeration description

Enum value	Description

Enum value	Description
`relay`	The ICE Agent uses only media relay candidates such as candidates passing through a TURN server. ***NOTE*** ***This can be used to prevent the remote endpoint from learning the user's IP addresses, which may be desired in certain use cases. For example, in a "call"-based application, the application may want to prevent an unknown caller from learning the callee's IP addresses until the callee has consented in some way.***
`all`	The ICE Agent can use any type of candidate when this value is specified. ***NOTE*** ***The implementation can still use its own candidate filtering policy in order to limit the IP addresses exposed to the application, as noted in the description of `RTCIceCandidate`.`address`.***

§ 4.2.4 `RTCBundlePolicy` Enum

As described in [RFC9429] (section 4.1.1.), bundle policy affects which media tracks are negotiated if the remote endpoint is not bundle-aware, and what ICE candidates are gathered. If the remote endpoint is bundle-aware, all media tracks and data channels are bundled onto the same transport.

WebIDL

```
enum RTCBundlePolicy {
  "balanced",
  "max-compat",
  "max-bundle"
};
```

`RTCBundlePolicy` Enumeration description

Enum value	Description
`balanced`	Gather ICE candidates for each media type in use (audio, video, and data). If the remote endpoint is not bundle-aware, negotiate only one audio and video track on separate transports.
`max-compat`	Gather ICE candidates for each track. If the remote endpoint is not bundle-aware, negotiate all media tracks on separate transports.
`max-bundle`	Gather ICE candidates for only one track. If the remote endpoint is not bundle-aware, negotiate only one media track.

§ 4.2.5 `RTCRtcpMuxPolicy` Enum

As described in [RFC9429] (section 4.1.1.), the `RTCRtcpMuxPolicy` affects what ICE candidates are gathered to support non-multiplexed RTCP. The only value defined in this spec is "`require`".

WebIDL

```
enum RTCRtcpMuxPolicy {
  "require"
};
```

RTCRtcpMuxPolicy Enumeration description

Enum value	Description
`require`	Gather ICE candidates only for RTP and multiplex RTCP on the RTP candidates. If the remote endpoint is not capable of rtcp-mux, session negotiation will fail.

§ 4.2.6 Offer/Answer Options

These dictionaries describe the options that can be used to control the offer/answer creation process.

WebIDL

```
dictionary RTCOfferAnswerOptions {};
```

§ *Dictionary* `RTCOfferAnswerOptions` *Members*

WebIDL

```
dictionary RTCOfferOptions : RTCOfferAnswerOptions {
  boolean iceRestart = false;
};
```

§ *Dictionary* `RTCOfferOptions` *Members*

***`iceRestart`* of type boolean, defaulting to `false`**

When the value of this dictionary member is `true`, or the relevant `RTCPeerConnection` object's `[[LocalIceCredentialsToReplace]]` slot is not empty, then the generated description will have ICE credentials that are different from the current credentials (as visible in the `currentLocalDescription` attribute's SDP). Applying the generated description will restart ICE, as described in section 9.1.1.1 of [RFC5245].

When the value of this dictionary member is `false`, and the relevant `RTCPeerConnection` object's `[[LocalIceCredentialsToReplace]]` slot is empty, and the `currentLocalDescription` attribute has valid ICE credentials, then the generated description will have the same ICE credentials as the current value from the `currentLocalDescription` attribute.

> ***NOTE***
>
> ***Performing an ICE restart is recommended when `iceConnectionState` transitions to "`failed`". An application may additionally choose to listen for the `iceConnectionState` transition to "`disconnected`" and then use other sources of information (such as using `getStats` to measure if the number of bytes sent or received over the next couple of seconds increases) to determine whether an ICE restart is advisable.***

The **`RTCAnswerOptions`** dictionary describe options specific to session description of type "`answer`" (none in this version of the specification).

WebIDL

```
dictionary RTCAnswerOptions : RTCOfferAnswerOptions {};
```

§ 4.3 State Definitions

§ 4.3.1 `RTCSignalingState` Enum

WebIDL

```
enum RTCSignalingState {
  "stable",
  "have-local-offer",
  "have-remote-offer",
  "have-local-pranswer",
  "have-remote-pranswer",
  "closed"
};
```

`RTCSignalingState` Enumeration description

Enum value	Description
`stable`	There is no offer/answer exchange in progress. This is also the initial state, in which case the local and remote descriptions are empty.
`have-local-offer`	A local description, of type "`offer`", has been successfully applied.
`have-remote-offer`	A remote description, of type "`offer`", has been successfully applied.
`have-local-pranswer`	A remote description of type "`offer`" has been successfully applied and a local description of type "`pranswer`" has been successfully applied.
`have-remote-pranswer`	A local description of type "`offer`" has been successfully applied and a remote description of type "`pranswer`" has been successfully applied.
`closed`	The `RTCPeerConnection` has been closed; its `[[IsClosed]]` slot is `true`.

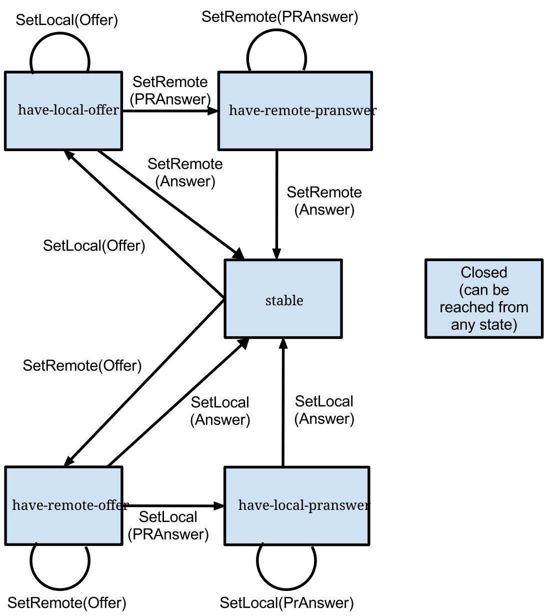


Figure 1 Non-normative signaling state transitions diagram. Method calls abbreviated.

An example set of transitions might be:

Caller transition:

- new RTCPeerConnection(): "`stable`"
- setLocalDescription(offer): "`have-local-offer`"
- setRemoteDescription(pranswer): "`have-remote-pranswer`"
- setRemoteDescription(answer): "`stable`"

Callee transition:

- new RTCPeerConnection(): "`stable`"

- setRemoteDescription(offer): "`have-remote-offer`"
- setLocalDescription(pranswer): "`have-local-pranswer`"
- setLocalDescription(answer): "`stable`"

§ 4.3.2 `RTCIceGatheringState` Enum

WebIDL

```
enum RTCIceGatheringState {
  "new",
  "gathering",
  "complete"
};
```

`RTCIceGatheringState` Enumeration description

Enum value	Description
`new`	Any of the `RTCIceTransport`s are in the "`new`" gathering state and none of the transports are in the "`gathering`" state, or there are no transports.
`gathering`	Any of the `RTCIceTransport`s are in the "`gathering`" state.
`complete`	At least one `RTCIceTransport` exists, and all `RTCIceTransport`s are in the "`complete`" gathering state.

PROPOSED CORRECTION 3: Update `RTCIceGatheringState` to clarify the relevant transport it represents (PR #2680)

Show Current and Future Show Current Show Future

The set of transports considered is the ~~set of transports~~ one presently referenced by the ~~PeerConnection's~~ `RTCPeerConnection`'s set of transceivers and the `RTCPeerConnection`'s `[[SctpTransport]]` internal slot if not `null`.

§ 4.3.3 `RTCPeerConnectionState` Enum

WebIDL

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

PROPOSED CORRECTION 3: Update `RTCPeerConnectionState` to clarify the relevant transport it represents (PR #2680)

PROPOSED CORRECTION 4: Ensure the `connecting` state happens whenever a ICE or DTLS transport is new (PR #2687)

Show Current and Future Show Current Show Future

`RTCPeerConnectionState` Enumeration description

~~**Enumeration description**~~

Enum value	Description
~~closed~~	~~The RTCPeerConnection object's [[IsClosed]] slot is true.~~
closed	[[IceConnectionState]] is "closed".
failed	The previous state doesn't ~~apply~~ apply, and ~~any RTCIceTransports are in the~~ either [[IceConnectionState]] is "failed" ~~state~~ or any RTCDtlsTransports are in the "failed" state.
disconnected	None of the previous states ~~apply and any RTCIceTransports are in the~~ apply, and [[IceConnectionState]] is "disconnected~~" state~~".
new	None of the previous states ~~apply~~ apply, and ~~all RTCIceTransports are in the~~ either [[IceConnectionState]] is "new~~" or "closed" state~~, and all RTCDtlsTransports are in the "new" or "closed" state, or there are no transports.
~~connecting~~	~~None of the previous states apply and any RTCIceTransport is in the "checking" state or any RTCDtlsTransport is in the "connecting" state.~~
connected	None of the previous states ~~apply and all RTCIceTransports are in the "connected"~~apply, [[IceConnectionState]] is "~~completed" or~~ connected"~~closed" state~~, and all RTCDtlsTransports are in the "connected" or "closed" state.
connecting	None of the previous states apply.

NOTE

In the "connecting" state, one or more RTCIceTransports are in the "new" or "checking" state, or one or more RTCDtlsTransports are in the "new" or "connecting" state.

The set of transports considered is the ~~set of transports~~ one presently referenced by the ~~PeerConnection's~~ RTCPeerConnection's set of transceivers and the RTCPeerConnection's [[SctpTransport]] internal slot if not null.

§ 4.3.4 RTCIceConnectionState Enum

WebIDL

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

RTCIceConnectionState Enumeration description

Enum value	Description
closed	The RTCPeerConnection object's [[IsClosed]] slot is true.
failed	The previous state doesn't apply and any RTCIceTransports are in the "failed" state.
disconnected	None of the previous states apply and any RTCIceTransports are in the "disconnected" state.
new	None of the previous states apply and all RTCIceTransports are in the "new" or "closed" state, or there are no transports.
checking	None of the previous states apply and any RTCIceTransports are in the "new" or "checking" state.
completed	None of the previous states apply and all RTCIceTransports are in the "completed" or "closed" state.
connected	None of the previous states apply and all RTCIceTransports are in the "connected", "completed" or "closed" state.

PROPOSED CORRECTION 3: Update `RTCIceConnectionState` to clarify the relevant transport it represents (PR #2680)

Show Current and Future Show Current Show Future

The set of transports considered is the ~~set of transports~~ one presently referenced by the ~~PeerConnection's~~ `RTCPeerConnection`'s set of transceivers and the `RTCPeerConnection`'s `[[SctpTransport]]` internal slot if not `null`.

Note that if an `RTCIceTransport` is discarded as a result of signaling (e.g. RTCP mux or bundling), or created as a result of signaling (e.g. adding a new media description), the state may advance directly from one state to another.

§ 4.4 RTCPeerConnection Interface

The [RFC9429] specification, as a whole, describes the details of how the `RTCPeerConnection` operates. References to specific subsections of [RFC9429] are provided as appropriate.

§ 4.4.1 Operation

Calling `new RTCPeerConnection(configuration)` creates an `RTCPeerConnection` object.

configuration.`iceServers` contains information used to find and access the servers used by ICE. The application can supply multiple servers of each type, and any TURN server *MAY* also be used as a STUN server for the purposes of gathering server reflexive candidates.

An `RTCPeerConnection` object has a `[[SignalingState]]`, and the aggregated states `[[ConnectionState]]`, `[[IceGatheringState]]`, and `[[IceConnectionState]]`. These are initialized when the object is created.

The ICE protocol implementation of an `RTCPeerConnection` is represented by an **ICE agent** [RFC5245]. Certain `RTCPeerConnection` methods involve interactions with the ICE Agent, namely `addIceCandidate`, `setConfiguration`, `setLocalDescription`, `setRemoteDescription` and `close`. These interactions are described in the relevant sections in this document and in [RFC9429]. The ICE Agent also provides indications to the user agent when the state of its internal representation of an `RTCIceTransport` changes, as described in 5.6 `RTCIceTransport` Interface.

The task source for the tasks listed in this section is the networking task source.

> ***NOTE***
>
> ***The state of the SDP negotiation is represented by the internal variables `[[SignalingState]]`, `[[CurrentLocalDescription]]`, `[[CurrentRemoteDescription]]`, `[[PendingLocalDescription]]` and `[[PendingRemoteDescription]]`. These are only set inside the `setLocalDescription` and `setRemoteDescription` operations, and modified by the `addIceCandidate` operation and the surface a candidate procedure. In each case, all the modifications to all the five variables are completed before the procedures fire any events or invoke any callbacks, so the modifications are made visible at a single point in time.***

As one of the unloading document cleanup steps, run the following steps:

1. Let *window* be *document*'s relevant global object.
2. For each `RTCPeerConnection` object *connection* whose relevant global object is *window*, close the connection with *connection* and the value `true`.

§ *4.4.1.1 Constructor*

When the **`RTCPeerConnection.constructor()`** is invoked, the user agent *MUST* run the following steps:

1. If any of the steps enumerated below fails for a reason not specified here, throw an `UnknownError` with the `message` attribute set to an appropriate description.
2. Let *connection* be a newly created `RTCPeerConnection` object.
3. Let *connection* have a **[[DocumentOrigin]]** internal slot, initialized to the relevant settings object's origin.
4. Let *configuration* be the method's first argument.
5. If the `certificates` value in *configuration* is non-empty, run the following steps for each *certificate* in certificates:
    1. If the value of *certificate*.`expires` is less than the current time, throw an `InvalidAccessError`.
    2. If *certificate*.`[[Origin]]` is not *same origin* with *connection*.`[[DocumentOrigin]]`, throw an `InvalidAccessError`.
    3. Store *certificate*.
6. Else, generate one or more new `RTCCertificate` instances with this `RTCPeerConnection` instance and store them. This *MAY* happen asynchronously and the value of `certificates` remains `undefined` for the subsequent steps. As noted in Section 4.3.2.3 of [RFC8826], WebRTC utilizes self-signed rather than Public Key Infrastructure (PKI) certificates, so that the expiration check is to ensure that keys are not used indefinitely and additional certificate checks are unnecessary.
7. Initialize *connection*'s ICE Agent.
8. Let *connection* have a **[[Configuration]]** internal slot, initialized to `null`. Set the configuration specified by *configuration*.
9. Let *connection* have an **[[IsClosed]]** internal slot, initialized to `false`.
10. Let *connection* have a **[[NegotiationNeeded]]** internal slot, initialized to `false`.
11. Let *connection* have an **[[SctpTransport]]** internal slot, initialized to `null`.
12. Let *connection* have a **[[DataChannels]]** internal slot, initialized to an empty ordered set.
13. Let *connection* have an **[[Operations]]** internal slot, representing an operations chain, initialized to an empty list.
14. Let *connection* have a **[[UpdateNegotiationNeededFlagOnEmptyChain]]** internal slot, initialized to `false`.
15. Let *connection* have an **[[LastCreatedOffer]]** internal slot, initialized to `""`.
16. Let *connection* have an **[[LastCreatedAnswer]]** internal slot, initialized to `""`.
17. Let *connection* have an **[[EarlyCandidates]]** internal slot, initialized to an empty list.
18. Let *connection* have an **[[SignalingState]]** internal slot, initialized to "`stable`".
19. Let *connection* have an **[[IceConnectionState]]** internal slot, initialized to "`new`".
20. Let *connection* have an **[[IceGatheringState]]** internal slot, initialized to "`new`".
21. Let *connection* have an **[[ConnectionState]]** internal slot, initialized to "`new`".
22. Let *connection* have a **[[PendingLocalDescription]]** internal slot, initialized to `null`.
23. Let *connection* have a **[[CurrentLocalDescription]]** internal slot, initialized to `null`.
24. Let *connection* have a **[[PendingRemoteDescription]]** internal slot, initialized to `null`.

25. Let *connection* have a **[[CurrentRemoteDescription]]** internal slot, initialized to `null`.

26. Let *connection* have a **[[LocalIceCredentialsToReplace]]** internal slot, initialized to an empty set.

27. Return *connection*.

§ *4.4.1.2 Chain an asynchronous operation*

An `RTCPeerConnection` object has an **operations chain**, `[[Operations]]`, which ensures that only one asynchronous operation in the chain executes concurrently. If subsequent calls are made while the returned promise of a previous call is still not settled, they are added to the chain and executed when all the previous calls have finished executing and their promises have settled.

To **chain an operation** to an `RTCPeerConnection` object's operations chain, run the following steps:

1. Let *connection* be the `RTCPeerConnection` object.
2. If *connection*.`[[IsClosed]]` is `true`, return a promise rejected with a newly created `InvalidStateError`.
3. Let *operation* be the operation to be chained.
4. Let *p* be a new promise.
5. Append *operation* to `[[Operations]]`.
6. If the length of `[[Operations]]` is exactly 1, execute *operation*.
7. Upon fulfillment or rejection of the promise returned by the *operation*, run the following steps:
    1. If *connection*.`[[IsClosed]]` is `true`, abort these steps.
    2. If the promise returned by *operation* was fulfilled with a value, fulfill *p* with that value.
    3. If the promise returned by *operation* was rejected with a value, reject *p* with that value.
    4. Upon fulfillment or rejection of *p*, execute the following steps:
        1. If *connection*.`[[IsClosed]]` is `true`, abort these steps.
        2. Remove the first element of `[[Operations]]`.
        3. If `[[Operations]]` is non-empty, execute the operation represented by the first element of `[[Operations]]`, and abort these steps.
        4. If *connection*.`[[UpdateNegotiationNeededFlagOnEmptyChain]]` is `false`, abort these steps.
        5. Set *connection*.`[[UpdateNegotiationNeededFlagOnEmptyChain]]` to `false`.
        6. Update the negotiation-needed flag for *connection*.
8. Return *p*.

§ *4.4.1.3 Update the connection state*

An `RTCPeerConnection` object has an aggregated `[[ConnectionState]]`. Whenever the state of an `RTCDtlsTransport` changes, the user agent *MUST* queue a task that runs the following steps:

1. Let *connection* be this `RTCPeerConnection` object associated with the `RTCDtlsTransport` object whose state changed.
2. **PROPOSED CORRECTION 35:** Don't fire connectionstatechange on pc.close() (PR #2876)

◉ Show Current and Future ○ Show Current ○ Show Future

If *connection*.`[[IsClosed]]` is `true`, abort these steps.

3. Let *newState* be the value of deriving a new state value as described by the `RTCPeerConnectionState` enum.

4. If *connection*.`[[ConnectionState]]` is equal to *newState*, abort these steps.

5. Set *connection*.`[[ConnectionState]]` to *newState*.

6. Fire an event named `connectionstatechange` at *connection*.

§ *4.4.1.4 Set the session description*

To **set a local session description** *description* on an `RTCPeerConnection` object *connection*, set the session description *description* on *connection* with the additional value `false`.

To **set a remote session description** *description* on an `RTCPeerConnection` object *connection*, set the session description *description* on *connection* with the additional value `true`.

To **set a session description** *description* on an `RTCPeerConnection` object *connection*, given a *remote* boolean, run the following steps:

1. Let *p* be a new promise.

2. If *description*.`type` is "`rollback`" and *connection*.`[[SignalingState]]` is either "`stable`", "`have-local-pranswer`", or "`have-remote-pranswer`", then reject *p* with a newly created `InvalidStateError` and abort these steps.

3. Let *jsepSetOfTransceivers* be a shallow copy of *connection*'s set of transceivers.

4. In parallel, start the process to apply *description* as described in [RFC9429] (section 5.5. and section 5.6.), with these additional restrictions:

   1. Use *jsepSetOfTransceivers* as the source of truth with regard to what "RtpTransceivers" exist, and their `[[JsepMid]]` internal slot as their "mid property".

   2. CANDIDATE CORRECTION 5: Forbid ICE gathering and connectivity checks on administrative prohibited candidates (PR #2708)

   ◉ Show Current and Future ○ Show Current ○ Show Future

   If *remote* is `false` and this triggers the ICE candidate gathering process in [RFC9429] (section 5.9.), the ICE Agent *MUST NOT* gather candidates that would be administratively prohibited.

   3. CANDIDATE CORRECTION 5: Forbid ICE gathering and connectivity checks on administrative prohibited candidates (PR #2708)

   ◉ Show Current and Future ○ Show Current ○ Show Future

   If *remote* is `true` and this triggers ICE connectivity checks in [RFC9429] (section 5.10.), the ICE Agent *MUST NOT* attempt to connect to candidates that are administratively prohibited.

   4. If *remote* is `true`, validate back-to-back offers as if answers were applied in between, by running the check for subsequent offers as if it were in stable state.

   CANDIDATE CORRECTION 37: Don't fail sRD(offer) over rid mismatch, just answer with unicast. (PR #2794)

   ◉ Show Current and Future ○ Show Current ○ Show Future

5. ~~If applying *description* leads to modifying a transceiver *transceiver*, and *transceiver*.[[Sender]].[[SendEncodings]] is non-empty, and not equal to the encodings that would result from processing *description*, the process of applying *description* fails. This specification does not allow remotely initiated RID renegotiation.~~

6. If the process to apply *description* fails for any reason, then the user agent *MUST* queue a task that runs the following steps:

    1. If *connection*.`[[IsClosed]]` is `true`, then abort these steps.
    2. If *description*.`type` is invalid for the current *connection*.`[[SignalingState]]` as described in [RFC9429] (section 5.5. and section 5.6.), then reject *p* with a newly created `InvalidStateError` and abort these steps.
    3. If the content of *description* is not valid SDP syntax, then reject *p* with an `RTCError` (with `errorDetail` set to "`sdp-syntax-error`" and the `sdpLineNumber` attribute set to the line number in the SDP where the syntax error was detected) and abort these steps.
    4. If *remote* is `true`, the *connection*'s `RTCRtcpMuxPolicy` is `require` and the description does not use RTCP mux, then reject *p* with a newly created `InvalidAccessError` and abort these steps.
    5. If the description attempted to renegotiate RIDs, as described above, then reject *p* with a newly created `InvalidAccessError` and abort these steps.
    6. If the content of *description* is invalid, then reject *p* with a newly created `InvalidAccessError` and abort these steps.
    7. For all other errors, reject *p* with a newly created `OperationError`.

7. If *description* is applied successfully, the user agent *MUST* queue a task that runs the following steps:

    1. If *connection*.`[[IsClosed]]` is `true`, then abort these steps.
    2. If *remote* is `true` and *description* is of type "`offer`", then if any `addTrack()` methods on *connection* succeeded during the process to apply *description*, abort these steps and start the process over as if they had succeeded prior, to include the extra transceiver(s) in the process.
    3. If any promises from `setParameters` methods on `RTCRtpSender`s associated with *connection* are not settled, abort these steps and start the process over.
    4. If *description* is of type "`offer`" and *connection*.`[[SignalingState]]` is "`stable`" then for each *transceiver* in *connection*'s set of transceivers, run the following steps:

        1. Set *transceiver*.`[[Sender]]`.`[[LastStableStateSenderTransport]]` to *transceiver*.`[[Sender]]`.`[[SenderTransport]]`.
        2. CANDIDATE CORRECTION 13: Rollback restores ridless encoding trounced by sRD(simulcastOffer). (PR #2797)

            Show Current and Future ○ Show Current ○ Show Future

            If *transceiver*.`[[Sender]]`.`[[SendEncodings]]`.length is 1 and the lone encoding contains no `rid` member, then set *transceiver*.`[[Sender]]`.`[[LastStableRidlessSendEncodings]]` to *transceiver*.`[[Sender]]`.`[[SendEncodings]]`; Otherwise, set *transceiver*.`[[Sender]]`.`[[LastStableRidlessSendEncodings]]` to null.
        3. Set *transceiver*.`[[Receiver]]`.`[[LastStableStateReceiverTransport]]` to *transceiver*.`[[Receiver]]`.`[[ReceiverTransport]]`.
        4. Set *transceiver*.`[[Receiver]]`.`[[LastStableStateAssociatedRemoteMediaStreams]]` to *transceiver*.`[[Receiver]]`.

`[[AssociatedRemoteMediaStreams]]`.

5. Set *transceiver*.`[[Receiver]]`.`[[LastStableStateReceiveCodecs]]` to *transceiver*.`[[Receiver]]`.`[[ReceiveCodecs]]`.

5. If *remote* is `false`, then run one of the following steps:

   1. If *description* is of type "`offer`", set *connection*.`[[PendingLocalDescription]]` to a new `RTCSessionDescription` object constructed from *description*, set *connection*.`[[SignalingState]]` to "`have-local-offer`", and release early candidates.

   2. If *description* is of type "`answer`", then this completes an offer answer negotiation. Set *connection*.`[[CurrentLocalDescription]]` to a new `RTCSessionDescription` object constructed from *description*, and set *connection*.`[[CurrentRemoteDescription]]` to *connection*.`[[PendingRemoteDescription]]`. Set both *connection*.`[[PendingRemoteDescription]]` and *connection*.`[[PendingLocalDescription]]` to `null`. Set both *connection*.`[[LastCreatedOffer]]` and *connection*.`[[LastCreatedAnswer]]` to `""`, set *connection*.`[[SignalingState]]` to "`stable`", and release early candidates. Finally, if none of the ICE credentials in *connection*.`[[LocalIceCredentialsToReplace]]` are present in *description*, then set *connection*.`[[LocalIceCredentialsToReplace]]` to an empty set.

   3. If *description* is of type "`pranswer`", then set *connection*.`[[PendingLocalDescription]]` to a new `RTCSessionDescription` object constructed from *description*, set *connection*.`[[SignalingState]]` to "`have-local-pranswer`", and release early candidates.

6. Otherwise, (if *remote* is `true`) run one of the following steps:

   1. If *description* is of type "`offer`", set *connection*.`[[PendingRemoteDescription]]` attribute to a new `RTCSessionDescription` object constructed from *description*, and set *connection*.`[[SignalingState]]` to "`have-remote-offer`".

   2. If *description* is of type "`answer`", then this completes an offer answer negotiation. Set *connection*.`[[CurrentRemoteDescription]]` to a new `RTCSessionDescription` object constructed from *description*, and set *connection*.`[[CurrentLocalDescription]]` to *connection*.`[[PendingLocalDescription]]`. Set both *connection*.`[[PendingRemoteDescription]]` and *connection*.`[[PendingLocalDescription]]` to `null`. Set both *connection*.`[[LastCreatedOffer]]` and *connection*.`[[LastCreatedAnswer]]` to `""`, and set *connection*.`[[SignalingState]]` to "`stable`". Finally, if none of the ICE credentials in *connection*.`[[LocalIceCredentialsToReplace]]` are present in the newly set *connection*.`[[CurrentLocalDescription]]`, then set *connection*.`[[LocalIceCredentialsToReplace]]` to an empty set.

   3. If *description* is of type "`pranswer`", then set *connection*.`[[PendingRemoteDescription]]` to a new `RTCSessionDescription` object constructed from *description* and set *connection*.`[[SignalingState]]` to "`have-remote-pranswer`".

7. If *description* is of type "`answer`", and it initiates the closure of an existing SCTP association, as defined in [RFC8841], Sections 10.3 and 10.4, set the value of *connection*.`[[SctpTransport]]` to `null`.

8. Let *trackEventInits*, *muteTracks*, *addList*, *removeList* and *errorList* be empty lists.

9. If *description* is of type "`answer`" or "`pranswer`", then run the following steps:

   1. If *description* initiates the establishment of a new SCTP association, as defined in [RFC8841], Sections 10.3 and 10.4, create an RTCSctpTransport with an initial state of "`connecting`" and assign the result to the `[[SctpTransport]]` slot. Otherwise, if an SCTP association is established, but the `max-message-size` SDP attribute is updated, update the data max message size of *connection*.`[[SctpTransport]]`.

   2. If *description* negotiates the DTLS role of the SCTP transport, then for each `RTCDataChannel`, *channel*, with a `null` `id`, run the following step:

      1. Give *channel* a new ID generated according to [RFC8832]. If no available ID

could be generated, set *channel*.`[[ReadyState]]` to "`closed`", and add *channnel* to *errorList*.

10. If *description* is not of type "`rollback`", then run the following steps:

    1. If *remote* is `false`, then run the following steps for each media description in *description*:

       CANDIDATE CORRECTION 26: Prune createAnswer()'s encodings and SendEncodings in sLD(answer). (PR #2801)

       Show Current and Future Show Current Show Future

       1. If the media description was not yet associated with an `RTCRtpTransceiver` object then run the following steps:

          1. Let *transceiver* be the `RTCRtpTransceiver` used to create the media description.

          2. Set *transceiver*.`[[Mid]]` to *transceiver*.`[[JsepMid]]`.

          3. If *transceiver*.`[[Stopped]]` is `true`, abort these sub steps.

          4. If the media description is indicated as using an existing ~~media~~ media transport according to [RFC8843], let *transport* be the `RTCDtlsTransport` object representing the RTP/RTCP component of that transport.

          5. Otherwise, let *transport* be a newly created `RTCDtlsTransport` object with a new underlying `RTCIceTransport`.

          6. Set *transceiver*.`[[Sender]]`.`[[SenderTransport]]` to *transport*.

          7. Set *transceiver*.`[[Receiver]]`.`[[ReceiverTransport]]` to *transport*.

       2. Let *transceiver* be the `RTCRtpTransceiver` associated with the media description.

       3. If *transceiver*.`[[Stopped]]` is `true`, abort these sub steps.

       4. Let *direction* be an `RTCRtpTransceiverDirection` value representing the direction from the ~~media~~ media description.

       5. If *direction* is "`sendrecv`" or "`recvonly`", set *transceiver*.`[[Receptive]]` to `true`, otherwise set it to `false`.

       6. Set *transceiver*.`[[Receiver]]`.`[[ReceiveCodecs]]` to the codecs that *description* negotiates for receiving and which the user agent is currently prepared to receive.

          NOTE

          If the direction is "`sendonly`" or "`inactive`", the receiver is not prepared to receive anything, and the list will be empty.

       7. If *description* is of type "`answer`" or "`pranswer`", then run the following steps:

          1. If *transceiver*.`[[Sender]]`.`[[SendEncodings]]`.length is greater than 1, then run the following steps:

             1. If *description* is missing all of the previously negotiated layers, then remove all dictionaries in *transceiver*.`[[Sender]]`.`[[SendEncodings]]` except the first one, and skip the next step.

             2. If *description* is missing any of the previously negotiated layers, then remove the dictionaries that correspond to the missing layers from *transceiver*.`[[Sender]]`.`[[SendEncodings]]`.

          2. Set *transceiver*.`[[Sender]]`.`[[SendCodecs]]` to the codecs that

description negotiates for sending and which the user agent is currently capable of sending, and set *transceiver*.`[[Sender]]`.`[[LastReturnedParameters]]` to `null`.

3. If *direction* is "`sendonly`" or "`inactive`", and *transceiver*.`[[FiredDirection]]` is either "`sendrecv`" or "`recvonly`", then run the following steps:

    1. Set the associated remote streams given *transceiver*.`[[Receiver]]`, an empty list, another empty list, and *removeList*.

    2. process the removal of a ~~remote~~ remote track for the media description, given *transceiver* and *muteTracks*.

4. Set *transceiver*.`[[CurrentDirection]]` and *transceiver*.`[[FiredDirection]]` to *direction*.

2. Otherwise, (if *remote* is `true`) run the following steps for each media description in *description*:

CANDIDATE CORRECTION 12: Remove interaction between encoding.active and simulcast ~rid (PR #2754)

CANDIDATE CORRECTION 14: Make RTCTransceiver.direction reflects local preference in offers and answers (PR #2759)

CANDIDATE CORRECTION 22: Allow remote offer rid pruning of encodings through the client answer. (PR #2758)

CANDIDATE CORRECTION 37: Don't fail sRD(offer) over rid mismatch, just answer with unicast. (PR #2794)

CANDIDATE CORRECTION 25: Remove duplicate rids in proposedSendEncodings. (PR #2800)

CANDIDATE CORRECTION 27: Ignore comma-separated rid alternatives. (PR #2813)

Show Current and Future Show Current Show Future

1. If the *description* is of type "`offer`" and the media description contains a request to receive simulcast, use the order of the rid values specified in the simulcast attribute to create an `RTCRtpEncodingParameters` dictionary for each of the simulcast layers, populating the `rid` member according to the corresponding rid ~~value~~value (using only the first value if comma-separated alternatives exist), and let ~~*sendEncodings*~~*proposedSendEncodings* be ~~the list~~ the list containing the created dictionaries. Otherwise, let ~~*sendEncodings*~~ *proposedSendEncodings* be ~~an~~ an empty list.

2. For each encoding, *encoding*, in *proposedSendEncodings* in reverse order, if *encoding*'s `rid` matches that of another encoding in *proposedSendEncodings*, remove *encoding* from *proposedSendEncodings*.

3. Let *supportedEncodings* be the maximum number of encodings that the implementation can support. If the length of ~~*sendEncodings*~~*proposedSendEncodings* is greater than *supportedEncodings*, truncate ~~*sendEncodings*~~*proposedSendEncodings* so that its length is *supportedEncodings*.

4. If ~~*sendEncodings*~~*proposedSendEncodings* is non-empty, ~~set~~ set each encoding's `scaleResolutionDownBy` to `2^(length of` ~~*sendEncodings*~~*proposedSendEncodings* `- encoding index - 1)`.

5. As described by [~~RFC8829~~RFC9429] (section 5.10.), attempt to find an existing

RTCRtpTransceiver object, *transceiver*, to represent the media description.

6. If a suitable transceiver was found (*transceiver* is set), and ~~*sendEncodings*~~*proposedSendEncodings* is non-empty,~~ set *transceiver*.[[Sender]].[[SendEncodings]] to *sendEncodings*, and set *transceiver*.[[Sender]].[[LastReturnedParameters]] to null.~~run the following steps:

    1. If the length of *transceiver*.[[Sender]].[[SendEncodings]] is 1, and the lone encoding contains no rid member, set *transceiver*.[[Sender]].[[SendEncodings]] to *proposedSendEncodings*, and set *transceiver*.[[Sender]].[[LastReturnedParameters]] to null.

7. If no suitable transceiver was found (*transceiver* is unset), run the following steps:

    1. Create an RTCRtpSender, *sender*, from the ~~media~~ media description using ~~*sendEncodings*~~*proposedSendEncodings*.

    2. Create an RTCRtpReceiver, *receiver*, from the ~~media~~ media description.

    3. Create an RTCRtpTransceiver with *sender*, *receiver* and an RTCRtpTransceiverDirection value of "recvonly", and let *transceiver* be the result.

    4. Add *transceiver* to the *connection*'s set ~~of~~ of transceivers.

8. If *description* is of type "answer" or "pranswer", and *transceiver*. [[Sender]].[[SendEncodings]] .length is greater than 1, then run the following steps:

    1. If *description* indicates that simulcast is not supported or desired, or *description* is missing all of the previously negotiated layers, then remove all dictionaries in *transceiver*.[[Sender]].[[SendEncodings]] except the first one and abort these sub steps.

    2. If *description* ~~rejects~~ is missing any of the ~~offered~~ previously negotiated layers, ~~then~~ then remove ~~the~~ the dictionaries that correspond ~~to rejected~~ to the missing layers from *transceiver*.[[Sender]].[[SendEncodings]].

    3. ~~Update the paused status as indicated by [RFC8853] of each simulcast layer by setting the active member on the corresponding dictionaries in *transceiver*.[[Sender]].[[SendEncodings]] to true for unpaused or to false for paused.~~

9. Set *transceiver*.[[Mid]] to *transceiver*.[[JsepMid]].

10. Let *direction* be an RTCRtpTransceiverDirection value representing the direction from the ~~media~~ media description, but with the send and receive directions reversed to represent this peer's point of view. If the media description is rejected, set *direction* to "inactive".

11. If *direction* is "sendrecv" or "recvonly", let *msids* be a list of the MSIDs that the media description indicates *transceiver*.[[Receiver]].[[ReceiverTrack]] is to be associated with. Otherwise, let *msids* be an empty list.

    > ***NOTE***
    > ***msids will be an empty list here if media description is rejected.***

12. Process remote tracks with *transceiver*, *direction*, *msids*, *addList*, *removeList*, and *trackEventInits*.

13. Set *transceiver*.[[Receiver]].[[ReceiveCodecs]] to the codecs that *description* negotiates for receiving and which the user agent is currently prepared to receive.

14. If *description* is of type "answer" or "pranswer", then run the following steps:

    1. Set *transceiver*.[[Sender]].[[SendCodecs]] to the codecs that

description negotiates for sending and which the user agent is currently capable of sending.

2. Set *transceiver*.`[[CurrentDirection]]` ~~and *transceiver*.[[Direction]]s~~ to *direction*.

3. Let *transport* be the `RTCDtlsTransport` object representing the RTP/RTCP component of the ~~media~~ media transport used by *transceiver*'s associated media description, according to [RFC8843].

4. Set *transceiver*.`[[Sender]]`.`[[SenderTransport]]` to *transport*.

5. Set *transceiver*.`[[Receiver]]`.`[[ReceiverTransport]]` to *transport*.

6. Set the `[[IceRole]]` of *transport* according to the rules of [RFC8445].

> ***NOTE***
>
> ***The rules of [RFC8445] that apply here are:***
>
> - ***If `[[IceRole]]` is not `unknown`, do not modify `[[IceRole]]`.***
> - ***If description is a local offer, set it to `controlling`.***
> - ***If description is a remote offer, and contains `a=ice-lite`, set `[[IceRole]]` to `controlling`.***
> - ***If description is a remote offer, and does not contain `a=ice-lite`, set `[[IceRole]]` to `controlled`.***
>
> ***This ensures that `[[IceRole]]` always has a value after the first offer is processed.***

15. If the media description is rejected, and *transceiver*.`[[Stopped]]` is `false`, then stop ~~the~~ the RTCRtpTransceiver *transceiver*.

11. Otherwise, (if *description* is of type "`rollback`") run the following steps:

1. Let *pendingDescription* be either *connection*.`[[PendingLocalDescription]]` or *connection*.`[[PendingRemoteDescription]]`, whichever one is not `null`.

2. For each *transceiver* in the *connection*'s set of transceivers run the following steps:

1. If *transceiver* was not associated with a media description prior to *pendingDescription* being set, disassociate it and set both *transceiver*.`[[JsepMid]]` and *transceiver*.`[[Mid]]` to `null`.

2. Set *transceiver*.`[[Sender]]`.`[[SenderTransport]]` to *transceiver*.`[[Sender]]`.`[[LastStableStateSenderTransport]]`.

3. CANDIDATE CORRECTION 13: Rollback restores ridless encoding trounced by sRD(simulcastOffer). (PR #2797)

Show Current and Future ○ Show Current ○ Show Future

If *transceiver*.`[[Sender]]`.`[[LastStableRidlessSendEncodings]]` is not `null`, and any encoding in *transceiver*.`[[Sender]]`.`[[SendEncodings]]` contains a `rid` member, then set *transceiver*.`[[Sender]]`.`[[SendEncodings]]` to *transceiver*.`[[Sender]]`.`[[LastStableRidlessSendEncodings]]`.

4. Set *transceiver*.`[[Receiver]]`.`[[ReceiverTransport]]` to *transceiver*.`[[Receiver]]`.`[[LastStableStateReceiverTransport]]`.

5. Set *transceiver*.`[[Receiver]]`.`[[ReceiveCodecs]]` to *transceiver*.`[[Receiver]]`.`[[LastStableStateReceiveCodecs]]`.

6. If *connection*.`[[SignalingState]]` is "`have-remote-offer`", run the following sub steps:

1. Let *msids* be a list of the `id`s of all `MediaStream` objects in *transceiver*.

[[Receiver]].[[LastStableStateAssociatedRemoteMediaStreams]], or an empty list if there are none.

2. Process remote tracks with *transceiver*, *transceiver*.[[CurrentDirection]], *msids*, *addList*, *removeList*, and *trackEventInits*.

7. If *transceiver* was created when *pendingDescription* was set, and a track has never been attached to it via addTrack(), then stop the RTCRtpTransceiver *transceiver*, and remove it from *connection*'s set of transceivers.

3. Set *connection*.[[PendingLocalDescription]] and *connection*.[[PendingRemoteDescription]] to null, and set *connection*.[[SignalingState]] to "stable".

12. If *description* is of type "answer", then run the following steps:

    1. For each *transceiver* in the *connection*'s set of transceivers run the following steps:

        1. If *transceiver* is stopped, associated with an m= section and the associated m= section is rejected in *connection*.[[CurrentLocalDescription]] or *connection*.[[CurrentRemoteDescription]], remove the *transceiver* from the *connection*'s set of transceivers.

13. If *connection*.[[SignalingState]] is now "stable", run the following steps:

    1. For any *transceiver* that was removed from the set of transceivers in a previous step, if any of its transports (*transceiver*.[[Sender]].[[SenderTransport]] or *transceiver*.[[Receiver]].[[ReceiverTransport]]) are still not closed and they're no longer referenced by a non-stopped transceiver, close the RTCDtlsTransports and their associated RTCIceTransports. This results in events firing on these objects in a queued task.

    2. CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

    Show Current and Future Show Current Show Future

    For each *transceiver* in *connection*'s set of transceivers:

        1. Let *codecs* be *transceiver*.[[Sender]].[[SendCodecs]].

        2. If *codecs* is not an empty list:

            1. For each *encoding* in *transceiver*.[[Sender]].[[SendEncodings]], if *encoding*.codec does not match any entry in *codecs*, remove *encoding*[codec].

    3. Clear the negotiation-needed flag and update the negotiation-needed flag.

14. If *connection*.[[SignalingState]] changed above, fire an event named signalingstatechange at *connection*.

15. For each *channel* in *errorList*, fire an event named error using the RTCErrorEvent interface with the errorDetail attribute set to "data-channel-failure" at *channel*.

16. For each *track* in *muteTracks*, set the muted state of *track* to the value true.

17. For each *stream* and *track* pair in *removeList*, remove the track *track* from *stream*.

18. For each *stream* and *track* pair in *addList*, add the track *track* to *stream*.

19. For each entry *entry* in *trackEventInits*, fire an event named track using the RTCTrackEvent interface with its receiver attribute initialized to *entry*.receiver, its track attribute initialized to *entry*.track, its streams attribute initialized to *entry*.streams and its transceiver attribute initialized to *entry*.transceiver at the *connection* object.

20. Resolve *p* with undefined.

5. Return *p*.

§ *4.4.1.5 Set the configuration*

To **set a configuration** with *configuration*, run the following steps:

PROPOSED CORRECTION 1: Set default values of the `RTCConfiguration` dictionary, aligning it with current implementations (PR #2691)

PROPOSED CORRECTION 6: Validate ICE transport settings upfront when setting a configuration (PR #2689)

◉ Show Current and Future ○ Show Current ○ Show Future

1. ~~Let *configuration* be the `RTCConfiguration` dictionary to be processed.~~
2. Let *connection* be the target `RTCPeerConnection` object.
3. Let *oldConfig* be *connection*.`[[Configuration]]`.
4. If ~~*configuration*.`certificates`~~ *oldConfig* is ~~set~~not `null`, run ~~the~~ the following steps~~:~~, and if any of them fail, throw an `InvalidModificationError`:
    1. If the length of *configuration*.`certificates` is different from the length of ~~*connection*~~*oldConfig*.~~[[Configuration]].~~`certificates`, ~~throw an `InvalidModificationError`~~fail.
    2. Let *index* be ~~initialized to~~ 0.
    3. ~~Let *size* be initialized to the length of *configuration*.`certificates`.~~
    4. While *index* is less than ~~*size*, run~~ the ~~following steps:~~ length of *configuration*.

       `certificates`, run the following steps:
        1. If the ECMAScript object represented by the value of *configuration*.`certificates` at *index* is not the same as the ECMAScript object represented by the value of ~~*connection*~~*oldConfig*.~~[[Configuration]].~~`certificates` at *index*, ~~throw an `InvalidModificationError`~~then fail.
        2. Increment *index* by 1.
    5. If the value of *configuration*.`bundlePolicy` differs from *oldConfig*.`bundlePolicy`, then fail.
    6. If the value of *configuration*.`rtcpMuxPolicy` differs from *oldConfig*.`rtcpMuxPolicy`, then fail.
    7. If the value of *configuration*.`iceCandidatePoolSize` differs from *oldConfig*.`iceCandidatePoolSize`, and `setLocalDescription` has already been called, then fail.
5. ~~If the value of *configuration*.`bundlePolicy` is set and its value differs from the *connection*'s bundle policy, throw an `InvalidModificationError`.~~
6. ~~If the value of *configuration*.`rtcpMuxPolicy` is set and its value differs from the *connection*'s rtcpMux policy, throw an `InvalidModificationError`.~~
7. ~~If the value of *configuration*.`iceCandidatePoolSize` is set and its value differs from the *connection*'s previously set `iceCandidatePoolSize`, and `setLocalDescription` has already been called, throw an `InvalidModificationError`.~~
8. Let *iceServers* be *configuration*.`iceServers`.
9. Truncate *iceServers* to the maximum number of supported elements.
10. For each *server* in *iceServers*, run the following steps:
    1. Let *urls* be *server*.`urls`.
    2. If *urls* is a string, set *urls* to a list consisting of just that string.

3. If *urls* is empty, throw a "SyntaxError" DOMException.

4. For each *url* in *urls*, run the validate an ICE server URL algorithm on *url*.

11. Set the ICE Agent's **ICE transports setting** to the value of *configuration*.iceTransportPolicy. As defined in [~~RFC8829~~RFC9429] (section 4.1.18.), if the new ~~ICE~~ ICE transports setting changes the existing setting, no action will be taken until the next gathering phase. If a script wants this to happen immediately, it should do an ICE restart.

12. Set the ICE Agent's prefetched **ICE candidate pool size** as defined in [~~RFC8829~~RFC9429] (section 3.5.4. and section 4.1.1.) to the value of *configuration*.iceCandidatePoolSize. If the new ICE candidate pool size changes the existing setting, this may result in immediate gathering of new pooled candidates, or discarding of existing pooled candidates, as defined in [~~RFC8829~~RFC9429] (section 4.1.18.).

13. ~~Let *validatedServers* be an empty list.~~

14. ~~If *configuration*.iceServers is defined, then run the following steps for each element:~~

1. ~~Let *server* be the current list element.~~

2. ~~Let *urls* be *server*.urls.~~

3. ~~If *urls* is a string, set *urls* to a list consisting of just that string.~~

4. ~~If *urls* is empty, throw a SyntaxError.~~

5. ~~For each *url* in *urls*, run the [=validate an ICE server URL=] algorithm on *url*.~~

6. ~~Append *server* to *validatedServers*.~~

15. Set the ICE Agent's **ICE servers list** ~~to *validatedServers*~~to *iceServers*.

As defined in [~~RFC8829~~RFC9429] (section 4.1.18.), if a new list of servers replaces the ICE Agent's existing ~~ICE servers list~~ICE servers list, no action will be taken until the next gathering phase. If a script wants this to happen immediately, it should do an ICE restart. However, if the ~~ICE~~ ICE candidate pool has a nonzero size, any ~~existing~~ existing pooled candidates will be discarded, and new candidates will be gathered from the new servers.

16. Store *configuration* in the [[Configuration]] internal slot.

To **validate an ICE server URL** *url*, run the following steps:

CANDIDATE CORRECTION 33: Use the url spec to parse ice server urls (PR #2853)

◉ Show Current and Future ○ Show Current ○ Show Future

1. ~~Parse the *url* using the generic URI syntax defined in [RFC3986] and obtain the *scheme name*. If the parsing based on the syntax defined in [RFC3986] fails, throw a SyntaxError. If the *scheme name* is not implemented by the browser throw a NotSupportedError. If *scheme name* is turn or turns, and parsing the *url* using the syntax defined in [RFC7065] fails, throw a SyntaxError. If *scheme name* is stun or stuns, and parsing the *url* using the syntax defined in [RFC7064] fails, throw a SyntaxError.~~

2. Let *parsedURL* be the result of parsing *url*.

3. If any of the following conditions apply, then throw a "SyntaxError" DOMException:

   - *parsedURL* is failure
   - *parsedURL*'s scheme is neither "stun", "stuns", "turn", nor "turns"
   - *parsedURL* does not have an opaque path
   - *parsedURL*'s' fragment is non-null
   - *parsedURL*'s' scheme is "stun" or "stuns", and *parsedURL*'s' query is non-null

4. If *parsedURL*'s scheme is not implemented by the user agent, then throw a NotSupportedError.

5. Let *hostAndPortURL* be result of parsing the concatenation of "https://" and *parsedURL*'s path.

6. If *hostAndPortURL* is failure, then throw a "SyntaxError" DOMException.

> ***NOTE***
>
> ***For "stun" and "stuns" schemes, this validates [RFC7064] section 3.1.***
> ***For "turn" and "turns" schemes, this and the steps below validate [RFC7065] section 3.1.***

7. TODO: validate ?transport=udp|tcp (see PR #2996)

8. If ~~*scheme name*~~*parsedURL*'s' scheme is ~~turn~~"turn" ~~or turns~~or "turns", and either of *server*.username or *server*.credential ~~are omitted~~do not exist, then throw an InvalidAccessError.

9. ~~If *scheme name* is turn or turns, and *server*.credentialType is "password", and *server*.credential is not a DOMString, then throw an InvalidAccessError.~~

§ 4.4.2 Interface Definition

The **RTCPeerConnection** interface presented in this section is extended by several partial interfaces throughout this specification. Notably, the RTP Media API section, which adds the APIs to send and receive MediaStreamTrack objects.

WebIDL

```
[Exposed=Window]
interface RTCPeerConnection : EventTarget  {
  constructor(optional RTCConfiguration configuration = {});
  Promise<RTCSessionDescriptionInit> createOffer(optional RTCOfferOptions options = {});
  Promise<RTCSessionDescriptionInit> createAnswer(optional RTCAnswerOptions options =
{});
  Promise<undefined> setLocalDescription(optional RTCLocalSessionDescriptionInit
description = {});
  readonly attribute RTCSessionDescription? localDescription;
  readonly attribute RTCSessionDescription? currentLocalDescription;
  readonly attribute RTCSessionDescription? pendingLocalDescription;
  Promise<undefined> setRemoteDescription(RTCSessionDescriptionInit description);
  readonly attribute RTCSessionDescription? remoteDescription;
  readonly attribute RTCSessionDescription? currentRemoteDescription;
  readonly attribute RTCSessionDescription? pendingRemoteDescription;
  Promise<undefined> addIceCandidate(optional RTCIceCandidateInit candidate = {});
  readonly attribute RTCSignalingState signalingState;
  readonly attribute RTCIceGatheringState iceGatheringState;
  readonly attribute RTCIceConnectionState iceConnectionState;
  readonly attribute RTCPeerConnectionState connectionState;
  readonly attribute boolean? canTrickleIceCandidates;
  undefined restartIce();
  RTCConfiguration getConfiguration();
  undefined setConfiguration(optional RTCConfiguration configuration = {});
  undefined close();
  attribute EventHandler onnegotiationneeded;
  attribute EventHandler onicecandidate;
  attribute EventHandler onicecandidateerror;
  attribute EventHandler onsignalingstatechange;
  attribute EventHandler oniceconnectionstatechange;
  attribute EventHandler onicegatheringstatechange;
  attribute EventHandler onconnectionstatechange;

  // Legacy Interface Extensions
  // Supporting the methods in this section is optional.
  // If these methods are supported
  // they must be implemented as defined
  // in section "Legacy Interface Extensions"
  Promise<undefined> createOffer(RTCSessionDescriptionCallback successCallback,
                                 RTCPeerConnectionErrorCallback failureCallback,
                                 optional RTCOfferOptions options = {});
  Promise<undefined> setLocalDescription(RTCLocalSessionDescriptionInit description,
                                         VoidFunction successCallback,
```

```
                                 RTCPeerConnectionErrorCallback failureCallback);
  Promise<undefined> createAnswer(RTCSessionDescriptionCallback successCallback,
                                  RTCPeerConnectionErrorCallback failureCallback);
  Promise<undefined> setRemoteDescription(RTCSessionDescriptionInit description,
                                          VoidFunction successCallback,
                                          RTCPeerConnectionErrorCallback failureCallback);
  Promise<undefined> addIceCandidate(RTCIceCandidateInit candidate,
                                     VoidFunction successCallback,
                                     RTCPeerConnectionErrorCallback failureCallback);
};
```

§ *Attributes*

localDescription **of type RTCSessionDescription, readonly, nullable**

The localDescription attribute *MUST* return [[PendingLocalDescription]] if it is not null and otherwise it *MUST* return [[CurrentLocalDescription]].

Note that [[CurrentLocalDescription]].sdp and [[PendingLocalDescription]].sdp need not be string-wise identical to the sdp value passed to the corresponding setLocalDescription call (i.e. SDP may be parsed and reformatted, and ICE candidates may be added).

currentLocalDescription **of type RTCSessionDescription, readonly, nullable**

The currentLocalDescription attribute *MUST* return [[CurrentLocalDescription]].

It represents the local description that was successfully negotiated the last time the RTCPeerConnection transitioned into the stable state plus any local candidates that have been generated by the ICE Agent since the offer or answer was created.

pendingLocalDescription **of type RTCSessionDescription, readonly, nullable**

The pendingLocalDescription attribute *MUST* return [[PendingLocalDescription]].

It represents a local description that is in the process of being negotiated plus any local candidates that have been generated by the ICE Agent since the offer or answer was created. If the RTCPeerConnection is in the stable state, the value is null.

remoteDescription **of type RTCSessionDescription, readonly, nullable**

The remoteDescription attribute *MUST* return [[PendingRemoteDescription]] if it is not null and otherwise it *MUST* return [[CurrentRemoteDescription]].

Note that [[CurrentRemoteDescription]].sdp and [[PendingRemoteDescription]].sdp need not be string-wise identical to the sdp value passed to the corresponding setRemoteDescription call (i.e. SDP may be parsed and reformatted, and ICE candidates may be added).

currentRemoteDescription **of type RTCSessionDescription, readonly, nullable**

The currentRemoteDescription attribute *MUST* return [[CurrentRemoteDescription]].

It represents the last remote description that was successfully negotiated the last time the RTCPeerConnection transitioned into the stable state plus any remote candidates that have been supplied via addIceCandidate() since the offer or answer was created.

pendingRemoteDescription **of type RTCSessionDescription, readonly, nullable**

The pendingRemoteDescription attribute *MUST* return [[PendingRemoteDescription]].

It represents a remote description that is in the process of being negotiated, complete with any remote candidates that have been supplied via addIceCandidate() since the offer or answer was created. If the RTCPeerConnection is in the stable state, the value is null.

signalingState **of type RTCSignalingState, readonly**

The signalingState attribute *MUST* return the RTCPeerConnection object's [[SignalingState]].

iceGatheringState **of type RTCIceGatheringState, readonly**

The iceGatheringState attribute *MUST* return the RTCPeerConnection object's [[IceGatheringState]].

iceConnectionState **of type RTCIceConnectionState, readonly**

The iceConnectionState attribute *MUST* return the RTCPeerConnection object's

`[[IceConnectionState]]`.

connectionState **of type `RTCPeerConnectionState`, readonly**
The `connectionState` attribute *MUST* return the `RTCPeerConnection` object's `[[ConnectionState]]`.

canTrickleIceCandidates **of type boolean, readonly, nullable**
The `canTrickleIceCandidates` attribute indicates whether the remote peer is able to accept trickled ICE candidates [RFC8838]. The value is determined based on whether a remote description indicates support for trickle ICE, as defined in [RFC9429] (section 4.1.17.). Prior to the completion of `setRemoteDescription`, this value is `null`.

onnegotiationneeded **of type EventHandler**
The event type of this event handler is `negotiationneeded`.

onicecandidate **of type EventHandler**
The event type of this event handler is `icecandidate`.

onicecandidateerror **of type EventHandler**
The event type of this event handler is `icecandidateerror`.

onsignalingstatechange **of type EventHandler**
The event type of this event handler is `signalingstatechange`.

oniceconnectionstatechange **of type EventHandler**
The event type of this event handler is `iceconnectionstatechange`

onicegatheringstatechange **of type EventHandler**
The event type of this event handler is `icegatheringstatechange`.

onconnectionstatechange **of type EventHandler**
The event type of this event handler is `connectionstatechange`.

§ *Methods*

createOffer
The `createOffer` method generates a blob of SDP that contains an RFC 3264 offer with the supported configurations for the session, including descriptions of the local `MediaStreamTrack`s attached to this `RTCPeerConnection`, the codec/RTP/RTCP capabilities supported by this implementation, and parameters of the ICE agent and the DTLS connection. The *options* parameter may be supplied to provide additional control over the offer generated.

If a system has limited resources (e.g. a finite number of decoders), `createOffer` needs to return an offer that reflects the current state of the system, so that `setLocalDescription` will succeed when it attempts to acquire those resources. The session descriptions *MUST* remain usable by `setLocalDescription` without causing an error until at least the end of the fulfillment callback of the returned promise.

Creating the SDP *MUST* follow the appropriate process for generating an offer described in [RFC9429], except the user agent *MUST* treat a `stopping` transceiver as `stopped` for the purposes of RFC9429 in this case.

As an offer, the generated SDP will contain the full set of codec/RTP/RTCP capabilities supported or preferred by the session (as opposed to an answer, which will include only a specific negotiated subset to use). In the event `createOffer` is called after the session is established, `createOffer` will generate an offer that is compatible with the current session, incorporating any changes that have been made to the session since the last complete offer-answer exchange, such as addition or removal of tracks. If no changes have been made, the offer will include the capabilities of the current local description as well as any additional capabilities that could be negotiated in an updated offer.

The generated SDP will also contain the ICE agent's `usernameFragment`, `password` and ICE options (as defined in [RFC5245], Section 14) and may also contain any local candidates that have been gathered by the agent.

The `certificates` value in *configuration* for the `RTCPeerConnection` provides the certificates configured by the application for the `RTCPeerConnection`. These certificates, along with any default certificates are used to produce a set of certificate fingerprints. These certificate fingerprints are used in the construction of SDP.

The process of generating an SDP exposes a subset of the media capabilities of the underlying system, which provides generally persistent cross-origin information on the device. It thus increases the fingerprinting surface of the application. In privacy-sensitive contexts, browsers can consider mitigations such as generating SDP matching only a common subset of the capabilities.

When the method is called, the user agent *MUST* run the following steps:

1. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.

2. If *connection*.`[[IsClosed]]` is `true`, return a promise rejected with a newly created `InvalidStateError`.

3. Return the result of chaining the result of creating an offer with *connection* to *connection*'s operations chain.

To **create an offer** given *connection* run the following steps:

1. If *connection*.`[[SignalingState]]` is neither "`stable`" nor "`have-local-offer`", return a promise rejected with a newly created `InvalidStateError`.

2. Let *p* be a new promise.

3. In parallel, begin the in-parallel steps to create an offer given *connection* and *p*.

4. Return *p*.

The **in-parallel steps to create an offer** given *connection* and a promise *p* are as follows:

1. If *connection* was not constructed with a set of certificates, and one has not yet been generated, wait for it to be generated.

2. Inspect the **offerer's system state** to determine the currently available resources as necessary for generating the offer, as described in [RFC9429] (section 4.1.8.).

3. If this inspection failed for any reason, reject *p* with a newly created `OperationError` and abort these steps.

4. Queue a task that runs the final steps to create an offer given *connection* and *p*.

The **final steps to create an offer** given *connection* and a promise *p* are as follows:

1. If *connection*.`[[IsClosed]]` is `true`, then abort these steps.

2. If *connection* was modified in such a way that additional inspection of the offerer's system state is necessary, then in parallel begin the in-parallel steps to create an offer again, given *connection* and *p*, and abort these steps.

> ***NOTE***
> ***This may be necessary if, for example, `createOffer` was called when only an audio `RTCRtpTransceiver` was added to connection, but while performing the in-parallel steps to create an offer, a video `RTCRtpTransceiver` was added, requiring additional inspection of video system resources.***

3. Given the information that was obtained from previous inspection, the current state of *connection* and its `RTCRtpTransceiver`s, generate an SDP offer, *sdpString*, as described in [RFC9429] (section 5.2.).

    1. As described in [RFC8843] (Section 7), if bundling is used (see `RTCBundlePolicy`) an offerer tagged m= section must be selected in order to negotiate a BUNDLE group. The user agent *MUST* choose the m= section that corresponds to the first non-stopped transceiver in the set of transceivers as the offerer tagged m= section. This allows the remote endpoint to predict which transceiver is the offerer tagged m= section without having to parse the SDP.

    2. The *codec preferences* of a media description's associated transceiver, *transceiver*, is said to be the value of *transceiver*.`[[PreferredCodecs]]` with the following filtering applied (or said not to be set if *transceiver*.`[[PreferredCodecs]]` is empty):

        1. Let *kind* be *transceiver*'s `[[Receiver]]`'s `[[ReceiverTrack]]`'s `kind`.

3. If *transceiver*.`direction` is "`sendonly`" or "`sendrecv`", exclude any codecs not included in the list of implemented send codecs for *kind*.

4. If *transceiver*.`direction` is "`recvonly`" or "`sendrecv`", exclude any codecs not included in the list of implemented receive codecs for *kind*.

The filtering *MUST NOT* change the order of the codec preferences.

3. If the length of the `[[SendEncodings]]` slot of the `RTCRtpSender` is larger than 1, then for each encoding given in `[[SendEncodings]]` of the `RTCRtpSender`, add an `a=rid send` line to the corresponding media section, and add an `a=simulcast:send` line giving the RIDs in the same order as given in the `encodings` field. No RID restrictions are set.

> ***NOTE***
>
> ***[RFC8853] section 5.2 specifies that the order of RIDs in the a=simulcast line suggests a proposed order of preference. If the browser decides not to transmit all encodings, one should expect it to stop sending the last encoding in the list first.***

4. Let *offer* be a newly created `RTCSessionDescriptionInit` dictionary with its `type` member initialized to the string "`offer`" and its `sdp` member initialized to *sdpString*.

5. Set the `[[LastCreatedOffer]]` internal slot to *sdpString*.

6. Resolve *p* with *offer*.

`createAnswer`

The `createAnswer` method generates an [SDP] answer with the supported configuration for the session that is compatible with the parameters in the remote configuration. Like `createOffer`, the returned blob of SDP contains descriptions of the local `MediaStreamTracks` attached to this `RTCPeerConnection`, the codec/RTP/RTCP options negotiated for this session, and any candidates that have been gathered by the ICE Agent. The *options* parameter may be supplied to provide additional control over the generated answer.

Like `createOffer`, the returned description *SHOULD* reflect the current state of the system. The session descriptions *MUST* remain usable by `setLocalDescription` without causing an error until at least the end of the fulfillment callback of the returned promise.

As an answer, the generated SDP will contain a specific codec/RTP/RTCP configuration that, along with the corresponding offer, specifies how the media plane should be established. The generation of the SDP *MUST* follow the appropriate process for generating an answer described in [RFC9429].

The generated SDP will also contain the ICE agent's `usernameFragment`, `password` and ICE options (as defined in [RFC5245], Section 14) and may also contain any local candidates that have been gathered by the agent.

The `certificates` value in *configuration* for the `RTCPeerConnection` provides the certificates configured by the application for the `RTCPeerConnection`. These certificates, along with any default certificates are used to produce a set of certificate fingerprints. These certificate fingerprints are used in the construction of SDP.

An answer can be marked as provisional, as described in [RFC9429] (section 4.1.10.1.), by setting the `type` to "`pranswer`".

When the method is called, the user agent *MUST* run the following steps:

1. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.

2. If *connection*.`[[IsClosed]]` is `true`, return a promise rejected with a newly created `InvalidStateError`.

3. Return the result of chaining the result of creating an answer with *connection* to *connection*'s operations chain.

To **create an answer** given *connection* run the following steps:

1. If *connection*.`[[SignalingState]]` is neither "`have-remote-offer`" nor "`have-local-pranswer`", return a promise rejected with a newly created `InvalidStateError`.

2. Let *p* be a new promise.

3. In parallel, begin the in-parallel steps to create an answer given *connection* and *p*.

4. Return *p*.

The **in-parallel steps to create an answer** given *connection* and a promise *p* are as follows:

1. If *connection* was not constructed with a set of certificates, and one has not yet been generated, wait for it to be generated.

2. Inspect the **answerer's system state** to determine the currently available resources as necessary for generating the answer, as described in [RFC9429] (section 4.1.9.).

3. If this inspection failed for any reason, reject *p* with a newly created `OperationError` and abort these steps.

4. Queue a task that runs the final steps to create an answer given *p*.

The **final steps to create an answer** given a promise *p* are as follows:

1. If *connection*.`[[IsClosed]]` is `true`, then abort these steps.

2. If *connection* was modified in such a way that additional inspection of the answerer's system state is necessary, then in parallel begin the in-parallel steps to create an answer again given *connection* and *p*, and abort these steps.

> ***NOTE***
> ***This may be necessary if, for example, `createAnswer` was called when an `RTCRtpTransceiver`'s direction was "`recvonly`", but while performing the in-parallel steps to create an answer, the direction was changed to "`sendrecv`", requiring additional inspection of video encoding resources.***

3. Given the information that was obtained from previous inspection and the current state of *connection* and its `RTCRtpTransceiver`s, generate an SDP answer, *sdpString*, as described in [RFC9429] (section 5.3.).

CANDIDATE CORRECTION 26: Prune createAnswer()'s encodings and SendEncodings in sLD(answer). (PR #2801)

CANDIDATE CORRECTION 27: Ignore comma-separated rid alternatives. (PR #2813)

◉ Show Current and Future ○ Show Current ○ Show Future

1. The *codec preferences* of an m= section's ~~associated transceiver~~ associated transceiver, *transceiver*, is said to be the value of ~~the `RTCRtpTransceiver`~~*transceiver*.`[[PreferredCodecs]]` with the following filtering applied (or said not to be set ~~if~~ if *transceiver*.`[[PreferredCodecs]]` is empty):

   1. ~~If the `direction` is "`sendrecv`", exclude any codecs not included in the intersection of `RTCRtpSender.getCapabilities(kind).codecs` and `RTCRtpReceiver.getCapabilities(kind).codecs`.~~

   2. Let *kind* be *transceiver*'s `[[Receiver]]`'s `[[ReceiverTrack]]`'s `kind`.

   3. ~~If the~~ If *transceiver*.`direction` ~~is~~ is "`sendonly`" or "`sendrecv`", exclude any codecs not included ~~in `RTCRtpSender.getCapabilities(kind)`.~~ in the list of implemented send codecs for *kind*.

   4. ~~If the~~ If *transceiver*.`direction` ~~is~~ is "`recvonly`" or "`sendrecv`", exclude any codecs not included ~~in `RTCRtpReceiver.getCapabilities(kind)`.~~ in the list of implemented receive codecs for *kind*.

   The filtering *MUST NOT* change the order of the codec preferences.

2. ~~If the length of the [[SendEncodings]] slot of the `RTCRtpSender` is larger than 1, then for each encoding given in [[SendEncodings]] of the `RTCRtpSender`, add an `a=rid send` line to~~

~~the corresponding media section, and add an a=simulcast:send line giving the RIDs in the same order as given in the encodings field. No RID restrictions are set.~~

3. If this is an answer to an offer to receive simulcast, then for each media section requesting to receive simulcast, run the following steps:

   1. If the a=simulcast attribute contains comma-separated alternatives for RIDs, remove all but the first ones.

   2. If there are any identically named RIDs in the a=simulcast attribute, remove all but the first one. No RID restrictions are set.

   3. Exclude from the media section in the answer any RID not found in the corresponding transceiver's `[[Sender]]`.`[[SendEncodings]]`.

   NOTE
   When a `setRemoteDescription(offer)` establishes a sender's proposed envelope, the sender's `[[SendEncodings]]` is updated in "`have-remote-offer`", exposing it to rollback. However, once a simulcast envelope has been established for the sender, subsequent pruning of the sender's `[[SendEncodings]]` happen when this answer is set with `setLocalDescription`.

4. Let *answer* be a newly created `RTCSessionDescriptionInit` dictionary with its `type` member initialized to the string "`answer`" and its `sdp` member initialized to *sdpString*.

5. Set the `[[LastCreatedAnswer]]` internal slot to *sdpString*.

6. Resolve *p* with *answer*.

`setLocalDescription`

The `setLocalDescription` method instructs the `RTCPeerConnection` to apply the supplied `RTCLocalSessionDescriptionInit` as the local description.

This API changes the local media state. In order to successfully handle scenarios where the application wants to offer to change from one media format to a different, incompatible format, the `RTCPeerConnection` *MUST* be able to simultaneously support use of both the current and pending local descriptions (e.g. support codecs that exist in both descriptions) until a final answer is received, at which point the `RTCPeerConnection` can fully adopt the pending local description, or rollback to the current description if the remote side rejected the change.

Passing in a description is optional. If left out, then `setLocalDescription` will implicitly create an offer or create an answer, as needed. As noted in [RFC9429] (section 5.4.), if a description with SDP is passed in, that SDP is not allowed to have changed from when it was returned from either `createOffer` or `createAnswer`.

When the method is invoked, the user agent *MUST* run the following steps:

1. Let *description* be the method's first argument.

2. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.

3. Let *sdp* be *description*.`sdp`.

4. Return the result of chaining the following steps to *connection*'s operations chain:

   1. Let *type* be *description*.`type` if present, or "`offer`" if not present and *connection*.`[[SignalingState]]` is either "`stable`", "`have-local-offer`", or "`have-remote-pranswer`"; otherwise "`answer`".

   2. If *type* is "`offer`", and *sdp* is not the empty string and not equal to *connection*.`[[LastCreatedOffer]]`, then return a promise rejected with a newly created `InvalidModificationError` and abort these steps.

   3. If *type* is "`answer`" or "`pranswer`", and *sdp* is not the empty string and not equal to *connection*.`[[LastCreatedAnswer]]`, then return a promise rejected with a newly created `InvalidModificationError` and abort these steps.

   4. If *sdp* is the empty string, and *type* is "`offer`", then run the following sub steps:

1. Set *sdp* to the value of *connection*.`[[LastCreatedOffer]]`.

2. If *sdp* is the empty string, or if it no longer accurately represents the offerer's system state of *connection*, then let *p* be the result of creating an offer with *connection*, and return the result of reacting to *p* with a fulfillment step that sets the local session description indicated by its first argument.

5. If *sdp* is the empty string, and *type* is "`answer`" or "`pranswer`", then run the following sub steps:

1. Set *sdp* to the value of *connection*.`[[LastCreatedAnswer]]`.

2. If *sdp* is the empty string, or if it no longer accurately represents the answerer's system state of *connection*, then let *p* be the result of creating an answer with *connection*, and return the result of reacting to *p* with the following fulfillment steps:

1. Let *answer* be the first argument to these fulfillment steps.

2. Return the result of setting the local session description indicated by `{type, answer.sdp}`.

6. Return the result of setting the local session description indicated by `{type, sdp}`.

NOTE

As noted in [RFC9429] (section 5.9.), calling this method may trigger the ICE candidate gathering process by the ICE Agent.

`setRemoteDescription`

The `setRemoteDescription` method instructs the `RTCPeerConnection` to apply the supplied `RTCSessionDescriptionInit` as the remote offer or answer. This API changes the local media state.

When the method is invoked, the user agent *MUST* run the following steps:

1. Let *description* be the method's first argument.

2. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.

3. Return the result of chaining the following steps to *connection*'s operations chain:

1. If *description*.`type` is "`offer`" and is invalid for the current *connection*.`[[SignalingState]]` as described in [RFC9429] (section 5.5. and section 5.6.), then run the following sub steps:

1. Let *p* be the result of setting the local session description indicated by `{type: "rollback"}`.

2. Return the result of reacting to *p* with a fulfillment step that sets the remote session description *description*, and abort these steps.

2. Return the result of setting the remote session description *description*.

`addIceCandidate`

The `addIceCandidate` method provides a remote candidate to the ICE Agent. This method can also be used to indicate the end of remote candidates when called with an empty string for the `candidate` member. The only members of the argument used by this method are `candidate`, `sdpMid`, `sdpMLineIndex`, and `usernameFragment`; the rest are ignored. When the method is invoked, the user agent *MUST* run the following steps:

1. Let *candidate* be the method's argument.

2. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.

3. If *candidate*.`candidate` is not an empty string and both *candidate*.`sdpMid` and *candidate*.`sdpMLineIndex` are `null`, return a promise rejected with a newly created `TypeError`.

4. Return the result of chaining the following steps to *connection*'s operations chain:

1. If `remoteDescription` is `null` return a promise rejected with a newly created `InvalidStateError`.

2. If *candidate*.`sdpMid` is not `null`, run the following steps:

   1. If *candidate*.`sdpMid` is not equal to the mid of any media description in `remoteDescription`, return a promise rejected with a newly created `OperationError`.

3. Else, if *candidate*.`sdpMLineIndex` is not `null`, run the following steps:

   1. If *candidate*.`sdpMLineIndex` is equal to or larger than the number of media descriptions in `remoteDescription`, return a promise rejected with a newly created `OperationError`.

4. If either *candidate*.`sdpMid` or *candidate*.`sdpMLineIndex` indicate a media description in `remoteDescription` whose associated transceiver is stopped, return a promise resolved with `undefined`.

5. If *candidate*.`usernameFragment` is not `null`, and is not equal to any username fragment present in the corresponding media description of an applied remote description, return a promise rejected with a newly created `OperationError`.

6. Let *p* be a new promise.

7. In parallel, if the candidate is not administratively prohibited, add the ICE candidate *candidate* as described in [RFC9429] (section 4.1.19.). Use *candidate*.`usernameFragment` to identify the ICE generation; if `usernameFragment` is `null`, process the *candidate* for the most recent ICE generation.

   If *candidate*.`candidate` is an empty string, process *candidate* as an end-of-candidates indication for the corresponding media description and ICE candidate generation. If both *candidate*.`sdpMid` and *candidate*.`sdpMLineIndex` are `null`, then this end-of-candidates indication applies to all media descriptions.

   1. If *candidate* could not be successfully added the user agent *MUST* queue a task that runs the following steps:

      1. If *connection*.`[[IsClosed]]` is `true`, then abort these steps.

      2. Reject *p* with a newly created `OperationError` and abort these steps.

   2. If *candidate* is applied successfully, or if the candidate was administratively prohibited the user agent *MUST* queue a task that runs the following steps:

      1. If *connection*.`[[IsClosed]]` is `true`, then abort these steps.

      2. If *connection*.`[[PendingRemoteDescription]]` is not `null`, and represents the ICE generation for which *candidate* was processed, add *candidate* to *connection*.`[[PendingRemoteDescription]]`.sdp.

      3. If *connection*.`[[CurrentRemoteDescription]]` is not `null`, and represents the ICE generation for which *candidate* was processed, add *candidate* to *connection*.`[[CurrentRemoteDescription]]`.sdp.

      4. Resolve *p* with `undefined`.

8. Return *p*.

A candidate is **administratively prohibited** if the UA has decided not to allow connection attempts to this address.

For privacy reasons, there is no indication to the developer about whether or not an address/port is blocked; it behaves exactly as if there was no response from the address.

The UA *MUST* prohibit connections to addresses on the [Fetch] block bad port list, and *MAY* choose to prohibit connections to other addresses.

If the `iceTransportPolicy` member of the `RTCConfiguration` is `relay`, candidates requiring external resolution, such as mDNS candidates and DNS candidates, *MUST* be prohibited.

> ***NOTE***
>
> ***Due to WebIDL processing, `addIceCandidate`(`null`) is interpreted as a call with the default dictionary present, which, in the above algorithm, indicates end-of-candidates for all media descriptions and ICE candidate generation. This is by design for legacy reasons.***

`restartIce`

The `restartIce` method tells the `RTCPeerConnection` that ICE should be restarted. Subsequent calls to `createOffer` will create descriptions that will restart ICE, as described in section 9.1.1.1 of [RFC5245].

When this method is invoked, the user agent *MUST* run the following steps:

1. Let *connection* be the `RTCPeerConnection` on which the method was invoked.
2. Empty *connection*.`[[LocalIceCredentialsToReplace]]`, and populate it with all ICE credentials (ice-ufrag and ice-pwd as defined in section 15.4 of [RFC5245]) found in *connection*.`[[CurrentLocalDescription]]`, as well as all ICE credentials found in *connection*.`[[PendingLocalDescription]]`.
3. Update the negotiation-needed flag for *connection*.

`getConfiguration`

Returns an `RTCConfiguration` object representing the current configuration of this `RTCPeerConnection` object.

When this method is called, the user agent *MUST* return the `RTCConfiguration` object stored in the `[[Configuration]]` internal slot.

`setConfiguration`

The `setConfiguration` method updates the configuration of this `RTCPeerConnection` object. This includes changing the configuration of the ICE Agent. As noted in [RFC9429] (section 3.5.1.), when the ICE configuration changes in a way that requires a new gathering phase, an ICE restart is required.

When the `setConfiguration` method is invoked, the user agent *MUST* run the following steps:

1. Let *connection* be the `RTCPeerConnection` on which the method was invoked.
2. If *connection*.`[[IsClosed]]` is `true`, throw an `InvalidStateError`.
3. Set the configuration specified by *configuration*.

`close`

When the `close` method is invoked, the user agent *MUST* run the following steps:

1. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.
2. close the connection with *connection* and the value `false`.

The **close the connection** algorithm given a *connection* and a *disappear* boolean, is as follows:

1. If *connection*.`[[IsClosed]]` is `true`, abort these steps.
2. Set *connection*.`[[IsClosed]]` to `true`.
3. Set *connection*.`[[SignalingState]]` to "`closed`". This does not fire any event.
4. Let *transceivers* be the result of executing the `CollectTransceivers` algorithm. For every `RTCRtpTransceiver` *transceiver* in *transceivers*, run the following steps:
    1. If *transceiver*.`[[Stopped]]` is `true`, abort these sub steps.
    2. Stop the RTCRtpTransceiver with *transceiver* and *disappear*.
5. Set the `[[ReadyState]]` slot of each of *connection*'s `RTCDataChannel`s to "`closed`".

> ***NOTE***
>
> ***The `RTCDataChannel`s will be closed abruptly and the closing procedure will not be invoked.***

6. If *connection*.`[[SctpTransport]]` is not `null`, tear down the underlying SCTP association by sending an SCTP ABORT chunk and set the `[[SctpTransportState]]` to "`closed`".

7. Set the `[[DtlsTransportState]]` slot of each of *connection*'s `RTCDtlsTransport`s to "`closed`".

8. Destroy *connection*'s ICE Agent, abruptly ending any active ICE processing and releasing any relevant resources (e.g. TURN permissions).

9. Set the `[[IceTransportState]]` slot of each of *connection*'s `RTCIceTransport`s to "`closed`".

10. Set *connection*.`[[IceConnectionState]]` to "`closed`". This does not fire any event.

11. Set *connection*.`[[ConnectionState]]` to "`closed`". This does not fire any event.

§ 4.4.3 Legacy Interface Extensions

> ***NOTE***
> ***The IDL definition of these methods are documented in the main definition of the `RTCPeerConnection` interface since overloaded functions are not allowed to be defined in partial interfaces.***

Supporting the methods in this section is optional. However, if these methods are supported it is mandatory to implement according to what is specified here.

> ***NOTE***
> ***The `addStream` method that used to exist on `RTCPeerConnection` is easy to polyfill as:***

```
RTCPeerConnection.prototype.addStream = function(stream) {
  stream.getTracks().forEach((track) => this.addTrack(track, stream));
};
```

§ *4.4.3.1 Method extensions*

§ METHODS

`createOffer`

When the `createOffer` method is called, the user agent *MUST* run the following steps:

1. Let *successCallback* be the method's first argument.

2. Let *failureCallback* be the callback indicated by the method's second argument.

3. Let *options* be the callback indicated by the method's third argument.

4. Run the steps specified by `RTCPeerConnection`'s `createOffer()` method with *options* as the sole argument, and let *p* be the resulting promise.

5. Upon fulfillment of *p* with value *offer*, invoke *successCallback* with *offer* as the argument.

6. Upon rejection of *p* with reason *r*, invoke *failureCallback* with *r* as the argument.

7. Return a promise resolved with `undefined`.

`setLocalDescription`

When the `setLocalDescription` method is called, the user agent *MUST* run the following steps:

1. Let *description* be the method's first argument.

2. Let *successCallback* be the callback indicated by the method's second argument.

3. Let *failureCallback* be the callback indicated by the method's third argument.

4. Run the steps specified by `RTCPeerConnection`'s `setLocalDescription` method with *description* as the sole argument, and let *p* be the resulting promise.

5. Upon fulfillment of *p*, invoke *successCallback* with `undefined` as the argument.

6. Upon rejection of *p* with reason *r*, invoke *failureCallback* with *r* as the argument.

7. Return a promise resolved with `undefined`.

`createAnswer`

> ***NOTE***
> ***The legacy `createAnswer` method does not take an `RTCAnswerOptions` parameter, since no known legacy `createAnswer` implementation ever supported it.***

When the `createAnswer` method is called, the user agent *MUST* run the following steps:

1. Let *successCallback* be the method's first argument.

2. Let *failureCallback* be the callback indicated by the method's second argument.

3. Run the steps specified by `RTCPeerConnection`'s `createAnswer()` method with no arguments, and let *p* be the resulting promise.

4. Upon fulfillment of *p* with value *answer*, invoke *successCallback* with *answer* as the argument.

5. Upon rejection of *p* with reason *r*, invoke *failureCallback* with *r* as the argument.

6. Return a promise resolved with `undefined`.

`setRemoteDescription`

When the `setRemoteDescription` method is called, the user agent *MUST* run the following steps:

1. Let *description* be the method's first argument.

2. Let *successCallback* be the callback indicated by the method's second argument.

3. Let *failureCallback* be the callback indicated by the method's third argument.

4. Run the steps specified by `RTCPeerConnection`'s `setRemoteDescription` method with *description* as the sole argument, and let *p* be the resulting promise.

5. Upon fulfillment of *p*, invoke *successCallback* with `undefined` as the argument.

6. Upon rejection of *p* with reason *r*, invoke *failureCallback* with *r* as the argument.

7. Return a promise resolved with `undefined`.

`addIceCandidate`

When the `addIceCandidate` method is called, the user agent *MUST* run the following steps:

1. Let *candidate* be the method's first argument.

2. Let *successCallback* be the callback indicated by the method's second argument.

3. Let *failureCallback* be the callback indicated by the method's third argument.

4. Run the steps specified by `RTCPeerConnection`'s `addIceCandidate()` method with *candidate* as the sole argument, and let *p* be the resulting promise.

5. Upon fulfillment of *p*, invoke *successCallback* with `undefined` as the argument.

6. Upon rejection of *p* with reason *r*, invoke *failureCallback* with *r* as the argument.

7. Return a promise resolved with `undefined`.

§ Callback Definitions

These callbacks are only used on the legacy APIs.

§ **`RTCPeerConnectionErrorCallback`**

WebIDL

```
callback RTCPeerConnectionErrorCallback = undefined (DOMException error);
```

§ CALLBACK `RTCPeerConnectionErrorCallback` PARAMETERS

`error` of type `DOMException`
An error object encapsulating information about what went wrong.

§ **`RTCSessionDescriptionCallback`**

WebIDL

```
callback RTCSessionDescriptionCallback = undefined (RTCSessionDescriptionInit
description);
```

§ CALLBACK `RTCSessionDescriptionCallback` PARAMETERS

`description` of type `RTCSessionDescriptionInit`
The object containing the SDP [SDP].

§ *4.4.3.2 Legacy configuration extensions*

This section describes a set of legacy extensions that may be used to influence how an offer is created, in addition to the media added to the `RTCPeerConnection`. Developers are encouraged to use the `RTCRtpTransceiver` API instead.

When `createOffer` is called with any of the legacy options specified in this section, run the followings steps instead of the regular `createOffer` steps:

1. Let *options* be the methods first argument.
2. Let *connection* be the current `RTCPeerConnection` object.
3. For each `offerToReceive<Kind>` member in *options* with kind, *kind*, run the following steps:
    1. If the value of the dictionary member is false,
        1. For each non-stopped "`sendrecv`" transceiver of transceiver kind *kind*, set *transceiver*.`[[Direction]]` to "`sendonly`".
        2. For each non-stopped "`recvonly`" transceiver of transceiver kind *kind*, set *transceiver*.`[[Direction]]` to "`inactive`".

    Continue with the next option, if any.
    2. If *connection* has any non-stopped "`sendrecv`" or "`recvonly`" transceivers of transceiver kind *kind*, continue with the next option, if any.

   3. Let *transceiver* be the result of invoking the equivalent of *connection*.`addTransceiver`(*kind*), except that this operation *MUST NOT* update the negotiation-needed flag.

   4. If *transceiver* is unset because the previous operation threw an error, abort these steps.

   5. Set *transceiver*.`[[Direction]]` to "`recvonly`".

4. Run the steps specified by `createOffer` to create the offer.

WebIDL

```
partial dictionary RTCOfferOptions {
  boolean offerToReceiveAudio;
  boolean offerToReceiveVideo;
};
```

§ ATTRIBUTES

***`offerToReceiveAudio`* of type boolean**
This setting provides additional control over the directionality of audio. For example, it can be used to ensure that audio can be received, regardless if audio is sent or not.

***`offerToReceiveVideo`* of type boolean**
This setting provides additional control over the directionality of video. For example, it can be used to ensure that video can be received, regardless if video is sent or not.

§ 4.4.4 Garbage collection

An `RTCPeerConnection` object *MUST* not be garbage collected as long as any event can cause an event handler to be triggered on the object. When the object's `[[IsClosed]]` internal slot is `true`, no such event handler can be triggered and it is therefore safe to garbage collect the object.

All `RTCDataChannel` and `MediaStreamTrack` objects that are connected to an `RTCPeerConnection` have a strong reference to the `RTCPeerConnection` object.

§ 4.5 Error Handling

§ 4.5.1 General Principles

All methods that return promises are governed by the standard error handling rules of promises. Methods that do not return promises may throw exceptions to indicate errors.

§ 4.6 Session Description Model

§ 4.6.1 `RTCSdpType`

The `RTCSdpType` enum describes the type of an `RTCSessionDescriptionInit`, `RTCLocalSessionDescriptionInit`, or `RTCSessionDescription` instance.

WebIDL

```
enum RTCSdpType {
```

```
  "offer",
  "pranswer",
  "answer",
  "rollback"
};
```

RTCSdpType Enumeration description

Enum value	Description
`offer`	An `RTCSdpType` of "`offer`" indicates that a description *MUST* be treated as an [SDP] offer.
`pranswer`	An `RTCSdpType` of "`pranswer`" indicates that a description *MUST* be treated as an [SDP] answer, but not a final answer. A description used as an SDP pranswer may be applied as a response to an SDP offer, or an update to a previously sent SDP pranswer.
`answer`	An `RTCSdpType` of "`answer`" indicates that a description *MUST* be treated as an [SDP] final answer, and the offer-answer exchange *MUST* be considered complete. A description used as an SDP answer may be applied as a response to an SDP offer or as an update to a previously sent SDP pranswer.
`rollback`	An `RTCSdpType` of "`rollback`" indicates that a description *MUST* be treated as canceling the current SDP negotiation and moving the SDP [SDP] offer back to what it was in the previous stable state. Note the local or remote SDP descriptions in the previous stable state could be `null` if there has not yet been a successful offer-answer negotiation. An "`answer`" or "`pranswer`" cannot be rolled back.

§ 4.6.2 `RTCSessionDescription` Class

The `RTCSessionDescription` class is used by `RTCPeerConnection` to expose local and remote session descriptions.

WebIDL

```
[Exposed=Window]
interface RTCSessionDescription {
  constructor(RTCSessionDescriptionInit descriptionInitDict);
  readonly attribute RTCSdpType type;
  readonly attribute DOMString sdp;
  [Default] RTCSessionDescriptionInit toJSON();
};
```

§ *Constructors*

constructor()

The `RTCSessionDescription()` constructor takes a dictionary argument, *description*, whose content is used to initialize the new `RTCSessionDescription` object. This constructor is deprecated; it exists for legacy compatibility reasons only.

§ *Attributes*

type **of type `RTCSdpType`, readonly**

The type of this session description.

sdp **of type DOMString, readonly, defaulting to ""**

The string representation of the SDP [SDP].

§ *Methods*

toJSON()
When called, run [WEBIDL]'s default toJSON steps.

WebIDL

```
dictionary RTCSessionDescriptionInit {
  required RTCSdpType type;
  DOMString sdp = "";
};
```

§ *Dictionary* `RTCSessionDescriptionInit` *Members*

***type* of type `RTCSdpType`, required**
The type of this session description.

***sdp* of type DOMString**
The string representation of the SDP [SDP]; if `type` is "`rollback`", this member is unused.

WebIDL

```
dictionary RTCLocalSessionDescriptionInit {
  RTCSdpType type;
  DOMString sdp = "";
};
```

§ *Dictionary* `RTCLocalSessionDescriptionInit` *Members*

***type* of type `RTCSdpType`**
The type of this description. If not present, then `setLocalDescription` will infer the type based on the `RTCPeerConnection`'s `[[SignalingState]]`.

***sdp* of type DOMString**
The string representation of the SDP [SDP]; if `type` is "`rollback`", this member is unused.

§ 4.7 Session Negotiation Model

Many changes to state of an `RTCPeerConnection` will require communication with the remote side via the signaling channel, in order to have the desired effect. The app can be kept informed as to when it needs to do signaling, by listening to the `negotiationneeded` event. This event is fired according to the state of the connection's **negotiation-needed flag**, represented by a `[[NegotiationNeeded]]` internal slot.

§ 4.7.1 Setting Negotiation-Needed

This section is non-normative.

If an operation is performed on an `RTCPeerConnection` that requires signaling, the connection will be marked as needing negotiation. Examples of such operations include adding or stopping an `RTCRtpTransceiver`, or adding the first `RTCDataChannel`.

Internal changes within the implementation can also result in the connection being marked as needing negotiation.

Note that the exact procedures for updating the negotiation-needed flag are specified below.

§ 4.7.2 Clearing Negotiation-Needed

This section is non-normative.

The negotiation-needed flag is cleared when a session description of type "`answer`" is set successfully, and the supplied description matches the state of the `RTCRtpTransceiver`s and `RTCDataChannel`s that currently exist on the `RTCPeerConnection`. Specifically, this means that all non-`stopped` transceivers have an associated section in the local description with matching properties, and, if any data channels have been created, a data section exists in the local description.

Note that the exact procedures for updating the negotiation-needed flag are specified below.

§ 4.7.3 Updating the Negotiation-Needed flag

The process below occurs where referenced elsewhere in this document. It also may occur as a result of internal changes within the implementation that affect negotiation. If such changes occur, the user agent *MUST* update the negotiation-needed flag.

To **update the negotiation-needed flag** for *connection*, run the following steps:

1. If the length of *connection*.`[[Operations]]` is not `0`, then set *connection*.`[[UpdateNegotiationNeededFlagOnEmptyChain]]` to `true`, and abort these steps.

2. Queue a task to run the following steps:

    1. If *connection*.`[[IsClosed]]` is `true`, abort these steps.

    2. If the length of *connection*.`[[Operations]]` is not `0`, then set *connection*.`[[UpdateNegotiationNeededFlagOnEmptyChain]]` to `true`, and abort these steps.

    3. If *connection*.`[[SignalingState]]` is not "`stable`", abort these steps.

        > ***NOTE***
        >
        > ***The negotiation-needed flag will be updated once the state transitions to "`stable`", as part of the steps for setting a session description.***

    4. If the result of checking if negotiation is needed is `false`, **clear the negotiation-needed flag** by setting *connection*.`[[NegotiationNeeded]]` to `false`, and abort these steps.

    5. If *connection*.`[[NegotiationNeeded]]` is already `true`, abort these steps.

    6. Set *connection*.`[[NegotiationNeeded]]` to `true`.

    7. Fire an event named `negotiationneeded` at *connection*.

> ***NOTE***
> ***The task queueing prevents `negotiationneeded` from firing prematurely, in the common situation where multiple modifications to connection are being made at once.***
>
> ***Additionally, we avoid racing with negotiation methods by only firing `negotiationneeded` when the operations chain is empty.***

To **check if negotiation is needed** for *connection*, perform the following checks:

1. If any implementation-specific negotiation is required, as described at the start of this section, return `true`.

2. If *connection*.`[[LocalIceCredentialsToReplace]]` is not empty, return `true`.

3. Let *description* be *connection*.`[[CurrentLocalDescription]]`.

4. If *connection* has created any `RTCDataChannel`s, and no m= section in *description* has been negotiated yet for data, return `true`.

5. For each *transceiver* in *connection*'s set of transceivers, perform the following checks:

    1. If *transceiver*.`[[Stopping]]` is `true` and *transceiver*.`[[Stopped]]` is `false`, return `true`.

    2. If *transceiver* isn't `stopped` and isn't yet associated with an m= section in *description*, return `true`.

    3. If *transceiver* isn't `stopped` and is associated with an m= section in *description* then perform the following checks:

        1. If *transceiver*.`[[Direction]]` is "`sendrecv`" or "`sendonly`", and the associated m= section in *description* either doesn't contain a single `a=msid` line, or the number of MSIDs from the `a=msid` lines in this `m=` section, or the MSID values themselves, differ from what is in *transceiver*.sender.`[[AssociatedMediaStreamIds]]`, return `true`.

        2. If *description* is of type "`offer`", and the direction of the associated m= section in neither *connection*.`[[CurrentLocalDescription]]` nor *connection*.`[[CurrentRemoteDescription]]` matches *transceiver*.`[[Direction]]`, return `true`. In this step, when the direction is compared with a direction found in `[[CurrentRemoteDescription]]`, the description's direction must be reversed to represent the peer's point of view.

        3. If *description* is of type "`answer`", and the direction of the associated m= section in the *description* does not match *transceiver*.`[[Direction]]` intersected with the offered direction (as described in [RFC9429] (section 5.3.1.)), return `true`.

    4. If *transceiver* is `stopped` and is associated with an m= section, but the associated m= section is not yet rejected in *connection*.`[[CurrentLocalDescription]]` or *connection*.`[[CurrentRemoteDescription]]`, return `true`.

6. If all the preceding checks were performed and `true` was not returned, nothing remains to be negotiated; return `false`.

§ 4.8 Interfaces for Interactive Connectivity Establishment

§ 4.8.1 `RTCIceCandidate` Interface

This interface describes an ICE candidate, described in [RFC5245] Section 2. Other than `candidate`, `sdpMid`, `sdpMLineIndex`, and `usernameFragment`, the remaining attributes are derived from parsing the `candidate` member in *candidateInitDict*, if it is well formed.

CANDIDATE ADDITION 16: Add RTCIceCandidate.relayProtocol (PR #2763)

CANDIDATE ADDITION 23: Add RTCIceCandidate.url (PR #2773)

◉ Show Current and Future ○ Show Current ○ Show Future

```
[Exposed=Window]
interface RTCIceCandidate {
  constructor(optional RTCIceCandidateInit candidateInitDict = {});
  readonly attribute DOMString candidate;
  readonly attribute DOMString? sdpMid;
  readonly attribute unsigned short? sdpMLineIndex;
  readonly attribute DOMString? foundation;
  readonly attribute RTCIceComponent? component;
  readonly attribute unsigned long? priority;
```

```
  readonly attribute DOMString? address;
  readonly attribute RTCIceProtocol? protocol;
  readonly attribute unsigned short? port;
  readonly attribute RTCIceCandidateType? type;
  readonly attribute RTCIceTcpCandidateType? tcpType;
  readonly attribute DOMString? relatedAddress;
  readonly attribute unsigned short? relatedPort;
  readonly attribute DOMString? usernameFragment;
  readonly attribute RTCIceServerTransportProtocol? relayProtocol;
  readonly attribute DOMString? url;
  RTCIceCandidateInit toJSON();
};
```

§ *Constructor*

constructor()

The **RTCIceCandidate()** constructor takes a dictionary argument, *candidateInitDict*, whose content is used to initialize the new `RTCIceCandidate` object.

When invoked, run the following steps:

1. If both the `sdpMid` and `sdpMLineIndex` members of *candidateInitDict* are `null`, throw a `TypeError`.
2. Return the result of creating an RTCIceCandidate with *candidateInitDict*.

To **create an RTCIceCandidate** with a *candidateInitDict* dictionary, run the following steps:

1. Let *iceCandidate* be a newly created `RTCIceCandidate` object.
2. Create internal slots for the following attributes of *iceCandidate*, initilized to `null`: `foundation`, `component`, `priority`, `address`, `protocol`, `port`, `type`, `tcpType`, `relatedAddress`, and `relatedPort`.
3. Create internal slots for the following attributes of *iceCandidate*, initilized to their namesakes in *candidateInitDict*: `candidate`, `sdpMid`, `sdpMLineIndex`, `usernameFragment`.
4. Let *candidate* be the `candidate` dictionary member of *candidateInitDict*. If *candidate* is not an empty string, run the following steps:
    1. Parse *candidate* using the `candidate-attribute` grammar.
    2. If parsing of `candidate-attribute` has failed, abort these steps.
    3. If any field in the parse result represents an invalid value for the corresponding attribute in *iceCandidate*, abort these steps.
    4. Set the corresponding internal slots in *iceCandidate* to the field values of the parsed result.
5. Return *iceCandidate*.

> ***NOTE***
>
> ***The constructor for `RTCIceCandidate` only does basic parsing and type checking for the dictionary members in candidateInitDict. Detailed validation on the well-formedness of `candidate`, `sdpMid`, `sdpMLineIndex`, `usernameFragment` with the corresponding session description is done when passing the `RTCIceCandidate` object to `addIceCandidate()`.***
>
> ***To maintain backward compatibility, any error on parsing the candidate attribute is ignored. In such case, the `candidate` attribute holds the raw `candidate` string given in candidateInitDict, but derivative attributes such as `foundation`, `priority`, etc are set to `null`.***

§ *Attributes*

Most attributes below are defined in section 15.1 of [RFC5245].

`candidate` **of type DOMString, readonly**
: This carries the `candidate-attribute` as defined in section 15.1 of [RFC5245]. If this `RTCIceCandidate` represents an end-of-candidates indication or a peer reflexive remote candidate, `candidate` is an empty string.

`sdpMid` **of type DOMString, readonly, nullable**
: If not `null`, this contains the **media stream "identification-tag"** defined in [RFC5888] for the media component this candidate is associated with.

`sdpMLineIndex` **of type unsigned short, readonly, nullable**
: If not `null`, this indicates the index (starting at zero) of the media description in the SDP this candidate is associated with.

`foundation` **of type DOMString, readonly, nullable**
: A unique identifier that allows ICE to correlate candidates that appear on multiple `RTCIceTransport`s.

`component` **of type `RTCIceComponent`, readonly, nullable**
: The assigned network component of the candidate ("`rtp`" or "`rtcp`"). This corresponds to the `component-id` field in `candidate-attribute`, decoded to the string representation as defined in `RTCIceComponent`.

`priority` **of type unsigned long, readonly, nullable**
: The assigned priority of the candidate.

`address` **of type DOMString, readonly, nullable**
: The address of the candidate, allowing for IPv4 addresses, IPv6 addresses, and fully qualified domain names (FQDNs). This corresponds to the `connection-address` field in `candidate-attribute`.

Remote candidates may be exposed, for instance via `[[SelectedCandidatePair]].remote`. By default, the user agent *MUST* leave the `address` attribute as `null` for any exposed remote candidate. Once a `RTCPeerConnection` instance learns on an address by the web application using `addIceCandidate`, the user agent can expose the `address` attribute value in any `RTCIceCandidate` of the `RTCPeerConnection` instance representing a remote candidate with that newly learnt address.

> ***NOTE***
> ***The addresses exposed in candidates gathered via ICE and made visibile to the application in `RTCIceCandidate` instances can reveal more information about the device and the user (e.g. location, local network topology) than the user might have expected in a non-WebRTC enabled browser.***
>
> ***These addresses are always exposed to the application, and potentially exposed to the communicating party, and can be exposed without any specific user consent (e.g. for peer connections used with data channels, or to receive media only).***
>
> ***These addresses can also be used as temporary or persistent cross-origin states, and thus contribute to the fingerprinting surface of the device.***
>
> ***Applications can avoid exposing addresses to the communicating party, either temporarily or permanently, by forcing the ICE Agent to report only relay candidates via the `iceTransportPolicy` member of `RTCConfiguration`.***
>
> ***To limit the addresses exposed to the application itself, browsers can offer their users different policies regarding sharing local addresses, as defined in [RFC8828].***

`protocol` **of type `RTCIceProtocol`, readonly, nullable**
: The protocol of the candidate ("`udp`"/"`tcp`"). This corresponds to the `transport` field in `candidate-attribute`.

`port` **of type unsigned short, readonly, nullable**
: The port of the candidate.

`type` **of type `RTCIceCandidateType`, readonly, nullable**
: The type of the candidate. This corresponds to the `candidate-types` field in `candidate-attribute`.

`tcpType` **of type `RTCIceTcpCandidateType`, readonly, nullable**
: If `protocol` is "`tcp`", `tcpType` represents the type of TCP candidate. Otherwise, `tcpType` is `null`. This corresponds to the `tcp-type` field in `candidate-attribute`.

`relatedAddress` **of type DOMString, readonly, nullable**
: For a candidate that is derived from another, such as a relay or reflexive candidate, the `relatedAddress` is the IP address of the candidate that it is derived from. For host candidates, the

`relatedAddress` is `null`. This corresponds to the `rel-address` field in `candidate-attribute`.

***`relatedPort`* of type unsigned short, readonly, nullable**
For a candidate that is derived from another, such as a relay or reflexive candidate, the `relatedPort` is the port of the candidate that it is derived from. For host candidates, the `relatedPort` is `null`. This corresponds to the `rel-port` field in `candidate-attribute`.

***`usernameFragment`* of type DOMString, readonly, nullable**
This carries the `ufrag` as defined in section 15.4 of [RFC5245].

CANDIDATE ADDITION 16: Add RTCIceCandidate.relayProtocol (PR #2763)

◉ Show Current and Future ○ Show Current ○ Show Future

***`relayProtocol`* of type RTCIceServerTransportProtocol, readonly, nullable**
For local candidates of type "`relay`" this is the protocol used by the endpoint to communicate with the TURN server. For all other candidates it is `null`.

CANDIDATE ADDITION 23: Add RTCIceCandidate.url (PR #2773)

◉ Show Current and Future ○ Show Current ○ Show Future

***`url`* of type DOMString, readonly, nullable**
For local candidates of type "`srflx`" or type "`relay`" this is the URL of the ICE server from which the candidate was obtained. For all other candidates it is `null`.

§ *Methods*

`toJSON()`
To invoke the `toJSON()` operation of the `RTCIceCandidate` interface, run the following steps:

1. Let *json* be a new `RTCIceCandidateInit` dictionary.
2. For each attribute identifier *attr* in «`candidate`, `sdpMid`, `sdpMLineIndex`, `usernameFragment`»:
    1. Let *value* be the result of getting the underlying value of the attribute identified by *attr*, given this `RTCIceCandidate` object.
    2. Set `json[attr]` to *value*.
3. Return *json*.

WebIDL

```
dictionary RTCIceCandidateInit {
  DOMString candidate = "";
  DOMString? sdpMid = null;
  unsigned short? sdpMLineIndex = null;
  DOMString? usernameFragment = null;
};
```

§ *Dictionary* `RTCIceCandidateInit` *Members*

***`candidate`* of type DOMString, defaulting to `""`**
This carries the `candidate-attribute` as defined in section 15.1 of [RFC5245]. If this represents an end-of-candidates indication, `candidate` is an empty string.

***`sdpMid`* of type DOMString, nullable, defaulting to `null`**
If not `null`, this contains the media stream "identification-tag" defined in [RFC5888] for the media component this candidate is associated with.

***`sdpMLineIndex`* of type unsigned short, nullable, defaulting to `null`**
If not `null`, this indicates the index (starting at zero) of the media description in the SDP this candidate is associated with.

***`usernameFragment`* of type DOMString, nullable, defaulting to `null`**
If not `null`, this carries the `ufrag` as defined in section 15.4 of [RFC5245].

§ 4.8.1.1 `candidate-attribute` Grammar

The `candidate-attribute` grammar is used to parse the `candidate` member of *candidateInitDict* in the `RTCIceCandidate()` constructor.

The primary grammar for `candidate-attribute` is defined in section 15.1 of [RFC5245]. In addition, the browser *MUST* support the grammar extension for ICE TCP as defined in section 4.5 of [RFC6544].

The browser *MAY* support other grammar extensions for `candidate-attribute` as defined in other RFCs.

§ 4.8.1.2 `RTCIceProtocol` Enum

The `RTCIceProtocol` represents the protocol of the ICE candidate.

WebIDL

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

`RTCIceProtocol` Enumeration description

Enum value	Description
`udp`	A UDP candidate, as described in [RFC5245].
`tcp`	A TCP candidate, as described in [RFC6544].

§ 4.8.1.3 `RTCIceTcpCandidateType` Enum

The `RTCIceTcpCandidateType` represents the type of the ICE TCP candidate, as defined in [RFC6544].

WebIDL

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

`RTCIceTcpCandidateType` Enumeration description

Enum value	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.

> ***NOTE***
>
> ***The user agent will typically only gather `active` ICE TCP candidates.***

§ 4.8.1.4 `RTCIceCandidateType` Enum

The RTCIceCandidateType represents the type of the ICE candidate, as defined in [RFC5245] section 15.1.

WebIDL

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

RTCIceCandidateType Enumeration description

Enum value	Description
host	A host candidate, as defined in Section 4.1.1.1 of [RFC5245].
srflx	A server reflexive candidate, as defined in Section 4.1.1.2 of [RFC5245].
prflx	A peer reflexive candidate, as defined in Section 4.1.1.2 of [RFC5245].
relay	A relay candidate, as defined in Section 7.1.3.2.1 of [RFC5245].

CANDIDATE ADDITION 16: Add RTCIceCandidate.relayProtocol (PR #2763)

Show Current and Future   Show Current   Show Future

§ *4.8.1.5* RTCIceServerTransportProtocol *Enum*

The RTCIceServerTransportProtocol represents the type of the transport protocol used between the client and the server, as defined in [RFC8656] section 3.1.

WebIDL

```
enum RTCIceServerTransportProtocol {
  "udp",
  "tcp",
  "tls",
};
```

RTCIceServerTransportProtocol Enumeration description

Enum value	Description
udp	The TURN client is using UDP as transport to the server.
tcp	The TURN client is using TCP as transport to the server.
tls	The TURN client is using TLS as transport to the server.

§ 4.8.2 RTCPeerConnectionIceEvent

The icecandidate event of the RTCPeerConnection uses the RTCPeerConnectionIceEvent interface.

When firing an RTCPeerConnectionIceEvent event that contains an RTCIceCandidate object, it *MUST* include values for both sdpMid and sdpMLineIndex. If the RTCIceCandidate is of type "srflx" or type "relay", the url property of the event *MUST* be set to the URL of the ICE server from which the candidate was obtained.

NOTE

The `icecandidate` event is used for three different types of indications:

- ***A candidate has been gathered. The `candidate` member of the event will be populated normally. It should be signaled to the remote peer and passed into `addIceCandidate`.***
- ***An `RTCIceTransport` has finished gathering a `generation` of candidates, and is providing an end-of-candidates indication as defined by Section 8.2 of [RFC8838]. This is indicated by `candidate`.`candidate` being set to an empty string. The `candidate` object should be signaled to the remote peer and passed into `addIceCandidate` like a typical ICE candidate, in order to provide the end-of-candidates indication to the remote peer.***
- ***All `RTCIceTransport`s have finished gathering candidates, and the `RTCPeerConnection`'s `RTCIceGatheringState` has transitioned to "`complete`". This is indicated by the `candidate` member of the event being set to `null`. This only exists for backwards compatibility, and this event does not need to be signaled to the remote peer. It's equivalent to an `icegatheringstatechange` event with the "`complete`" state.***

WebIDL

```
[Exposed=Window]
interface RTCPeerConnectionIceEvent : Event {
  constructor(DOMString type, optional RTCPeerConnectionIceEventInit eventInitDict =
{});
  readonly attribute RTCIceCandidate? candidate;
  readonly attribute DOMString? url;
};
```

§ *Constructors*

`RTCPeerConnectionIceEvent.constructor()`

§ *Attributes*

`candidate` of type `RTCIceCandidate`, readonly, nullable

The `candidate` attribute is the `RTCIceCandidate` object with the new ICE candidate that caused the event.

This attribute is set to `null` when an event is generated to indicate the end of candidate gathering.

NOTE

Even where there are multiple media components, only one event containing a `null` candidate is fired.

`url` of type DOMString, readonly, nullable

The `url` attribute is the STUN or TURN URL that identifies the STUN or TURN server used to gather this candidate. If the candidate was not gathered from a STUN or TURN server, this parameter will be set to `null`.

CANDIDATE CORRECTION 23: Mark RTCPeerConnectionIceEvent.url as deprecated (PR #2773)

◉ Show Current and Future ○ Show Current ○ Show Future

This attribute is deprecated; it exists for legacy compatibility reasons only. Prefer the candidate `url`.

WebIDL

```
dictionary RTCPeerConnectionIceEventInit : EventInit {
  RTCIceCandidate? candidate;
```

```
  DOMString? url;
};
```

§ *Dictionary* **`RTCPeerConnectionIceEventInit`** *Members*

`candidate` **of type `RTCIceCandidate`, nullable**
See the `candidate` attribute of the `RTCPeerConnectionIceEvent` interface.

`url` **of type DOMString, nullable**
The `url` attribute is the STUN or TURN URL that identifies the STUN or TURN server used to gather this candidate.

§ 4.8.3 `RTCPeerConnectionIceErrorEvent`

The `icecandidateerror` event of the `RTCPeerConnection` uses the `RTCPeerConnectionIceErrorEvent` interface.

WebIDL

```
[Exposed=Window]
interface RTCPeerConnectionIceErrorEvent : Event {
  constructor(DOMString type, RTCPeerConnectionIceErrorEventInit eventInitDict);
  readonly attribute DOMString? address;
  readonly attribute unsigned short? port;
  readonly attribute DOMString url;
  readonly attribute unsigned short errorCode;
  readonly attribute USVString errorText;
};
```

§ *Constructors*

`RTCPeerConnectionIceErrorEvent.constructor()`

§ *Attributes*

`address` **of type DOMString, readonly, nullable**
The `address` attribute is the local IP address used to communicate with the STUN or TURN server.

On a multihomed system, multiple interfaces may be used to contact the server, and this attribute allows the application to figure out on which one the failure occurred.

If the local IP address value is not already exposed as part of a local candidate, the `address` attribute will be set to `null`.

`port` **of type unsigned short, readonly, nullable**
The `port` attribute is the port used to communicate with the STUN or TURN server.

If the `address` attribute is `null`, the `port` attribute is also set to `null`.

`url` **of type DOMString, readonly**
The `url` attribute is the STUN or TURN URL that identifies the STUN or TURN server for which the failure occurred.

`errorCode` **of type unsigned short, readonly**
The `errorCode` attribute is the numeric STUN error code returned by the STUN or TURN server [STUN-PARAMETERS].

If no host candidate can reach the server, `errorCode` will be set to the value 701 which is outside the STUN error code range. This error is only fired once per server URL while in the `RTCIceGatheringState` of "`gathering`".

***`errorText`* of type USVString, readonly**

The `errorText` attribute is the STUN reason text returned by the STUN or TURN server [STUN-PARAMETERS].

If the server could not be reached, `errorText` will be set to an implementation-specific value providing details about the error.

WebIDL

```
dictionary RTCPeerConnectionIceErrorEventInit : EventInit {
  DOMString? address;
  unsigned short? port;
  DOMString url;
  required unsigned short errorCode;
  USVString errorText;
};
```

§ *Dictionary* **`RTCPeerConnectionIceErrorEventInit`** *Members*

***`address`* of type DOMString, nullable**

The local address used to communicate with the STUN or TURN server, or `null`.

***`port`* of type unsigned short, nullable**

The local port used to communicate with the STUN or TURN server, or `null`.

***`url`* of type DOMString**

The STUN or TURN URL that identifies the STUN or TURN server for which the failure occurred.

***`errorCode`* of type unsigned short, required**

The numeric STUN error code returned by the STUN or TURN server.

***`errorText`* of type USVString**

The STUN reason text returned by the STUN or TURN server.

§ 4.9 Certificate Management

The certificates that `RTCPeerConnection` instances use to authenticate with peers use the `RTCCertificate` interface. These objects can be explicitly generated by applications using the `generateCertificate` method and can be provided in the `RTCConfiguration` when constructing a new `RTCPeerConnection` instance.

The explicit certificate management functions provided here are optional. If an application does not provide the `certificates` configuration option when constructing an `RTCPeerConnection` a new set of certificates *MUST* be generated by the user agent. That set *MUST* include an ECDSA certificate with a private key on the P-256 curve and a signature with a SHA-256 hash.

WebIDL

```
partial interface RTCPeerConnection {
  static Promise<RTCCertificate>
      generateCertificate(AlgorithmIdentifier keygenAlgorithm);
};
```

§ **Methods**

***`generateCertificate`*, static**

The `generateCertificate` function causes the user agent to create an X.509 certificate [X509V3] and corresponding private key. A handle to information is provided in the form of the `RTCCertificate` interface. The returned `RTCCertificate` can be used to control the certificate that is offered in the DTLS sessions established by `RTCPeerConnection`.

The *keygenAlgorithm* argument is used to control how the private key associated with the certificate is generated. The *keygenAlgorithm* argument uses the WebCrypto [WebCryptoAPI] AlgorithmIdentifier type.

The following values *MUST* be supported by a user agent: `{ name: "RSASSA-PKCS1-v1_5", modulusLength: 2048, publicExponent: new Uint8Array([1, 0, 1]), hash: "SHA-256" }`, and `{ name: "ECDSA", namedCurve: "P-256" }`.

> ***NOTE***
>
> ***It is expected that a user agent will have a small or even fixed set of values that it will accept.***

The certificate produced by this process also contains a signature. The validity of this signature is only relevant for compatibility reasons. Only the public key and the resulting certificate fingerprint are used by `RTCPeerConnection`, but it is more likely that a certificate will be accepted if the certificate is well formed. The browser selects the algorithm used to sign the certificate; a browser *SHOULD* select SHA-256 [FIPS-180-4] if a hash algorithm is needed.

The resulting certificate *MUST NOT* include information that can be linked to a user or user agent. Randomized values for distinguished name and serial number *SHOULD* be used.

When the method is called, the user agent *MUST* run the following steps:

1. Let *keygenAlgorithm* be the first argument to `generateCertificate`.
2. Let *expires* be a value of 2592000000 (30*24*60*60*1000)

   > ***NOTE***
   > ***This means the certificate will by default expire in 30 days from the time of the `generateCertificate` call.***

3. If *keygenAlgorithm* is an object, run the following steps:

   1. Let *certificateExpiration* be the result of converting the ECMAScript object represented by *keygenAlgorithm* to an `RTCCertificateExpiration` dictionary.
   2. If the conversion fails with an *error*, return a promise that is rejected with *error*.
   3. If *certificateExpiration*.`expires` is not `undefined`, set *expires* to *certificateExpiration*.`expires`.
   4. If *expires* is greater than 31536000000, set *expires* to 31536000000.

      > ***NOTE***
      > ***This means the certificate cannot be valid for longer than 365 days from the time of the `generateCertificate` call.***

   A user agent *MAY* further cap the value of *expires*.

4. Let *normalizedKeygenAlgorithm* be the result of normalizing an algorithm with an operation name of generateKey and a supportedAlgorithms value specific to production of certificates for `RTCPeerConnection`.
5. If the above normalization step fails with an *error*, return a promise that is rejected with *error*.
6. If the *normalizedKeygenAlgorithm* parameter identifies an algorithm that the user agent cannot or will not use to generate a certificate for `RTCPeerConnection`, return a promise that is rejected with a `DOMException` of type `NotSupportedError`. In particular, *normalizedKeygenAlgorithm MUST* be an asymmetric algorithm that can be used to produce a

signature used to authenticate DTLS connections.

7. Let *p* be a new promise.
8. Run the following steps in parallel:
   1. Perform the generate key operation specified by *normalizedKeygenAlgorithm* using *keygenAlgorithm*.
   2. Let *generatedKeyingMaterial* and *generatedKeyCertificate* be the private keying material and certificate generated by the above step.
   3. Let *certificate* be a new `RTCCertificate` object.
   4. Set *certificate*.[[Expires]] to the current time plus *expires* value.
   5. Set *certificate*.`[[Origin]]` to the relevant settings object's origin.
   6. Store the *generatedKeyingMaterial* in a secure module, and let *handle* be a reference identifier to it.
   7. Set *certificate*.`[[KeyingMaterialHandle]]` to *handle*.
   8. Set *certificate*.`[[Certificate]]` to *generatedCertificate*.
   9. Resolve *p* with *certificate*.
9. Return *p*.

PROPOSED CORRECTION 7: Replace `DOMTimeStamp` in the definition of the `RTCCertificateExpiration.expires` and of `RTCCertificate.expires`, and change its origin to certificate creation time (PR #2686, PR #2700)

Show Current and Future Show Current Show Future

~~4.9.1 `RTCCertificateExpiration` Dictionary~~

§ 4.9.1 `RTCCertificateExpiration` Dictionary

`RTCCertificateExpiration` is used to set an expiration date on certificates generated by `generateCertificate`.

```
dictionary RTCCertificateExpiration {
  [EnforceRange] DOMTimeStamp unsigned long long expires;
};
```

***`expires`*, of type ~~`DOMTimeStamp`~~ unsigned long long**
: An optional `expires` attribute *MAY* be added to the definition of the algorithm that is passed to `generateCertificate`. If this parameter is present it indicates the maximum time in milliseconds that the `RTCCertificate` is valid ~~for relative to~~ for, measured from the ~~current time~~time the certificate is created.

PROPOSED CORRECTION 7: Replace `DOMTimeStamp` in the definition of the `RTCCertificateExpiration.expires` and of `RTCCertificate.expires`, and change its origin to certificate creation time (PR #2686, PR #2700)

Show Current and Future Show Current Show Future

~~4.9.2 `RTCCertificate` Interface~~

§ 4.9.2 `RTCCertificate` Interface

The `RTCCertificate` interface represents a certificate used to authenticate WebRTC communications. In addition to the visible properties, internal slots contain a handle to the generated private keying materal (**[[KeyingMaterialHandle]]**), a certificate (**[[Certificate]]**) that `RTCPeerConnection` uses to authenticate with a peer, and the origin (**[[Origin]]**) that created the object.

```
[Exposed=Window, Serializable]
interface RTCCertificate {
  readonly attribute DOMTimeStamp EpochTimeStamp expires;
  sequence<RTCDtlsFingerprint> getFingerprints();
};
```

~~*Attributes*~~

§ *Attributes*

`expires` **of type ~~DOMTimeStamp~~`EpochTimeStamp`, readonly**

The *expires* attribute indicates the date and time in milliseconds relative to 1970-01-01T00:00:00Z after which the certificate will be considered invalid by the browser. After this time, attempts to construct an `RTCPeerConnection` using this certificate fail.

Note that this value might not be reflected in a `notAfter` parameter in the certificate itself.

~~*Methods*~~

§ *Methods*

`getFingerprints`

Returns the list of certificate fingerprints, one of which is computed with the digest algorithm used in the certificate signature.

For the purposes of this API, the `[[Certificate]]` slot contains unstructured binary data. No mechanism is provided for applications to access the `[[KeyingMaterialHandle]]` internal slot or the keying material it references. Implementations *MUST* support applications storing and retrieving `RTCCertificate` objects from persistent storage, in a manner that also preserves the keying material referenced by `[[KeyingMaterialHandle]]`. Implementations *SHOULD* store the sensitive keying material in a secure module safe from same-process memory attacks. This allows the private key to be stored and used, but not easily read using a memory attack.

`RTCCertificate` objects are serializable objects [HTML]. Their serialization steps, given *value* and *serialized*, are:

1. Set *serialized*.[[Expires]] to the value of *value*.`expires` attribute.
2. Set *serialized*.[[Certificate]] to a copy of the unstructured binary data in *value*.`[[Certificate]]`.
3. Set *serialized*.[[Origin]] to a copy of the unstructured binary data in *value*.`[[Origin]]`.
4. Set *serialized*.[[KeyingMaterialHandle]] to a serialization of the handle in *value*.`[[KeyingMaterialHandle]]` (not the private keying material itself).

Their deserialization steps, given *serialized* and *value*, are:

1. Initialize *value*.`expires` attribute to contain *serialized*.[[Expires]].
2. Set *value*.`[[Certificate]]` to a copy of *serialized*.[[Certificate]]~~.~~

3. Set *value*.`[[Origin]]` to a copy of *serialized*.[[Origin]]~~.~~
4. Set *value*.`[[KeyingMaterialHandle]]` to the private keying material handle resulting from deserializing *serialized*.[[KeyingMaterialHandle]]~~.~~

> ***NOTE***
>
> ***Supporting structured cloning in this manner allows `RTCCertificate` instances to be persisted to stores. It also allows instances to be passed to other origins using APIs like `postMessage(message, options)` [html]. However, the object cannot be used by any other origin than the one that originally created it.***

§ 5. RTP Media API

The **RTP media API** lets a web application send and receive `MediaStreamTrack`s over a peer-to-peer connection. Tracks, when added to an `RTCPeerConnection`, result in signaling; when this signaling is forwarded to a remote peer, it causes corresponding tracks to be created on the remote side.

> ***NOTE***
>
> ***There is not an exact 1:1 correspondence between tracks sent by one `RTCPeerConnection` and received by the other. For one, IDs of tracks sent have no mapping to the IDs of tracks received. Also, `replaceTrack` changes the track sent by an `RTCRtpSender` without creating a new track on the receiver side; the corresponding `RTCRtpReceiver` will only have a single track, potentially representing multiple sources of media stitched together. Both `addTransceiver` and `replaceTrack` can be used to cause the same track to be sent multiple times, which will be observed on the receiver side as multiple receivers each with its own separate track. Thus it's more accurate to think of a 1:1 relationship between an `RTCRtpSender` on one side and an `RTCRtpReceiver`'s track on the other side, matching senders and receivers using the `RTCRtpTransceiver`'s `mid` if necessary.***

When sending media, the sender may need to rescale or resample the media to meet various requirements, including the envelope negotiated by SDP, alignment restrictions of the encoder, or even CPU overuse detection or bandwidth estimation.

Following the rules in [RFC9429] (section 3.6.), the video *MAY* be downscaled. The media *MUST NOT* be upscaled to create fake data that did not occur in the input source, the media *MUST NOT* be cropped except as needed to satisfy constraints on pixel counts, and the aspect ratio *MUST NOT* be changed.

> ***NOTE***
>
> ***The WebRTC Working Group is seeking implementation feedback on the need and timeline for a more complex handling of this situation. Some possible designs have been discussed in GitHub issue 1283.***

PROPOSED CORRECTION 43: Allow encoder resolution alignment in scaleResolutionDownBy. (PR #2808)

◉ Show Current and Future ○ Show Current ○ Show Future

~~When~~ Whenever video is ~~rescaled, for example for certain combinations of width or height and~~ rescaled as a result of `scaleResolutionDownBy` ~~values,~~ situations when the resulting width or height is not an integer may occur. ~~In such situations the~~ The user agent ~~*MUST*~~*MUST NOT* ~~use~~ transmit video larger than the integer ~~part of the result~~part of the scaled width and height from `scaleResolutionDownBy`, except to respect an encoder's minimum resolution. What to transmit if the integer part ~~of~~ of the scaled width ~~or~~ or height is zero is ~~implementation-specific~~implementation-defined.

The actual encoding and transmission of `MediaStreamTrack`s is managed through objects called `RTCRtpSender`s. Similarly, the reception and decoding of `MediaStreamTrack`s is managed through objects called RTCRtpReceivers. Each RTCRtpSender is associated with at most one track, and each track to be received is associated with exactly one RTCRtpReceiver.

The encoding and transmission of each MediaStreamTrack *SHOULD* be made such that its characteristics (width, height and frameRate for video tracks; sampleSize, sampleRate and channelCount for audio tracks) are to a reasonable degree retained by the track created on the remote side. There are situations when this does not apply, there may for example be resource constraints at either endpoint or in the network or there may be RTCRtpSender settings applied that instruct the implementation to act differently.

An RTCPeerConnection object contains a **set of RTCRtpTransceivers**, representing the paired senders and receivers with some shared state. This set is initialized to the empty set when the RTCPeerConnection object is created. RTCRtpSenders and RTCRtpReceivers are always created at the same time as an RTCRtpTransceiver, which they will remain attached to for their lifetime. RTCRtpTransceivers are created implicitly when the application attaches a MediaStreamTrack to an RTCPeerConnection via the addTrack() method, or explicitly when the application uses the addTransceiver method. They are also created when a remote description is applied that includes a new media description. Additionally, when a remote description is applied that indicates the remote endpoint has media to send, the relevant MediaStreamTrack and RTCRtpReceiver are surfaced to the application via the track event.

In order for an RTCRtpTransceiver to send and/or receive media with another endpoint this must be negotiated with SDP such that both endpoints have an RTCRtpTransceiver object that is associated with the same media description.

When creating an offer, enough media descriptions will be generated to cover all transceivers on that end. When this offer is set as the local description, any disassociated transceivers get associated with media descriptions in the offer.

When an offer is set as the remote description, any media descriptions in it not yet associated with a transceiver get associated with a new or existing transceiver. In this case, only disassociated transceivers that were created via the addTrack() method may be associated. Disassociated transceivers created via the addTransceiver() method, however, won't get associated even if media descriptions are available in the remote offer. Instead, new transceivers will be created and associated if there aren't enough addTrack()-created transceivers. This sets addTrack()-created and addTransceiver()-created transceivers apart in a critical way that is not observable from inspecting their attributes.

When creating an answer, only media descriptions that were present in the offer may be listed in the answer. As a consequence, any transceivers that were not associated when setting the remote offer remain disassociated after setting the local answer. This can be remedied by the answerer creating a follow-up offer, initiating another offer/answer exchange, or in the case of using addTrack()-created transceivers, making sure that enough media descriptions are offered in the initial exchange.

§ 5.1 RTCPeerConnection Interface Extensions

The RTP media API extends the RTCPeerConnection interface as described below.

WebIDL

```
        partial interface RTCPeerConnection {
  sequence<RTCRtpSender> getSenders();
  sequence<RTCRtpReceiver> getReceivers();
  sequence<RTCRtpTransceiver> getTransceivers();
  RTCRtpSender addTrack(MediaStreamTrack track, MediaStream... streams);
  undefined removeTrack(RTCRtpSender sender);
  RTCRtpTransceiver addTransceiver((MediaStreamTrack or DOMString) trackOrKind,
                                   optional RTCRtpTransceiverInit init = {});
  attribute EventHandler ontrack;
};
```

§ Attributes

`ontrack` **of type EventHandler**
The event type of this event handler is `track`.

§ Methods

`getSenders`
Returns a sequence of `RTCRtpSender` objects representing the RTP senders that belong to non-stopped `RTCRtpTransceiver` objects currently attached to this `RTCPeerConnection` object.

When the `getSenders` method is invoked, the user agent *MUST* return the result of executing the `CollectSenders` algorithm.

We define the **CollectSenders** algorithm as follows:

1. Let *transceivers* be the result of executing the `CollectTransceivers` algorithm.
2. Let *senders* be a new empty sequence.
3. For each *transceiver* in *transceivers*,
    1. If *transceiver*.`[[Stopped]]` is `false`, add *transceiver*.`[[Sender]]` to *senders*.
4. Return *senders*.

`getReceivers`
Returns a sequence of `RTCRtpReceiver` objects representing the RTP receivers that belong to non-stopped `RTCRtpTransceiver` objects currently attached to this `RTCPeerConnection` object.

When the `getReceivers` method is invoked, the user agent *MUST* run the following steps:

1. Let *transceivers* be the result of executing the `CollectTransceivers` algorithm.
2. Let *receivers* be a new empty sequence.
3. For each *transceiver* in *transceivers*,
    1. If *transceiver*.`[[Stopped]]` is `false`, add *transceiver*.`[[Receiver]]` to *receivers*.
4. Return *receivers*.

`getTransceivers`
Returns a sequence of `RTCRtpTransceiver` objects representing the RTP transceivers that are currently attached to this `RTCPeerConnection` object.

The `getTransceivers` method *MUST* return the result of executing the `CollectTransceivers` algorithm.

We define the **CollectTransceivers** algorithm as follows:

1. Let *transceivers* be a new sequence consisting of all `RTCRtpTransceiver` objects in this `RTCPeerConnection` object's set of transceivers, in insertion order.
2. Return *transceivers*.

`addTrack`
Adds a new track to the `RTCPeerConnection`, and indicates that it is contained in the specified `MediaStream`s.

When the `addTrack` method is invoked, the user agent *MUST* run the following steps:

1. Let *connection* be the `RTCPeerConnection` object on which this method was invoked.
2. Let *track* be the `MediaStreamTrack` object indicated by the method's first argument.
3. Let *kind* be *track.kind*.
4. Let *streams* be a list of `MediaStream` objects constructed from the method's remaining arguments, or an empty list if the method was called with a single argument.
5. If *connection*.`[[IsClosed]]` is `true`, throw an `InvalidStateError`.
6. Let *senders* be the result of executing the `CollectSenders` algorithm. If an `RTCRtpSender` for *track* already exists in *senders*, throw an `InvalidAccessError`.

7. The steps below describe how to determine if an existing sender can be reused. Doing so will cause future calls to `createOffer` and `createAnswer` to mark the corresponding media description as `sendrecv` or `sendonly` and add the MSID of the sender's streams, as defined in [RFC9429] (section 5.2.2. and section 5.3.2.).

If any `RTCRtpSender` object in *senders* matches all the following criteria, let *sender* be that object, or `null` otherwise:

- The sender's track is null.
- The transceiver kind of the `RTCRtpTransceiver`, associated with the sender, matches *kind*.
- The `[[Stopping]]` slot of the `RTCRtpTransceiver` associated with the sender is `false`.
- The sender has never been used to send. More precisely, the `[[CurrentDirection]]` slot of the `RTCRtpTransceiver` associated with the sender has never had a value of "`sendrecv`" or "`sendonly`".

8. If *sender* is not `null`, run the following steps to use that sender:

   1. Set *sender*.`[[SenderTrack]]` to *track*.
   2. Set *sender*.`[[AssociatedMediaStreamIds]]` to an empty set.
   3. For each *stream* in *streams*, add *stream.id* to `[[AssociatedMediaStreamIds]]` if it's not already there.
   4. Let *transceiver* be the `RTCRtpTransceiver` associated with *sender*.
   5. If *transceiver*.`[[Direction]]` is "`recvonly`", set *transceiver*.`[[Direction]]` to "`sendrecv`".
   6. If *transceiver*.`[[Direction]]` is "`inactive`", set *transceiver*.`[[Direction]]` to "`sendonly`".

9. If *sender* is `null`, run the following steps:

   1. Create an RTCRtpSender with *track*, *kind* and *streams*, and let *sender* be the result.
   2. Create an RTCRtpReceiver with *kind*, and let *receiver* be the result.
   3. Create an RTCRtpTransceiver with *sender*, *receiver* and an `RTCRtpTransceiverDirection` value of "`sendrecv`", and let *transceiver* be the result.
   4. Add *transceiver* to *connection*'s set of transceivers.

10. A track could have contents that are inaccessible to the application. This can be due to anything that would make a track CORS cross-origin. These tracks can be supplied to the `addTrack()` method, and have an `RTCRtpSender` created for them, but content *MUST NOT* be transmitted. Silence (audio), black frames (video) or equivalently absent content is sent in place of track content.

Note that this property can change over time.

11. Update the negotiation-needed flag for *connection*.

12. Return *sender*.

`removeTrack`

Stops sending media from *sender*. The `RTCRtpSender` will still appear in `getSenders`. Doing so will cause future calls to `createOffer` to mark the media description for the corresponding transceiver as "`recvonly`" or "`inactive`", as defined in [RFC9429] (section 5.2.2.).

When the other peer stops sending a track in this manner, the track is removed from any remote `MediaStream`s that were initially revealed in the `track` event, and if the `MediaStreamTrack` is not already muted, a `mute` event is fired at the track.

NOTE

The same effect as `removeTrack()` can be achieved by setting the `RTCRtpTransceiver.direction` attribute of the corresponding transceiver and invoking `RTCRtpSender.replaceTrack`(null) on the sender. One minor difference is that `replaceTrack()` is asynchronous and `removeTrack()` is synchronous.

When the `removeTrack` method is invoked, the user agent *MUST* run the following steps:

PROPOSED CORRECTION 34: Make removeTrack() a no-op after transceiver.stop() (PR #2875)

◉ Show Current and Future ○ Show Current ○ Show Future

1. Let *sender* be the argument to `removeTrack`.
2. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.
3. If *connection*.`[[IsClosed]]` is `true`, throw an `InvalidStateError`.
4. If *sender* was not created by *connection*, throw an `InvalidAccessError`.
5. Let *transceiver* be the `RTCRtpTransceiver` object corresponding to *sender*.
6. If *transceiver*.`[[Stopping]]` is true, abort these steps.
7. ~~Let~~

Let *senders* be the result of executing the `CollectSenders` algorithm.

8. If *sender* is not in *senders* (which indicates its transceiver was stopped or removed due to setting a session description of `type` "`rollback`"), then abort these steps.
9. If *sender*.`[[SenderTrack]]` is null, abort these steps.
10. Set *sender*.`[[SenderTrack]]` to null.
11. ~~Let *transceiver* be the `RTCRtpTransceiver` object corresponding to *sender*.~~
12. If *transceiver*.`[[Direction]]` is "`sendrecv`", set *transceiver*.`[[Direction]]` to "`recvonly`".
13. If *transceiver*.`[[Direction]]` is "`sendonly`", set *transceiver*.`[[Direction]]` to "`inactive`".
14. Update the negotiation-needed flag for *connection*.

`addTransceiver`

Create a new `RTCRtpTransceiver` and add it to the set of transceivers.

Adding a transceiver will cause future calls to `createOffer` to add a media description for the corresponding transceiver, as defined in [RFC9429] (section 5.2.2.).

The initial value of `mid` is null. Setting a session description may later change it to a non-null value.

The `sendEncodings` argument can be used to specify the number of offered simulcast encodings, and optionally their RIDs and encoding parameters.

When this method is invoked, the user agent *MUST* run the following steps:

1. Let *init* be the second argument.
2. Let *streams* be *init*.`streams`.
3. Let *sendEncodings* be *init*.`sendEncodings`.
4. Let *direction* be *init*.`direction`.
5. If the first argument is a string, let *kind* be the first argument and run the following steps:
    1. If *kind* is neither `"audio"` nor `"video"`, throw a `TypeError`.
    2. Let *track* be `null`.

6. If the first argument is a `MediaStreamTrack`, let *track* be the first argument and let *kind* be *track*.`kind`.

7. If *connection*.`[[IsClosed]]` is `true`, throw an `InvalidStateError`.

8. Validate *sendEncodings* by running the following **addTransceiver sendEncodings validation steps**, where each `RTCRtpEncodingParameters` dictionary in it is an "encoding":

CANDIDATE CORRECTION 18: TypeError unless all or none of encodings have rids and on duplicate rids (PR #2774, PR #2775)

Show Current and Future Show Current Show Future

1. ~~Verify that each `rid` value in *sendEncodings* conforms to the grammar specified in Section 10 of [RFC8851]. If one of the RIDs does not meet these requirements, throw a `TypeError`.~~

If any of the following conditions are met, throw a `TypeError`:

- Any encoding contains a `rid` member whose value does not conform to the grammar requirements specified in Section 10 of [RFC8851].
- Some but not all encodings contain a `rid` member.
- Any encoding contains a `rid` member whose value is the same as that of a `rid` contained in another encoding in *sendEncodings*.

2. If any encoding contains a read-only parameter other than `rid`, throw an `InvalidAccessError`.

3. CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

Show Current and Future Show Current Show Future

If any encoding contains a `codec` member whose value does not match any codec in `RTCRtpSender.getCapabilities(`*kind*`).codecs`, throw an `OperationError`.

4. CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

Show Current and Future Show Current Show Future

If the user agent does not support changing codecs without negotiation or does not support setting codecs for individual encodings, return a promise rejected with a newly created `OperationError`.

5. **PROPOSED ADDITION 19:** Add RTCRtpEncodingParameters.maxFramerate (PR #2785)

PROPOSED CORRECTION 20: Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio (PR #2772, PR #2799)

Show Current and Future Show Current Show Future

If *kind* is `"audio"`, remove the `scaleResolutionDownBy` and `maxFramerate` members from all encodings that contain any of them.

6. If any encoding contains a `scaleResolutionDownBy` member whose value is less than `1.0`, throw a `RangeError`.

7. **PROPOSED ADDITION 19:** Add RTCRtpEncodingParameters.maxFramerate (PR #2785)

◉ Show Current and Future ○ Show Current ○ Show Future

Verify that the value of each `maxFramerate` member in *sendEncodings* that is defined is greater than 0.0. If one of the `maxFramerate` values does not meet this requirement, throw a `RangeError`.

8. Let *maxN* be the maximum number of total simultaneous encodings the user agent may support for this *kind*, at minimum 1.This should be an optimistic number since the codec to be used is not known yet.

9. If any encoding contains a `scaleResolutionDownBy` member, then for each encoding without one, add a `scaleResolutionDownBy` member with the value `1.0`.

10. If the number of encodings stored in *sendEncodings* exceeds *maxN*, then trim *sendEncodings* from the tail until its length is *maxN*.

PROPOSED CORRECTION 20: Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio (PR #2772, PR #2799)

◉ Show Current and Future ○ Show Current ○ Show Future

11. ~~If the `scaleResolutionDownBy` attribues of *sendEncodings* are still undefined, initialize each encoding's `scaleResolutionDownBy` to `2^(length of sendEncodings - encoding index - 1)`. This results in smaller-to-larger resolutions where the last encoding has no scaling applied to it, e.g. 4:2:1 if the length is 3.~~

If *kind* is `"video"` and none of the encodings contain a `scaleResolutionDownBy` member, then for each encoding, add a `scaleResolutionDownBy` member with the value `2^(length of sendEncodings - encoding index - 1)`. This results in smaller-to-larger resolutions where the last encoding has no scaling applied to it, e.g. 4:2:1 if the length is 3.

12. If the number of encodings now stored in *sendEncodings* is `1`, then remove any `rid` member from the lone entry.

> ***NOTE***
> ***Providing a single, default `RTCRtpEncodingParameters` in sendEncodings allows the application to subsequently set encoding parameters using `setParameters`, even when simulcast isn't used.***

9. Create an RTCRtpSender with *track*, *kind*, *streams* and *sendEncodings* and let *sender* be the result.

If *sendEncodings* is set, then subsequent calls to `createOffer` will be configured to send multiple RTP encodings as defined in [RFC9429] (section 5.2.2. and section 5.2.1.). When `setRemoteDescription` is called with a corresponding remote description that is able to receive multiple RTP encodings as defined in [RFC9429] (section 3.7.), the `RTCRtpSender` may send multiple RTP encodings and the parameters retrieved via the transceiver's `sender.getParameters()` will reflect the encodings negotiated.

10. Create an RTCRtpReceiver with *kind* and let *receiver* be the result.

11. Create an RTCRtpTransceiver with *sender*, *receiver* and *direction*, and let *transceiver* be the result.

12. Add *transceiver* to *connection*'s set of transceivers.

13. Update the negotiation-needed flag for *connection*.

14. Return *transceiver*.

WebIDL

```
dictionary RTCRtpTransceiverInit {
  RTCRtpTransceiverDirection direction = "sendrecv";
  sequence<MediaStream> streams = [];
```

```
  sequence<RTCRtpEncodingParameters> sendEncodings = [];
};
```

§ Dictionary RTCRtpTransceiverInit Members

direction **of type RTCRtpTransceiverDirection, defaulting to "sendrecv"**
: The direction of the RTCRtpTransceiver.

streams **of type sequence<MediaStream>**
: When the remote RTCPeerConnection's track event fires corresponding to the RTCRtpReceiver being added, these are the streams that will be put in the event.

sendEncodings **of type sequence<RTCRtpEncodingParameters>**
: A sequence containing parameters for sending RTP encodings of media.

WebIDL

```
enum RTCRtpTransceiverDirection {
  "sendrecv",
  "sendonly",
  "recvonly",
  "inactive",
  "stopped"
};
```

`RTCRtpTransceiverDirection` *Enumeration description*

Enum value	Description
`sendrecv`	The RTCRtpTransceiver's RTCRtpSender *sender* will offer to send RTP, and will send RTP if the remote peer accepts and *sender*.getParameters().encodings[*i*].active is `true` for any value of *i*. The RTCRtpTransceiver's RTCRtpReceiver will offer to receive RTP, and will receive RTP if the remote peer accepts.
`sendonly`	The RTCRtpTransceiver's RTCRtpSender *sender* will offer to send RTP, and will send RTP if the remote peer accepts and *sender*.getParameters().encodings[*i*].active is `true` for any value of *i*. The RTCRtpTransceiver's RTCRtpReceiver will not offer to receive RTP, and will not receive RTP.
`recvonly`	The RTCRtpTransceiver's RTCRtpSender will not offer to send RTP, and will not send RTP. The RTCRtpTransceiver's RTCRtpReceiver will offer to receive RTP, and will receive RTP if the remote peer accepts.
`inactive`	The RTCRtpTransceiver's RTCRtpSender will not offer to send RTP, and will not send RTP. The RTCRtpTransceiver's RTCRtpReceiver will not offer to receive RTP, and will not receive RTP.
`stopped`	The RTCRtpTransceiver will neither send nor receive RTP. It will generate a zero port in the offer. In answers, its RTCRtpSender will not offer to send RTP, and its RTCRtpReceiver will not offer to receive RTP. This is a terminal state.

§ 5.1.1 Processing Remote MediaStreamTracks

An application can reject incoming media descriptions by setting the transceiver's direction to either "inactive" to turn off both directions temporarily, or to "sendonly" to reject only the incoming side. To permanently reject an m-line in a manner that makes it available for reuse, the application would need to call RTCRtpTransceiver.stop() and subsequently initiate negotiation from its end.

To **process remote tracks** given an RTCRtpTransceiver *transceiver*, *direction*, *msids*, *addList*, *removeList*, and *trackEventInits*, run the following steps:

1. Set the associated remote streams with *transceiver*.[[Receiver]], *msids*, *addList*, and *removeList*.

2. If *direction* is "sendrecv" or "recvonly" and *transceiver*.[[FiredDirection]] is neither "sendrecv" nor "recvonly", or the previous step increased the length of *addList*, process the addition of a remote

track with *transceiver* and *trackEventInits*.

3. If *direction* is "sendonly" or "inactive", set *transceiver*.[[Receptive]] to false.

4. If *direction* is "sendonly" or "inactive", and *transceiver*.[[FiredDirection]] is either "sendrecv" or "recvonly", process the removal of a remote track for the media description, with *transceiver* and *muteTracks*.

5. Set *transceiver*.[[FiredDirection]] to *direction*.

To **process the addition of a remote track** given an RTCRtpTransceiver *transceiver* and *trackEventInits*, run the following steps:

1. Let *receiver* be *transceiver*.[[Receiver]].
2. Let *track* be *receiver*.[[ReceiverTrack]].
3. Let *streams* be *receiver*.[[AssociatedRemoteMediaStreams]].
4. Create a new RTCTrackEventInit dictionary with *receiver*, *track*, *streams* and *transceiver* as members and add it to *trackEventInits*.

To **process the removal of a remote track** with an RTCRtpTransceiver *transceiver* and *muteTracks*, run the following steps:

1. Let *receiver* be *transceiver*.[[Receiver]].
2. Let *track* be *receiver*.[[ReceiverTrack]].
3. If *track.muted* is false, add *track* to *muteTracks*.

To **set the associated remote streams** given RTCRtpReceiver *receiver*, *msids*, *addList*, and *removeList*, run the following steps:

1. Let *connection* be the RTCPeerConnection object associated with *receiver*.
2. For each MSID in *msids*, unless a MediaStream object has previously been created with that id for this *connection*, create a MediaStream object with that id.
3. Let *streams* be a list of the MediaStream objects created for this *connection* with the ids corresponding to *msids*.
4. Let *track* be *receiver*.[[ReceiverTrack]].
5. For each *stream* in *receiver*.[[AssociatedRemoteMediaStreams]] that is not present in *streams*, add *stream* and *track* as a pair to *removeList*.
6. For each *stream* in *streams* that is not present in *receiver*.[[AssociatedRemoteMediaStreams]], add *stream* and *track* as a pair to *addList*.
7. Set *receiver*.[[AssociatedRemoteMediaStreams]] to *streams*.

§ 5.2 **RTCRtpSender** Interface

The RTCRtpSender interface allows an application to control how a given MediaStreamTrack is encoded and transmitted to a remote peer. When setParameters is called on an RTCRtpSender object, the encoding is changed appropriately.

To **create an RTCRtpSender** with a MediaStreamTrack, *track*, a string, *kind*, a list of MediaStream objects, *streams*, and optionally a list of RTCRtpEncodingParameters objects, *sendEncodings*, run the following steps:

1. Let *sender* be a new RTCRtpSender object.
2. Let *sender* have a **[[SenderTrack]]** internal slot initialized to *track*.
3. Let *sender* have a **[[SenderTransport]]** internal slot initialized to null.

4. Let *sender* have a **[[LastStableStateSenderTransport]]** internal slot initialized to `null`.

5. Let *sender* have a **[[Dtmf]]** internal slot initialized to `null`.

6. If *kind* is `"audio"` then create an RTCDTMFSender *dtmf* and set the `[[Dtmf]]` internal slot to *dtmf*.

7. Let *sender* have an **[[AssociatedMediaStreamIds]]** internal slot, representing a list of Ids of `MediaStream` objects that this sender is to be associated with. The `[[AssociatedMediaStreamIds]]` slot is used when *sender* is represented in SDP as described in [RFC9429] (section 5.2.1.).

8. Set *sender*.`[[AssociatedMediaStreamIds]]` to an empty set.

9. For each *stream* in *streams*, add *stream.id* to `[[AssociatedMediaStreamIds]]` if it's not already there.

10. Let *sender* have a **[[SendEncodings]]** internal slot, representing a list of `RTCRtpEncodingParameters` dictionaries.

PROPOSED CORRECTION 21: Default RTCRtpEncodingParameters.scaleResolutionDownBy to 1 for video (PR #2772)

Show Current and Future Show Current Show Future

11. If *sendEncodings* is given as input to this algorithm, and is non-empty, set the `[[SendEncodings]]` slot to *sendEncodings*. Otherwise, set it to a list containing a single new `RTCRtpEncodingParameters` ~~with `active`~~dictionary, ~~set~~ and if *kind* is `"video"`, add a `scaleResolutionDownBy` member with the value `1.0` to ~~`true`~~that dictionary.

> ***NOTE***
>
> ***`RTCRtpEncodingParameters` dictionaries contain `active` members whose values are `true` by default.***

12. CANDIDATE CORRECTION 13: Rollback restores ridless encoding trounced by sRD(simulcastOffer). (PR #2797)

Show Current and Future Show Current Show Future

Let *sender* have a **[[LastStableRidlessSendEncodings]]** internal slot initialized to `null`.

13. Let *sender* have a **[[SendCodecs]]** internal slot, representing a list of `RTCRtpCodecParameters` dictionaries, and initialized to an empty list.

14. Let *sender* have a **[[LastReturnedParameters]]** internal slot, which will be used to match `getParameters` and `setParameters` transactions.

15. Return *sender*.

WebIDL

```
[Exposed=Window]
interface RTCRtpSender {
  readonly attribute MediaStreamTrack? track;
  readonly attribute RTCDtlsTransport? transport;
  static RTCRtpCapabilities? getCapabilities(DOMString kind);
  Promise<undefined> setParameters(RTCRtpSendParameters parameters,
      optional RTCSetParameterOptions setParameterOptions = {});
  RTCRtpSendParameters getParameters();
  Promise<undefined> replaceTrack(MediaStreamTrack? withTrack);
  undefined setStreams(MediaStream... streams);
  Promise<RTCStatsReport> getStats();
};
```

§ Attributes

track **of type `MediaStreamTrack`, readonly, nullable**

The `track` attribute is the track that is associated with this `RTCRtpSender` object. If `track` is ended, or if the track's output is disabled, i.e. the track is disabled and/or muted, the `RTCRtpSender` *MUST* send black frames (video) and *MUST NOT* send (audio). In the case of video, the `RTCRtpSender` *SHOULD* send one black frame per second. If `track` is `null` then the `RTCRtpSender` does not send. On getting, the attribute *MUST* return the value of the `[[SenderTrack]]` slot.

transport **of type `RTCDtlsTransport`, readonly, nullable**

The `transport` attribute is the transport over which media from `track` is sent in the form of RTP packets. Prior to construction of the `RTCDtlsTransport` object, the `transport` attribute will be null. When bundling is used, multiple `RTCRtpSender` objects will share one `transport` and will all send RTP and RTCP over the same transport.

On getting, the attribute *MUST* return the value of the `[[SenderTransport]]` slot.

§ Methods

getCapabilities**, static**

The static `RTCRtpSender.getCapabilities()` method provides a way to discover the types of capabilities the user agent supports for sending media of the given kind, without reserving any resources, ports, or other state.

When the `getCapabilities` method is called, the user agent *MUST* run the following steps:

1. Let *kind* be the method's first argument.
2. If *kind* is neither `"video"` nor `"audio"` return `null`.
3. Return a new `RTCRtpCapabilities` dictionary, with its `codecs` member initialized to the list of implemented send codecs for *kind*, and its `headerExtensions` member initialized to the list of implemented header extensions for sending with *kind*.

The **list of implemented send codecs**, given *kind*, is an implementation-defined list of `RTCRtpCodec` dictionaries representing the most optimistic view of the codecs the user agent supports for sending media of the given *kind* (video or audio).

The **list of implemented header extensions for sending**, given *kind*, is an implementation-defined list of `RTCRtpHeaderExtensionCapability` dictionaries representing the most optimistic view of the header extensions the user agent supports for sending media of the given *kind* (video or audio).

These capabilities provide generally persistent cross-origin information on the device and thus increases the fingerprinting surface of the application. In privacy-sensitive contexts, user agents *MAY* consider mitigations such as reporting only a common subset of the capabilities.

> ***NOTE***
> ***The codec capabilities returned affect the `setCodecPreferences()` algorithm and what inputs it throws `InvalidModificationError` on, and should also be consistent with information revealed by `createOffer()` and `createAnswer()` about codecs negotiated for sending, to ensure any privacy mitigations are effective.***

setParameters

The `setParameters` method updates how `track` is encoded and transmitted to a remote peer.

When the `setParameters` method is called, the user agent *MUST* run the following steps:

1. Let *parameters* be the method's first argument.
2. Let *sender* be the `RTCRtpSender` object on which `setParameters` is invoked.
3. Let *transceiver* be the `RTCRtpTransceiver` object associated with *sender* (i.e. *sender* is *transceiver*.`[[Sender]]`).

PROPOSED CORRECTION 32: Reject setParameters(), replaceTrack(), & insertDTMF() after stop() (PR #2829)

Show Current and Future Show Current Show Future

4. If *transceiver*.[[~~Stopped~~Stopping]] is `true`, return a promise rejected with a newly created `InvalidStateError`.

5. If *sender*.`[[LastReturnedParameters]]` is `null`, return a promise rejected with a newly created `InvalidStateError`.

6. Validate *parameters* by running the following **setParameters validation steps**:

   1. Let *encodings* be *parameters*.`encodings`.

   2. Let *codecs* be *parameters*.`codecs`.

   3. CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

      Show Current and Future Show Current Show Future

      Let *choosableCodecs* be *codecs*.

   4. CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

      Show Current and Future Show Current Show Future

      If *choosableCodecs* is an empty list, set *choosableCodecs* to transceiver.`[[PreferredCodecs]]`.

   5. CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

      Show Current and Future Show Current Show Future

      If *choosableCodecs* is still an empty list, set *choosableCodecs* to the list of implemented send codecs for transceiver's kind.

   6. Let *N* be the number of `RTCRtpEncodingParameters` stored in *sender*.`[[SendEncodings]]`.

   7. If any of the following conditions are met, return a promise rejected with a newly created `InvalidModificationError`:

      - *`encodings`*`.length` is different from *N*.
      - *encodings* has been re-ordered.
      - Any parameter in *parameters* is marked as a **Read-only parameter** (such as RID) and has a value that is different from the corresponding parameter value in *sender*.`[[LastReturnedParameters]]`. Note that this also applies to *transactionId*.
      - CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

        Show Current and Future Show Current Show Future

        Any *encoding* in encodings contains a codec not found in *choosableCodecs*.

   8. **PROPOSED CORRECTION 20:** Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio (PR #2772, PR #2799)

      PROPOSED ADDITION 19: Add RTCRtpEncodingParameters.maxFramerate (PR #2785)

      Show Current and Future Show Current Show Future

If transceiver kind is "audio", remove the scaleResolutionDownBy and maxFramerate members from all *encodings* that contain any of them.

9. **PROPOSED CORRECTION 21:** Default RTCRtpEncodingParameters.scaleResolutionDownBy to 1 for video (PR #2772)

Show Current and Future Show Current Show Future

If transceiver kind is "video", then for each encoding in *encodings* that doesn't contain a scaleResolutionDownBy member, add a scaleResolutionDownBy member with the value 1.0.

PROPOSED CORRECTION 20: Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio (PR #2772, PR #2799)

Show Current and Future Show Current Show Future

10. ~~Verify that each~~ If transceiver kind is "video", and any encoding in *encodings* contains ~~has a~~ a scaleResolutionDownBy member whose value is ~~greater~~ less than ~~or equal to 1.0. If one of the scaleResolutionDownBy values does not meet this requirement~~1.0, return a promise rejected with a newly created RangeError.

11. **PROPOSED ADDITION 19:** Add RTCRtpEncodingParameters.maxFramerate (PR #2785)

Show Current and Future Show Current Show Future

Verify that each encoding in *encodings* has a maxFramerate member whose value is greater than or equal to 0.0. If one of the maxFramerate values does not meet this requirement, return a promise rejected with a newly created RangeError.

12. CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

Show Current and Future Show Current Show Future

If the user agent does not support setting the codec for any encoding or mixing different codec values on the different encodings, return a promise rejected with a newly created OperationError.

7. Let *p* be a new promise.
8. In parallel, configure the media stack to use *parameters* to transmit *sender*.[[SenderTrack]].
   1. If the media stack is successfully configured with *parameters*, queue a task to run the following steps:
      1. Set *sender*.[[LastReturnedParameters]] to null.
      2. Set *sender*.[[SendEncodings]] to *parameters*.encodings.
      3. Resolve *p* with undefined.
   2. If any error occurred while configuring the media stack, queue a task to run the following steps:
      1. If an error occurred due to hardware resources not being available, reject *p* with a newly created RTCError whose errorDetail is set to "hardware-encoder-not-available" and abort these steps.
      2. If an error occurred due to a hardware encoder not supporting *parameters*, reject *p* with a newly created RTCError whose errorDetail is set to "hardware-encoder-error" and abort these steps.
      3. For all other errors, reject *p* with a newly created OperationError.
9. Return *p*.

`setParameters` does not cause SDP renegotiation and can only be used to change what the media stack is sending or receiving within the envelope negotiated by Offer/Answer. The attributes in the `RTCRtpSendParameters` dictionary are designed to not enable this, so attributes like `cname` that cannot be changed are read-only. Other things, like bitrate, are controlled using limits such as `maxBitrate`, where the user agent needs to ensure it does not exceed the maximum bitrate specified by `maxBitrate`, while at the same time making sure it satisfies constraints on bitrate specified in other places such as the SDP.

getParameters

The `getParameters()` method returns the `RTCRtpSender` object's current parameters for how `track` is encoded and transmitted to a remote `RTCRtpReceiver`.

When `getParameters` is called, the user agent *MUST* run the following steps:

1. Let *sender* be the `RTCRtpSender` object on which the getter was invoked.
2. If *sender*.`[[LastReturnedParameters]]` is not `null`, return *sender*.`[[LastReturnedParameters]]`, and abort these steps.
3. Let *result* be a new `RTCRtpSendParameters` dictionary constructed as follows:
    - `transactionId` is set to a new unique identifier.
    - `encodings` is set to the value of the `[[SendEncodings]]` internal slot.
    - The `headerExtensions` sequence is populated based on the header extensions that have been negotiated for sending.
    - `codecs` is set to the value of the `[[SendCodecs]]` internal slot.
    - `rtcp`.`cname` is set to the CNAME of the associated `RTCPeerConnection`. `rtcp`.`reducedSize` is set to `true` if reduced-size RTCP has been negotiated for sending, and `false` otherwise.
4. Set *sender*.`[[LastReturnedParameters]]` to *result*.
5. Queue a task that sets *sender*.`[[LastReturnedParameters]]` to `null`.
6. Return *result*.

`getParameters` may be used with `setParameters` to change the parameters in the following way:

EXAMPLE 2

```
async function updateParameters() {
  try {
    const params = sender.getParameters();
    // ... make changes to parameters
    params.encodings[0].active = false;
    await sender.setParameters(params);
  } catch (err) {
    console.error(err);
  }
}
```

After a completed call to `setParameters`, subsequent calls to `getParameters` will return the modified set of parameters.

replaceTrack

Attempts to replace the `RTCRtpSender`'s current `track` with another track provided (or with a `null` track), without renegotiation.

When the `replaceTrack` method is invoked, the user agent *MUST* run the following steps:

1. Let *sender* be the `RTCRtpSender` object on which `replaceTrack` is invoked.
2. Let *transceiver* be the `RTCRtpTransceiver` object associated with *sender*.
3. Let *connection* be the `RTCPeerConnection` object associated with *sender*.
4. Let *withTrack* be the argument to this method.

5. If *withTrack* is non-null and `withTrack.kind` differs from the transceiver kind of *transceiver*, return a promise rejected with a newly created `TypeError`.

6. Return the result of chaining the following steps to *connection*'s operations chain:

PROPOSED CORRECTION 32: Reject setParameters(), replaceTrack(), & insertDTMF() after stop() (PR #2829)

◉ Show Current and Future ○ Show Current ○ Show Future

1. If *transceiver*.[[~~Stopped~~Stopping]] is `true`, return a promise rejected with a newly created `InvalidStateError`.

2. Let *p* be a new promise.

3. Let *sending* be `true` if *transceiver*.`[[CurrentDirection]]` is "`sendrecv`" or "`sendonly`", and `false` otherwise.

4. Run the following steps in parallel:

    1. If *sending* is `true`, and *withTrack* is `null`, have the sender stop sending.

    2. If *sending* is `true`, and *withTrack* is not `null`, determine if *withTrack* can be sent immediately by the sender without violating the sender's already-negotiated envelope, and if it cannot, then reject *p* with a newly created `InvalidModificationError`, and abort these steps.

    3. If *sending* is `true`, and *withTrack* is not `null`, have the sender switch seamlessly to transmitting *withTrack* instead of the sender's existing track.

    4. Queue a task that runs the following steps:

        1. If *connection*.`[[IsClosed]]` is `true`, abort these steps.

        2. Set *sender*.`[[SenderTrack]]` to *withTrack*.

        3. Resolve *p* with `undefined`.

5. Return *p*.

NOTE

Changing dimensions and/or frame rates might not require negotiation. Cases that may require negotiation include:

1. ***Changing a resolution to a value outside of the negotiated imageattr bounds, as described in [RFC6236].***
2. ***Changing a frame rate to a value that causes the block rate for the codec to be exceeded.***
3. ***A video track differing in raw vs. pre-encoded format.***
4. ***An audio track having a different number of channels.***
5. ***Sources that also encode (typically hardware encoders) might be unable to produce the negotiated codec; similarly, software sources might not implement the codec that was negotiated for an encoding source.***

`setStreams`

Sets the `MediaStream`s to be associated with this sender's track.

When the `setStreams` method is invoked, the user agent *MUST* run the following steps:

1. Let *sender* be the `RTCRtpSender` object on which this method was invoked.

2. Let *connection* be the `RTCPeerConnection` object on which this method was invoked.

3. If *connection*.`[[IsClosed]]` is `true`, throw an `InvalidStateError`.

4. Let *streams* be a list of `MediaStream` objects constructed from the method's arguments, or an

empty list if the method was called without arguments.

5. Set *sender*.`[[AssociatedMediaStreamIds]]` to an empty set.

6. For each *stream* in *streams*, add *stream.id* to `[[AssociatedMediaStreamIds]]` if it's not already there.

7. Update the negotiation-needed flag for *connection*.

`getStats`

Gathers stats for this sender only and reports the result asynchronously.

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

1. Let *selector* be the `RTCRtpSender` object on which the method was invoked.

2. Let *p* be a new promise, and run the following steps in parallel:

    1. Gather the stats indicated by *selector* according to the stats selection algorithm.

    2. Resolve *p* with the resulting `RTCStatsReport` object, containing the gathered stats.

3. Return *p*.

§ 5.2.1 `RTCRtpParameters` Dictionary

WebIDL

```
dictionary RTCRtpParameters {
  required sequence<RTCRtpHeaderExtensionParameters> headerExtensions;
  required RTCRtcpParameters rtcp;
  required sequence<RTCRtpCodecParameters> codecs;
};
```

§ *Dictionary `RTCRtpParameters` Members*

`headerExtensions` **of type sequence<`RTCRtpHeaderExtensionParameters`>, required**

A sequence containing parameters for RTP header extensions. Read-only parameter.

`rtcp` **of type `RTCRtcpParameters`, required**

Parameters used for RTCP. Read-only parameter.

`codecs` **of type sequence<`RTCRtpCodecParameters`>, required**

A sequence containing the media codecs that an `RTCRtpSender` will choose from, as well as entries for RTX, RED and FEC mechanisms. Corresponding to each media codec where retransmission via RTX is enabled, there will be an entry in `codecs` with a `mimeType` attribute indicating retransmission via `audio/rtx` or `video/rtx`, and an `sdpFmtpLine` attribute (providing the "apt" and "rtx-time" parameters). Read-only parameter.

§ 5.2.2 `RTCRtpSendParameters` Dictionary

WebIDL

```
dictionary RTCRtpSendParameters : RTCRtpParameters {
  required DOMString transactionId;
  required sequence<RTCRtpEncodingParameters> encodings;
};
```

§ *Dictionary `RTCRtpSendParameters` Members*

***`transactionId`* of type DOMString, required**
A unique identifier for the last set of parameters applied. Ensures that `setParameters` can only be called based on a previous `getParameters`, and that there are no intervening changes. Read-only parameter.

***`encodings`* of type sequence<`RTCRtpEncodingParameters`>, required**
A sequence containing parameters for RTP encodings of media.

§ 5.2.3 `RTCRtpReceiveParameters` Dictionary

WebIDL

```
dictionary RTCRtpReceiveParameters : RTCRtpParameters {
};
```

§ 5.2.4 `RTCRtpCodingParameters` Dictionary

WebIDL

```
dictionary RTCRtpCodingParameters {
  DOMString rid;
};
```

§ *Dictionary `RTCRtpCodingParameters` Members*

***`rid`* of type DOMString**
If set, this RTP encoding will be sent with the RID header extension as defined by [RFC9429] (section 5.2.1.). The RID is not modifiable via `setParameters`. It can only be set or modified in `addTransceiver` on the sending side. Read-only parameter.

PROPOSED CORRECTION 11: Remove unused RTCRtpDecodingParameters dictionary (PR #2753)

Show Current and Future Show Current Show Future

~~5.2.5 `RTCRtpDecodingParameters` Dictionary~~

```
dictionary RTCRtpDecodingParameters : RTCRtpCodingParameters {};
```

§ 5.2.5 `RTCRtpEncodingParameters` Dictionary

PROPOSED ADDITION 19: Add RTCRtpEncodingParameters.maxFramerate (PR #2785)

Show Current and Future Show Current Show Future

```
dictionary RTCRtpEncodingParameters : RTCRtpCodingParameters {
```

```
  boolean active = true;
  RTCRtpCodec codec;
  unsigned long maxBitrate;
  double maxFramerate;
  double scaleResolutionDownBy;
};
```

§ *Dictionary `RTCRtpEncodingParameters` Members*

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

Indicates that this encoding is actively being sent. Setting it to `false` causes this encoding to no longer be sent. Setting it to `true` causes this encoding to be sent. Since setting the value to `false` does not cause the SSRC to be removed, an RTCP BYE is not sent.

CANDIDATE ADDITION 49: Add codec to RTCRtpEncodingParameters (PR #2985)

Show Current and Future   Show Current   Show Future

`codec` **of type RTCRtpCodec**

Optional value selecting which codec is used for this encoding's RTP stream. If absent, the user agent can chose to use any negotiated codec.

`maxBitrate` **of type unsigned long**

When present, indicates the maximum bitrate that can be used to send this encoding. The user agent is free to allocate bandwidth between the encodings, as long as the `maxBitrate` value is not exceeded. The encoding may also be further constrained by other limits (such as per-transport or per-session bandwidth limits) below the maximum specified here. `maxBitrate` is computed the same way as the Transport Independent Application Specific Maximum (TIAS) bandwidth defined in [RFC3890] Section 6.2.2, which is the maximum bandwidth needed without counting IP or other transport layers like TCP or UDP. The unit of `maxBitrate` is bits per second.

> ***NOTE***
>
> ***How the bitrate is achieved is media and encoding dependent. For video, a frame will always be sent as fast as possible, but frames may be dropped until bitrate is low enough. Thus, even a bitrate of zero will allow sending one frame. For audio, it might be necessary to stop playing if the bitrate does not allow the chosen encoding enough bandwidth to be sent.***

`maxFramerate` **of type double**

This member can only be present if the sender's `kind` is `"video"`. When present, indicates the maximum frame rate that can be used to send this encoding, in frames per second. The user agent is free to allocate bandwidth between the encodings, as long as the `maxFramerate` value is not exceeded.

If changed with `setParameters()`, the new frame rate takes effect after the current picture is completed; setting the max frame rate to zero thus has the effect of freezing the video on the next frame.

`scaleResolutionDownBy` **of type double**

This member is only present if the sender's `kind` is `"video"`. The video's resolution will be scaled down in each dimension by the given value before sending. For example, if the value is 2.0, the video will be scaled down by a factor of 2 in each dimension, resulting in sending a video of one quarter the size. If the value is 1.0, the video will not be affected. The value must be greater than or equal to 1.0. By default, scaling is applied in reverse order by a factor of two, to produce an order of smaller to higher resolutions, e.g. 4:2:1. If there is only one layer, the sender will by default not apply any scaling, (i.e. `scaleResolutionDownBy` will be 1.0).

§ 5.2.6 `RTCRtcpParameters` Dictionary

WebIDL

```
dictionary RTCRtcpParameters {
  DOMString cname;
  boolean reducedSize;
};
```

§ *Dictionary `RTCRtcpParameters` Members*

***`cname`* of type DOMString**
: The Canonical Name (CNAME) used by RTCP (e.g. in SDES messages). Read-only parameter.

***`reducedSize`* of type boolean**
: Whether reduced size RTCP [RFC5506] is configured (if true) or compound RTCP as specified in [RFC3550] (if false). Read-only parameter.

§ 5.2.7 `RTCRtpHeaderExtensionParameters` Dictionary

WebIDL

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

§ *Dictionary `RTCRtpHeaderExtensionParameters` Members*

***`uri`* of type DOMString, required**
: The URI of the RTP header extension, as defined in [RFC5285]. Read-only parameter.

***`id`* of type unsigned short, required**
: The value put in the RTP packet to identify the header extension. Read-only parameter.

***`encrypted`* of type boolean**
: Whether the header extension is encrypted or not. Read-only parameter.

> ***NOTE***
> ***The `RTCRtpHeaderExtensionParameters` dictionary enables an application to determine whether a header extension is configured for use within an `RTCRtpSender` or `RTCRtpReceiver`. For an `RTCRtpTransceiver` transceiver, an application can determine the "direction" parameter (defined in Section 5 of [RFC5285]) of a header extension as follows without having to parse SDP:***
>
> 1. ***sendonly: The header extension is only included in transceiver.`sender`.`getParameters`().`headerExtensions`.***
> 2. ***recvonly: The header extension is only included in transceiver.`receiver`.`getParameters`().`headerExtensions`.***
> 3. ***sendrecv: The header extension is included in both transceiver.`sender`.`getParameters`().`headerExtensions` and transceiver.`receiver`.`getParameters`().`headerExtensions`.***
> 4. ***inactive: The header extension is included in neither transceiver.`sender`.`getParameters`().`headerExtensions` nor transceiver.`receiver`.`getParameters`().`headerExtensions`.***

§ 5.2.8 `RTCRtpCodec` Dictionary

WebIDL

```
dictionary RTCRtpCodec {
  required DOMString mimeType;
  required unsigned long clockRate;
  unsigned short channels;
  DOMString sdpFmtpLine;
};
```

§ *Dictionary `RTCRtpCodec` Members*

The `RTCRtpCodec` dictionary provides information about codec objects.

`mimeType` **of type DOMString, required**
: The codec MIME media type/subtype. Valid media types and subtypes are listed in [IANA-RTP-2].

`clockRate` **of type unsigned long, required**
: The codec clock rate expressed in Hertz.

`channels` **of type unsigned short**
: If present, indicates the maximum number of channels (mono=1, stereo=2).

`sdpFmtpLine` **of type DOMString**
: The "format specific parameters" field from the `a=fmtp` line in the SDP corresponding to the codec, if one exists, as defined by [RFC9429] (section 5.8.).

PROPOSED CORRECTION 29: Create RTCRtpCodec dictionary and reuse in RTCRtpCodecCapability and RTCRtpCodecParameters definitions (PR #2834)

◉ Show Current and Future ○ Show Current ○ Show Future

~~5.2.9 `RTCRtpCodecParameters` Dictionary~~

```
dictionary RTCRtpCodecParameters {
  required octet payloadType;
  required DOMString mimeType;
  required unsigned long clockRate;
  unsigned short channels;
  DOMString sdpFmtpLine;
};
```

~~*Dictionary `RTCRtpCodecParameters` Members*~~

~~***`payloadType`*** **of type octet, required**~~
: ~~The RTP payload type used to identify this codec. Read-only parameter.~~

~~***`mimeType`*** **of type DOMString, required**~~
: ~~The codec MIME media type/subtype. Valid media types and subtypes are listed in [IANA-RTP-2]. Read-only parameter.~~

~~***`clockRate`*** **of type unsigned long, required**~~
: ~~The codec clock rate expressed in Hertz. Read-only parameter.~~

~~***`channels`*** **of type unsigned short**~~
: ~~When present, indicates the number of channels (mono=1, stereo=2). Read-only parameter.~~

~~***`sdpFmtpLine`*** **of type DOMString**~~

~~The "format specific parameters" field from the `a=fmtp` line in the SDP corresponding to the codec, if one exists, as defined by [RFC8829] (section 5.8.). For an `RTCRtpSender`, these parameters come from the remote description, and for an `RTCRtpReceiver`, they come from the local description. Read-only parameter.~~

§ 5.2.9 `RTCRtpCodecParameters` Dictionary

```
dictionary RTCRtpCodecParameters : RTCRtpCodec {
  required octet payloadType;
};
```

§ *Dictionary `RTCRtpCodecParameters` Members*

The `RTCRtpCodecParameters` dictionary provides information about the negotiated codecs. The fields inherited from `RTCRtpCodec` *MUST* all be Read-only parameters.

For an `RTCRtpSender`, the `sdpFmtpLine` parameters come from the `[[CurrentRemoteDescription]]`, and for an `RTCRtpReceiver`, they come from the local description (which is `[[PendingLocalDescription]]` if not `null`, and `[[CurrentLocalDescription]]` otherwise).

`payloadType` **of type octet, required**
: The RTP payload type used to identify this codec. Read-only parameter.

§ 5.2.10 `RTCRtpCapabilities` Dictionary

WebIDL

```
dictionary RTCRtpCapabilities {
  required sequence<RTCRtpCodec> codecs;
  required sequence<RTCRtpHeaderExtensionCapability> headerExtensions;
};
```

§ *Dictionary `RTCRtpCapabilities` Members*

`codecs` **of type sequence<`RTCRtpCodec`>, required**
: Supported media codecs as well as entries for RTX, RED and FEC mechanisms. Only combinations that would utilize distinct payload types in a generated SDP offer are to be provided. For example:

1. Two H.264/AVC codecs, one for each of two supported packetization-mode values.
2. Two CN codecs with different clock rates.

There *MUST* only be a single entry in `codecs` for retransmission via RTX, with `sdpFmtpLine` not present.

`headerExtensions` **of type sequence<`RTCRtpHeaderExtensionCapability`>, required**
: Supported RTP header extensions.

§ 5.2.11 `RTCRtpHeaderExtensionCapability` Dictionary

PROPOSED CORRECTION 30: Make RTCRtpHeaderExtensionCapability.uri required (PR #2841)

◉ Show Current and Future ○ Show Current ○ Show Future

```
dictionary RTCRtpHeaderExtensionCapability {
  required DOMString uri;
};
```

§ *Dictionary* `RTCRtpHeaderExtensionCapability` *Members*

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

PROPOSED ADDITION 36: Add empty setParameterOptions as second argument to setParameters for extensibility (PR #2885)

◉ Show Current and Future ○ Show Current ○ Show Future

§ 5.2.12 `RTCSetParameterOptions` Dictionary

WebIDL

```
dictionary RTCSetParameterOptions {
};
```

§ *Dictionary* `RTCSetParameterOptions` *Members*

RTCSetParameterOptions is defined as an empty dictionary to allow for extensibility.

§ 5.3 `RTCRtpReceiver` Interface

The `RTCRtpReceiver` interface allows an application to inspect the receipt of a `MediaStreamTrack`.

To **create an RTCRtpReceiver** with a string, *kind*, run the following steps:

1. Let *receiver* be a new `RTCRtpReceiver` object.
2. Let *track* be a new `MediaStreamTrack` object [GETUSERMEDIA]. The source of *track* is a **remote source** provided by *receiver*. Note that the *track*.`id` is generated by the user agent and does not map to any track IDs on the remote side.
3. Initialize *track.kind* to *kind*.
4. Initialize *track.label* to the result of concatenating the string `"remote "` with *kind*.
5. Initialize *track.readyState* to `live`.
6. Initialize *track.muted* to `true`. See the MediaStreamTrack section about how the `muted` attribute reflects if a `MediaStreamTrack` is receiving media data or not.
7. Let *receiver* have a **[[ReceiverTrack]]** internal slot initialized to *track*.
8. Let *receiver* have a **[[ReceiverTransport]]** internal slot initialized to `null`.
9. Let *receiver* have a **[[LastStableStateReceiverTransport]]** internal slot initialized to `null`.
10. Let *receiver* have an **[[AssociatedRemoteMediaStreams]]** internal slot, representing a list of

`MediaStream` objects that the `MediaStreamTrack` object of this receiver is associated with, and initialized to an empty list.

11. Let *receiver* have a **[[LastStableStateAssociatedRemoteMediaStreams]]** internal slot and initialize it to an empty list.

12. Let *receiver* have a **[[ReceiveCodecs]]** internal slot, representing a list of `RTCRtpCodecParameters` dictionaries, and initialized to an empty list.

13. Let *receiver* have a **[[LastStableStateReceiveCodecs]]** internal slot and initialize it to an empty list.

14. **PROPOSED ADDITION 44:** Add control for the receiver's jitter buffer (PR #2953)

◉ Show Current and Future ○ Show Current ○ Show Future

Let *receiver* have a **[[JitterBufferTarget]]** internal slot initialized to `null`.

15. Return *receiver*.

PROPOSED ADDITION 44: Add control for the receiver's jitter buffer (PR #2953)

◉ Show Current and Future ○ Show Current ○ Show Future

```
[Exposed=Window]
interface RTCRtpReceiver {
  readonly attribute MediaStreamTrack track;
  readonly attribute RTCDtlsTransport? transport;
  static RTCRtpCapabilities? getCapabilities(DOMString kind);
  RTCRtpReceiveParameters getParameters();
  sequence<RTCRtpContributingSource> getContributingSources();
  sequence<RTCRtpSynchronizationSource> getSynchronizationSources();
  Promise<RTCStatsReport> getStats();
  attribute DOMHighResTimeStamp? jitterBufferTarget;
};
```

§ Attributes

`track` **of type `MediaStreamTrack`, readonly**

The `track` attribute is the track that is associated with this `RTCRtpReceiver` object *receiver*.

Note that `track.stop()` is final, although clones are not affected. Since *receiver*.`track.stop()` does not implicitly stop *receiver*, Receiver Reports continue to be sent. On getting, the attribute *MUST* return the value of the `[[ReceiverTrack]]` slot.

`transport` **of type `RTCDtlsTransport`, readonly, nullable**

The `transport` attribute is the transport over which media for the receiver's `track` is received in the form of RTP packets. Prior to construction of the `RTCDtlsTransport` object, the `transport` attribute will be `null`. When bundling is used, multiple `RTCRtpReceiver` objects will share one `transport` and will all receive RTP and RTCP over the same transport.

On getting, the attribute *MUST* return the value of the `[[ReceiverTransport]]` slot.

PROPOSED ADDITION 44: Add control for the receiver's jitter buffer (PR #2953)

◉ Show Current and Future ○ Show Current ○ Show Future

`jitterBufferTarget` **of type `DOMHighResTimeStamp`, nullable**

This attribute allows the application to specify a target duration of time in milliseconds of media for the `RTCRtpReceiver`'s jitter buffer to hold. This influences the amount of buffering done by the user agent, which in turn affects retransmissions and packet loss recovery. Altering the target value allows applications to control the tradeoff between playout delay and the risk of running out of audio or video frames due to network jitter.

The user agent *MUST* have a **minimum allowed target** and a **maximum allowed target** reflecting what the user agent is able or willing to provide based on network conditions and memory constraints, which can change at any time.

> ***NOTE***
> ***This is a target value. The resulting change in delay can be gradually observed over time. The receiver's average jitter buffer delay can be measured as the delta `jitterBufferDelay` divided by the delta `jitterBufferEmittedCount`.***
>
> ***An average delay is expected even if DTX is used. For example, if DTX is used and packets start flowing after silence, larger targets can influence the user agent to buffer these packets rather than playing them out.***

On getting, this attribute *MUST* return the value of the `[[JitterBufferTarget]]` internal slot.

On setting, the user agent *MUST* run the following steps:

1. Let *receiver* be the `RTCRtpReceiver` object on which the setter is invoked.
2. Let *target* be the argument to the setter.
3. If *target* is negative or larger than 4000 milliseconds, then throw a `RangeError`.
4. Set *receiver*'s `[[JitterBufferTarget]]` to *target*.
5. Let *track* be *receiver*'s `[[ReceiverTrack]]`.
6. in parallel, begin executing the following steps:

   1. Update the underlying system about the new *target*, or that there is no application preference if *target* is `null`.

      If *track* is synchronized with another `RTCRtpReceiver`'s track for audio/video synchronization, then the user agent *SHOULD* use the larger of the two receivers' `[[JitterBufferTarget]]` for both receivers.

      When the underlying system is applying a jitter buffer target, it will continuously make sure that the actual jitter buffer target is clamped within the minimum allowed target and maximum allowed target.

      > ***NOTE***
      >
      > ***If the user agent ends up using a target different from the requested one (e.g. due to network conditions or physical memory constraints), this is not reflected in the `[[JitterBufferTarget]]` internal slot.***

   2. Modifying the jitter buffer target of the underlying system *SHOULD* affect the internal audio or video buffering gradually in order not to hurt user experience. Audio samples or video frames *SHOULD* be accelerated or decelerated before playout, similarly to how it is done for audio/video synchronization or in response to congestion control.

      The acceleration or deceleration rate may vary depending on network conditions or the type of audio received (e.g. speech or background noise). It *MAY* take several seconds to achieve 1 second of buffering but *SHOULD* not take more than 30 seconds assuming packets are being received. The speed *MAY* be different for audio and video.

      > ***NOTE***
      >
      > ***For audio, acceleration and deceleration can be measured with `insertedSamplesForDeceleration` and `removedSamplesForAcceleration`. For video, this may result in the same frame being rendered multiple times or frames may be dropped.***

§ Methods

***getCapabilities*, static**

The static `RTCRtpReceiver`.`getCapabilities()` method provides a way to discover the types of capabilities the user agent supports for receiving media of the given kind, without reserving any resources, ports, or other state.

When the `getCapabilities` method is called, the user agent *MUST* run the following steps:

1. Let *kind* be the method's first argument.
2. If *kind* is neither `"video"` nor `"audio"` return `null`.
3. Return a new `RTCRtpCapabilities` dictionary, with its `codecs` member initialized to the list of implemented receive codecs for *kind*, and its `headerExtensions` member initialized to the list of implemented header extensions for receiving for *kind*.

The **list of implemented receive codecs**, given *kind*, is an implementation-defined list of `RTCRtpCodec` dictionaries representing the most optimistic view of the codecs the user agent supports for receiving media of the given *kind* (video or audio).

The **list of implemented header extensions for receiving**, given *kind*, is an implementation-defined list of `RTCRtpHeaderExtensionCapability` dictionaries representing an optimistic view of the header extensions the user agent supports for receiving media of the given *kind* (video or audio).

These capabilities provide generally persistent cross-origin information on the device and thus increases the fingerprinting surface of the application. In privacy-sensitive contexts, user agents *MAY* consider mitigations such as reporting only a common subset of the capabilities.

> ***NOTE***
> ***The codec capabilities returned affect the `setCodecPreferences()` algorithm and what inputs it throws `InvalidModificationError` on, and should also be consistent with information revealed by `createOffer()` and `createAnswer()` about codecs negotiated for reception, to ensure any privacy mitigations are effective.***

getParameters

The `getParameters()` method returns the `RTCRtpReceiver` object's current parameters for how `track` is decoded.

When `getParameters` is called, the `RTCRtpReceiveParameters` dictionary is constructed as follows:

- The `headerExtensions` sequence is populated based on the header extensions that the receiver is currently prepared to receive.
- `codecs` is set to the value of the `[[ReceiveCodecs]]` internal slot.

> ***NOTE***
> ***Both the local and remote description may affect this list of codecs. For example, if three codecs are offered, the receiver will be prepared to receive each of them and will return them all from `getParameters`. But if the remote endpoint only answers with two, the absent codec will no longer be returned by `getParameters` as the receiver no longer needs to be prepared to receive it.***

- `rtcp`.`reducedSize` is set to `true` if the receiver is currently prepared to receive reduced-size RTCP packets, and `false` otherwise. `rtcp`.`cname` is left out.

getContributingSources

Returns an `RTCRtpContributingSource` for each unique CSRC identifier received by this `RTCRtpReceiver` in the last 10 seconds, in descending `timestamp` order.

getSynchronizationSources

Returns an `RTCRtpSynchronizationSource` for each unique SSRC identifier received by this `RTCRtpReceiver` in the last 10 seconds, in descending `timestamp` order.

getStats

Gathers stats for this receiver only and reports the result asynchronously.

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

1. Let *selector* be the `RTCRtpReceiver` object on which the method was invoked.
2. Let *p* be a new promise, and run the following steps in parallel:
    1. Gather the stats indicated by *selector* according to the stats selection algorithm.
    2. Resolve *p* with the resulting `RTCStatsReport` object, containing the gathered stats.
3. Return *p*.

The **`RTCRtpContributingSource`** and **`RTCRtpSynchronizationSource`** dictionaries contain information about a given contributing source (CSRC) or synchronization source (SSRC) respectively. When an audio or video frame from one or more RTP packets is delivered to the `RTCRtpReceiver`'s `MediaStreamTrack`, the user agent *MUST* queue a task to update the relevant information for the `RTCRtpContributingSource` and `RTCRtpSynchronizationSource` dictionaries based on the content of those packets. The information relevant to the `RTCRtpSynchronizationSource` dictionary corresponding to the SSRC identifier, is updated each time, and if an RTP packet contains CSRC identifiers, then the information relevant to the `RTCRtpContributingSource` dictionaries corresponding to those CSRC identifiers is also updated. The user agent *MUST* process RTP packets in order of ascending RTP timestamps. The user agent *MUST* keep information from RTP packets delivered to the `RTCRtpReceiver`'s `MediaStreamTrack` in the previous 10 seconds.

> ***NOTE***
> ***Even if the `MediaStreamTrack` is not attached to any sink for playout, `getSynchronizationSources` and `getContributingSources` returns up-to-date information as long as the track is not ended; sinks are not a prerequisite for decoding RTP packets.***

> ***NOTE***
> ***As stated in the conformance section, requirements phrased as algorithms may be implemented in any manner so long as the end result is equivalent. So, an implementation does not need to literally queue a task for every frame, as long as the end result is that within a single event loop task execution, all returned `RTCRtpSynchronizationSource` and `RTCRtpContributingSource` dictionaries for a particular `RTCRtpReceiver` contain information from a single point in the RTP stream.***

WebIDL

```
dictionary RTCRtpContributingSource {
  required DOMHighResTimeStamp timestamp;
  required unsigned long source;
  double audioLevel;
  required unsigned long rtpTimestamp;
};
```

§ Dictionary RTCRtpContributingSource Members

`timestamp` **of type `DOMHighResTimeStamp`, required**

The `timestamp` indicating the most recent time a frame from an RTP packet, originating from this source, was delivered to the `RTCRtpReceiver`'s `MediaStreamTrack`. The `timestamp` is defined as `Performance.timeOrigin` + `Performance.now()` at that time.

`source` **of type unsigned long, required**

The CSRC or SSRC identifier of the contributing or synchronization source.

`audioLevel` **of type double**

Only present for audio receivers. This is a value between 0..1 (linear), where 1.0 represents 0 dBov, 0 represents silence, and 0.5 represents approximately 6 dBSPL change in the sound pressure level from 0 dBov.

For CSRCs, this *MUST* be converted from the level value defined in [RFC6465] if the RFC 6465 header extension is present, otherwise this member *MUST* be absent.

For SSRCs, this *MUST* be converted from the level value defined in [RFC6464]. If the RFC 6464 header extension is not present in the received packets (such as if the other endpoint is not a user agent or is a legacy endpoint), this value *SHOULD* be absent.

Both RFCs define the level as an integral value from 0 to 127 representing the audio level in negative decibels relative to the loudest signal that the system could possibly encode. Thus, 0 represents the loudest signal the system could possibly encode, and 127 represents silence.

To convert these values to the linear 0..1 range, a value of 127 is converted to 0, and all other values are converted using the equation: `10^(-rfc_level/20)`.

`rtpTimestamp` **of type unsigned long, required**
The RTP timestamp, as defined in [RFC3550] Section 5.1, of the media played out at *timestamp*.

WebIDL

```
dictionary RTCRtpSynchronizationSource : RTCRtpContributingSource {};
```

The `RTCRtpSynchronizationSource` dictionary is expected to serve as an extension point for the specification to surface data only available in SSRCs.

§ 5.4 `RTCRtpTransceiver` Interface

The `RTCRtpTransceiver` interface represents a combination of an `RTCRtpSender` and an `RTCRtpReceiver` that share a common media stream "identification-tag". As defined in [RFC9429] (section 3.4.1.), an `RTCRtpTransceiver` is said to be **associated** with a media description if its "mid" property is non-null and matches a media stream "identification-tag" in the media description; otherwise it is said to be disassociated with that media description.

> ***NOTE***
> ***A `RTCRtpTransceiver` may become associated with a new pending description in RFC9429 while still being disassociated with the current description. This may happen in check if negotiation is needed.***

The **transceiver kind** of an `RTCRtpTransceiver` is defined by the kind of the associated `RTCRtpReceiver`'s `MediaStreamTrack` object.

To **create an RTCRtpTransceiver** with an `RTCRtpReceiver` object, *receiver*, `RTCRtpSender` object, *sender*, and an `RTCRtpTransceiverDirection` value, *direction*, run the following steps:

1. Let *transceiver* be a new `RTCRtpTransceiver` object.
2. Let *transceiver* have a **[[Sender]]** internal slot, initialized to *sender*.
3. Let *transceiver* have a **[[Receiver]]** internal slot, initialized to *receiver*.
4. Let *transceiver* have a **[[Stopping]]** internal slot, initialized to `false`.
5. Let *transceiver* have a **[[Stopped]]** internal slot, initialized to `false`.
6. Let *transceiver* have a **[[Direction]]** internal slot, initialized to *direction*.
7. Let *transceiver* have a **[[Receptive]]** internal slot, initialized to `false`.
8. Let *transceiver* have a **[[CurrentDirection]]** internal slot, initialized to `null`.
9. Let *transceiver* have a **[[FiredDirection]]** internal slot, initialized to `null`.
10. Let *transceiver* have a **[[PreferredCodecs]]** internal slot, initialized to an empty list.
11. Let *transceiver* have a **[[JsepMid]]** internal slot, initialized to `null`. This is the "RtpTransceiver mid property" defined in [RFC9429] (section 5.2.1. and section 5.3.1.), and is only modified there.

12. Let *transceiver* have a **[[Mid]]** internal slot, initialized to `null`.

13. Return *transceiver*.

> ***NOTE***
> ***Creating a transceiver does not create the underlying `RTCDtlsTransport` and `RTCIceTransport` objects. This will only occur as part of the process of setting a session description.***

WebIDL

```
[Exposed=Window]
interface RTCRtpTransceiver {
  readonly attribute DOMString? mid;
  [SameObject] readonly attribute RTCRtpSender sender;
  [SameObject] readonly attribute RTCRtpReceiver receiver;
  attribute RTCRtpTransceiverDirection direction;
  readonly attribute RTCRtpTransceiverDirection? currentDirection;
  undefined stop();
  undefined setCodecPreferences(sequence<RTCRtpCodec> codecs);
};
```

§ Attributes

`mid` **of type DOMString, readonly, nullable**

The `mid` attribute is the media stream "identification-tag" negotiated and present in the local and remote descriptions. On getting, the attribute *MUST* return the value of the `[[Mid]]` slot.

`sender` **of type `RTCRtpSender`, readonly**

The `sender` attribute exposes the `RTCRtpSender` corresponding to the RTP media that may be sent with mid = `[[Mid]]`. On getting, the attribute *MUST* return the value of the `[[Sender]]` slot.

`receiver` **of type `RTCRtpReceiver`, readonly**

The `receiver` attribute is the `RTCRtpReceiver` corresponding to the RTP media that may be received with mid = `[[Mid]]`. On getting the attribute *MUST* return the value of the `[[Receiver]]` slot.

`direction` **of type `RTCRtpTransceiverDirection`**

As defined in [RFC9429] (section 4.2.4.), the *direction* attribute indicates the preferred direction of this transceiver, which will be used in calls to `createOffer` and `createAnswer`. An update of directionality does not take effect immediately. Instead, future calls to `createOffer` and `createAnswer` mark the corresponding media description as `sendrecv`, `sendonly`, `recvonly` or `inactive` as defined in [RFC9429] (section 5.2.2. and section 5.3.2.)

On getting, the user agent *MUST* run the following steps:

1. Let *transceiver* be the `RTCRtpTransceiver` object on which the getter is invoked.

2. If *transceiver*.`[[Stopping]]` is `true`, return "`stopped`".

3. Otherwise, return the value of the `[[Direction]]` slot.

On setting, the user agent *MUST* run the following steps:

1. Let *transceiver* be the `RTCRtpTransceiver` object on which the setter is invoked.

2. Let *connection* be the `RTCPeerConnection` object associated with *transceiver*.

3. If *transceiver*.`[[Stopping]]` is `true`, throw an `InvalidStateError`.

4. Let *newDirection* be the argument to the setter.

5. If *newDirection* is equal to *transceiver*.`[[Direction]]`, abort these steps.

6. If *newDirection* is equal to "`stopped`", throw a `TypeError`.

7. Set *transceiver*.`[[Direction]]` to *newDirection*.

8. Update the negotiation-needed flag for *connection*.

currentDirection **of type `RTCRtpTransceiverDirection`, readonly, nullable**

As defined in [RFC9429] (section 4.2.5.), the *currentDirection* attribute indicates the current direction negotiated for this transceiver. The value of *currentDirection* is independent of the value of `RTCRtpEncodingParameters`.`active` since one cannot be deduced from the other. If this transceiver has never been represented in an offer/answer exchange, the value is `null`. If the transceiver is `stopped`, the value is "`stopped`".

On getting, the user agent *MUST* run the following steps:

1. Let *transceiver* be the `RTCRtpTransceiver` object on which the getter is invoked.
2. If *transceiver*.`[[Stopped]]` is `true`, return "`stopped`".
3. Otherwise, return the value of the `[[CurrentDirection]]` slot.

§ Methods

stop

Irreversibly marks the transceiver as `stopping`, unless it is already `stopped`. This will immediately cause the transceiver's sender to no longer send, and its receiver to no longer receive. Calling `stop()` also updates the negotiation-needed flag for the `RTCRtpTransceiver`'s associated `RTCPeerConnection`.

A **stopping** transceiver will cause future calls to `createOffer` to generate a zero port in the media description for the corresponding transceiver, as defined in [RFC9429] (section 4.2.1.) (The user agent *MUST* treat a `stopping` transceiver as `stopped` for the purposes of RFC9429 only in this case). However, to avoid problems with [RFC8843], a transceiver that is `stopping`, but not `stopped`, will not affect `createAnswer`.

A **stopped** transceiver will cause future calls to `createOffer` or `createAnswer` to generate a zero port in the media description for the corresponding transceiver, as defined in [RFC9429] (section 4.2.1.).

The transceiver will remain in the `stopping` state, unless it becomes `stopped` by `setRemoteDescription` processing a rejected m-line in a remote offer or answer.

> ***NOTE***
>
> ***A transceiver that is `stopping` but not `stopped` will always need negotiation. In practice, this means that calling `stop()` on a transceiver will cause the transceiver to become `stopped` eventually, provided negotiation is allowed to complete on both ends.***

When the `stop` method is invoked, the user agent *MUST* run the following steps:

1. Let *transceiver* be the `RTCRtpTransceiver` object on which the method is invoked.
2. Let *connection* be the `RTCPeerConnection` object associated with *transceiver*.
3. If *connection*.`[[IsClosed]]` is `true`, throw an `InvalidStateError`.
4. If *transceiver*.`[[Stopping]]` is `true`, abort these steps.
5. Stop sending and receiving with *transceiver*.
6. Update the negotiation-needed flag for *connection*.

The **stop sending and receiving** algorithm given a *transceiver* and, optionally, a *disappear* boolean defaulting to `false`, is as follows:

1. Let *sender* be *transceiver*.`[[Sender]]`.
2. Let *receiver* be *transceiver*.`[[Receiver]]`.
3. In parallel, stop sending media with *sender*, and send an RTCP BYE for each RTP stream that was being sent by *sender*, as specified in [RFC3550].
4. In parallel, stop receiving media with *receiver*.
5. If *disappear* is `false`, execute the steps for *receiver*.`[[ReceiverTrack]]` to be ended. This fires

an event.

6. Set *transceiver*.`[[Direction]]` to "`inactive`".

7. Set *transceiver*.`[[Stopping]]` to `true`.

The **stop the RTCRtpTransceiver** algorithm given a *transceiver* and, optionally, a *disappear* boolean defaulting to `false`, is as follows:

1. If *transceiver*.`[[Stopping]]` is `false`, stop sending and receiving with *transceiver* and *disappear*.

2. Set *transceiver*.`[[Stopped]]` to `true`.

3. Set *transceiver*.`[[Receptive]]` to `false`.

4. Set *transceiver*.`[[CurrentDirection]]` to `null`.

`setCodecPreferences`

CANDIDATE CORRECTION 42: setCodecPreferences only takes into account receive codecs (PR #2926)

Show Current and Future Show Current Show Future

The `setCodecPreferences` method overrides the default receive codec preferences used by the user agent. When generating a session description using either `createOffer` or `createAnswer`, the user agent *MUST* use the indicated codecs, in the order specified in the *codecs* argument, for the media section corresponding to this `RTCRtpTransceiver`.

This method allows applications to disable the negotiation of specific codecs (including RTX/RED/FEC). It also allows an application to cause a remote peer to prefer the codec that appears first in the list for sending.

Codec preferences remain in effect for all calls to `createOffer` and `createAnswer` that include this `RTCRtpTransceiver` until this method is called again. Setting *codecs* to an empty sequence resets codec preferences to any default value.

> **NOTE**
>
> ***Codecs have their payload types listed under each m= section in the SDP, defining the mapping between payload types and codecs. These payload types are referenced by the m=video or m=audio lines in the order of preference, and codecs that are not negotiated do not appear in this list as defined in section 5.2.1 of [~~RFC8829~~RFC9429]. A previously negotiated codec that is subsequently removed disappears from the m=video or m=audio line, and while its codec payload type is not to be reused in future offers or answers, its payload type may also be removed from the mapping of payload types in the SDP.***

~~The *codecs* sequence passed into~~ `setCodecPreferences` ~~can only contain codecs that are returned by `RTCRtpSender.getCapabilities(`*kind*`)`~~will reject attempts to set *codecs* ~~or~~ not matching codecs found in `RTCRtpReceiver.getCapabilities`(*kind*), where *kind* is the kind of the `RTCRtpTransceiver` on which the method is called. ~~Additionally, the `RTCRtpCodecCapability` dictionary members cannot be modified. If *codecs* does not fulfill these requirements, the user agent *MUST* throw an `InvalidModificationError`.~~

> **NOTE**
>
> ~~***Due to a recommendation in [SDP], calls to `createAnswer` SHOULD use only the common subset of the codec preferences and the codecs that appear in the offer. For example, if codec preferences are "C, B, A", but only codecs "A, B" were offered, the answer should only contain codecs "B, A". However, [RFC8829] (section 5.3.1.) allows adding codecs that were not in the offer, so implementations can behave differently.***~~

When `setCodecPreferences()` ~~in~~ is invoked, the user agent *MUST* run the following steps:

1. Let *transceiver* be the `RTCRtpTransceiver` object this method was invoked on.
2. Let *codecs* be the first argument.
3. If *codecs* is an empty list, set *transceiver*.`[[PreferredCodecs]]` to *codecs* and abort these steps.
4. ~~Remove any duplicate values in *codecs*. Start at the back of the list such that the priority of the codecs is maintained; the index of the first occurrence of a codec within the list is the same before and after this step.~~
5. Remove any duplicate values in *codecs*, ensuring that the first occurrence of each value remains in place.
6. Let *kind* be the *transceiver*'s transceiver kind.
7. ~~If the intersection between *codecs* and `RTCRtpSender.getCapabilities`(*kind*).`codecs` or the intersection between *codecs* and `RTCRtpReceiver.getCapabilities`(*kind*).`codecs` only contains RTX, RED or FEC codecs or is an empty set, throw `InvalidModificationError`. This ensures that we always have something to offer, regardless of *transceiver*.`direction`.~~
8. Let *codecCapabilities* ~~be the union of `RTCRtpSender.getCapabilities`(*kind*).`codecs` and~~ be `RTCRtpReceiver.getCapabilities`(*kind*).`codecs`.
9. ~~For each *codec* in *codecs*,~~
   1. ~~If *codec* is not in *codecCapabilities*, throw `InvalidModificationError`.~~
10. For each *codec* in *codecs*,
    1. If *codec* does not match any codec in *codecCapabilities*, throw `InvalidModificationError`.
11. If *codecs* only contains entries for RTX, RED, FEC or Comfort Noise or is an empty set, throw `InvalidModificationError`. This ensures that we always have something to offer, regardless of *transceiver*.`direction`.
12. Set *transceiver*.`[[PreferredCodecs]]` to *codecs*.

PROPOSED CORRECTION 31: Fix ambiguities in the setCodecPreferences() algorithm (PR #2847)

PROPOSED CORRECTION 47: setCodecPreferences() must use case-insensitive mimeType comparison (PR #2975)

◉ Show Current and Future ○ Show Current ○ Show Future

The **codec dictionary match** algorithm given two `RTCRtpCodec` dictionaries *first* and *second* is as follows:

1. If *first*.`mimeType` is not an ASCII case-insensitive match for *second*.`mimeType`, return `false`.
2. If *first*.`clockRate` is different from *second*.`clockRate`, return `false`.
3. If either (but not both) of *first*.`channels` and *second*.`channels` are missing, or if they both exist and *first*.`channels` is different from *second*.`channels`, return `false`.
4. If either (but not both) of *first*.`sdpFmtpLine` and *second*.`sdpFmtpLine` are missing, or if they both exist and *first*.`sdpFmtpLine` is different from *second*.`sdpFmtpLine`, return `false`.
5. Return `true`.

NOTE

If set, the offerer's receive codec preferences will decide the order of the codecs in the offer. If the answerer does not have any codec preferences then the same order will be used in the answer. However, if the answerer also has codec preferences, these preferences override the order in the answer. In this case, the offerer's preferences would affect which codecs were on offer but not the final order.

§ 5.4.1 Simulcast functionality

PROPOSED CORRECTION 15: Clarify simulcast envelope is determined by negotiation (PR #2760)

PROPOSED CORRECTION 28: Update explanation of simulcast envelope. (PR #2814)

Show Current and Future Show Current Show Future

Simulcast sending functionality is ~~provided via~~ enabled by the addTransceiver method ~~of~~ via its sendEncodings argument, or the setRemoteDescription method with a remote offer to receive simulcast, which are both methods on the RTCPeerConnection ~~object and~~ object. Additionally, the setParameters method ~~of the~~ on each RTCRtpSender ~~object~~ object can be used to inspect and modify the functionality.

~~The addTransceiver method establishes the~~ An RTCRtpSender's **simulcast envelope** ~~which~~ is established in the first successful negotiation that involves it sending simulcast instead of unicast, and includes the maximum number of simulcast streams that can be sent, as well as the ordering of ~~the~~ its encodings. ~~While characteristics~~ This simulcast envelope may be narrowed (reducing the number of layers) in subsequent renegotiation, but cannot be reexpanded. Characteristics of individual simulcast streams can be modified using the setParameters method, but the simulcast envelope itself cannot be ~~changed~~changed by that method.

~~One of the implications of this model is that the addTrack() method cannot provide simulcast functionality since it does not take sendEncodings as an argument, and therefore cannot configure an RTCRtpTransceiver to send simulcast.~~

~~Another implication is that the answerer cannot set the simulcast envelope directly. Upon calling the setRemoteDescription method of the RTCPeerConnection object, the simulcast envelope is configured on the RTCRtpTransceiver to contain the layers described by the specified session description. Once the envelope is determined, layers cannot be removed. They can be marked as inactive by setting the active member to false effectively disabling the layer.~~

One way to configure simulcast is with the sendEncodings option to addTransceiver(). While the addTrack() method lacks the sendEncodings argument necessary to configure simulcast, senders can be promoted to simulcast when the user agent is the answerer. Upon calling the setRemoteDescription method with a remote offer to receive simulcast, a **proposed envelope** is configured on an RTCRtpSender to contain the layers described in the specified session description. As long as this description isn't rolled back, the proposed envelope becomes the RTCRtpSender's simulcast envelope when negotiation completes. As above, this simulcast envelope may be narrowed in subsequent renegotiation, but not reexpanded.

CANDIDATE CORRECTION 12: Mark RTP Pause/Resume as not supported (PR #2755)

Show Current and Future Show Current Show Future

While setParameters cannot modify the ~~simulcast~~ simulcast envelope~~,~~, it is still possible to control the number of streams that are sent and the characteristics of those streams. Using setParameters, simulcast streams can be made inactive by setting the active member to false, or can be reactivated by setting the active member to true. [RFC7728] (RTP Pause/Resume) is not supported, nor is signaling of pause/resume via SDP Offer/Answer. Using setParameters, stream characteristics can be changed by modifying attributes such as maxBitrate.

> ***NOTE***
>
> ***Simulcast is frequently used to send multiple encodings to an SFU, which will then forward one of the simulcast streams to the end user. The user agent is therefore expected to allocate bandwidth between encodings in such a way that all simulcast streams are usable on their own; for instance, if two simulcast streams have the same `maxBitrate`, one would expect to see a similar bitrate on both streams. If bandwidth does not permit all simulcast streams to be sent in an usable form, the user agent is expected to stop sending some of the simulcast streams.***

As defined in [RFC9429] (section 3.7.), an offer from a user-agent will only contain a "send" description and no "recv" description on the `a=simulcast` line. Alternatives and restrictions (described in [RFC8853]) are not supported.

This specification does not define how to configure reception of multiple RTP encodings using `createOffer`, `createAnswer` or `addTransceiver`. However when `setRemoteDescription` is called with a corresponding remote description that is able to send multiple RTP encodings as defined in [RFC9429], and the browser supports receiving multiple RTP encodings, the `RTCRtpReceiver` may receive multiple RTP encodings and the parameters retrieved via the transceiver's `receiver`.`getParameters()` will reflect the encodings negotiated.

> ***NOTE***
>
> ***An `RTCRtpReceiver` can receive multiple RTP streams in a scenario where a Selective Forwarding Unit (SFU) switches between simulcast streams it receives from user agents. If the SFU does not rewrite RTP headers so as to arrange the switched streams into a single RTP stream prior to forwarding, the `RTCRtpReceiver` will receive packets from distinct RTP streams, each with their own SSRC and sequence number space. While the SFU may only forward a single RTP stream at any given time, packets from multiple RTP streams can become intermingled at the receiver due to reordering. An `RTCRtpReceiver` equipped to receive multiple RTP streams will therefore need to be able to correctly order the received packets, recognize potential loss events and react to them. Correct operation in this scenario is non-trivial and therefore is optional for implementations of this specification.***

§ *5.4.1.1 Encoding Parameter Examples*

This section is non-normative.

Examples of simulcast scenarios implemented with encoding parameters:

EXAMPLE 3

```
// Example of 3-layer spatial simulcast with all but the lowest resolution layer
disabled
var encodings = [
  {rid: 'q', active: true, scaleResolutionDownBy: 4.0}
  {rid: 'h', active: false, scaleResolutionDownBy: 2.0},
  {rid: 'f', active: false},
];
```

§ 5.4.2 "Hold" functionality

This section is non-normative.

Together, the `direction` attribute and the `replaceTrack` method enable developers to implement "hold" scenarios.

To send music to a peer and cease rendering received audio (music-on-hold):

EXAMPLE 4

```
async function playMusicOnHold() {
  try {
    // Assume we have an audio transceiver and a music track named musicTrack
    await audio.sender.replaceTrack(musicTrack);
    // Mute received audio
    audio.receiver.track.enabled = false;
    // Set the direction to send-only (requires negotiation)
    audio.direction = 'sendonly';
  } catch (err) {
    console.error(err);
  }
}
```

To respond to a remote peer's "sendonly" offer:

EXAMPLE 5

```
async function handleSendonlyOffer() {
  try {
    // Apply the sendonly offer first,
    // to ensure the receiver is ready for ICE candidates.
    await pc.setRemoteDescription(sendonlyOffer);
    // Stop sending audio
    await audio.sender.replaceTrack(null);
    // Align our direction to avoid further negotiation
    audio.direction = 'recvonly';
    // Call createAnswer and send a recvonly answer
    await doAnswer();
  } catch (err) {
    // handle signaling error
  }
}
```

To stop sending music and send audio captured from a microphone, as well to render received audio:

EXAMPLE 6

```
async function stopOnHoldMusic() {
  // Assume we have an audio transceiver and a microphone track named micTrack
  await audio.sender.replaceTrack(micTrack);
  // Unmute received audio
  audio.receiver.track.enabled = true;
  // Set the direction to sendrecv (requires negotiation)
  audio.direction = 'sendrecv';
}
```

To respond to being taken off hold by a remote peer:

EXAMPLE 7

```
async function onOffHold() {
  try {
    // Apply the sendrecv offer first, to ensure receiver is ready for ICE
candidates.
    await pc.setRemoteDescription(sendrecvOffer);
    // Start sending audio
    await audio.sender.replaceTrack(micTrack);
    // Set the direction sendrecv (just in time for the answer)
    audio.direction = 'sendrecv';
    // Call createAnswer and send a sendrecv answer
    await doAnswer();
  } catch (err) {
    // handle signaling error
  }
}
```

§ 5.5 `RTCDtlsTransport` Interface

The `RTCDtlsTransport` interface allows an application access to information about the Datagram Transport Layer Security (DTLS) transport over which RTP and RTCP packets are sent and received by `RTCRtpSender` and `RTCRtpReceiver` objects, as well other data such as SCTP packets sent and received by data channels. In particular, DTLS adds security to an underlying transport, and the `RTCDtlsTransport` interface allows access to information about the underlying transport and the security added. `RTCDtlsTransport` objects are constructed as a result of calls to `setLocalDescription()` and `setRemoteDescription()`. Each `RTCDtlsTransport` object represents the DTLS transport layer for the RTP or RTCP `component` of a specific `RTCRtpTransceiver`, or a group of `RTCRtpTransceiver`s if such a group has been negotiated via [RFC8843].

> ***NOTE***
> ***A new DTLS association for an existing `RTCRtpTransceiver` will be represented by an existing `RTCDtlsTransport` object, whose `state` will be updated accordingly, as opposed to being represented by a new object.***

An `RTCDtlsTransport` has a **[[DtlsTransportState]]** internal slot initialized to "`new`" and a **[[RemoteCertificates]]** slot initialized to an empty list.

When the underlying DTLS transport experiences an error, such as a certificate validation failure, or a fatal alert (see [RFC5246] section 7.2), the user agent *MUST* queue a task that runs the following steps:

1. Let *transport* be the `RTCDtlsTransport` object to receive the state update and error notification.

2. If the state of *transport* is already "`failed`", abort these steps.

3. Set *transport*.`[[DtlsTransportState]]` to "`failed`".

4. Fire an event named `error` using the `RTCErrorEvent` interface with its errorDetail attribute set to either "`dtls-failure`" or "`fingerprint-failure`", as appropriate, and other fields set as described under the `RTCErrorDetailType` enum description, at *transport*.

5. Fire an event named `statechange` at *transport*.

When the underlying DTLS transport needs to update the state of the corresponding `RTCDtlsTransport` object for any other reason, the user agent *MUST* queue a task that runs the following steps:

1. Let *transport* be the `RTCDtlsTransport` object to receive the state update.

2. Let *newState* be the new state.

3. Set *transport*.`[[DtlsTransportState]]` to *newState*.

4. If *newState* is `connected` then let *newRemoteCertificates* be the certificate chain in use by the

remote side, with each certificate encoded in binary Distinguished Encoding Rules (DER) [X690], and set *transport*.`[[RemoteCertificates]]` to *newRemoteCertificates*.

5. Fire an event named `statechange` at *transport*.

WebIDL

```
[Exposed=Window]
interface RTCDtlsTransport : EventTarget {
  [SameObject] readonly attribute RTCIceTransport iceTransport;
  readonly attribute RTCDtlsTransportState state;
  sequence<ArrayBuffer> getRemoteCertificates();
  attribute EventHandler onstatechange;
  attribute EventHandler onerror;
};
```

§ Attributes

`iceTransport` **of type `RTCIceTransport`, readonly**
: The `iceTransport` attribute is the underlying transport that is used to send and receive packets. The underlying transport may not be shared between multiple active `RTCDtlsTransport` objects.

`state` **of type `RTCDtlsTransportState`, readonly**
: The `state` attribute *MUST*, on getting, return the value of the `[[DtlsTransportState]]` slot.

`onstatechange` **of type EventHandler**
: The event type of this event handler is `statechange`.

`onerror` **of type EventHandler**
: The event type of this event handler is `error`.

§ Methods

`getRemoteCertificates`
: Returns the value of `[[RemoteCertificates]]`.

§ 5.5.1 `RTCDtlsTransportState` Enum

WebIDL

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

`RTCDtlsTransportState` Enumeration description

Enum value	Description
`new`	DTLS has not started negotiating yet.
`connecting`	DTLS is in the process of negotiating a secure connection and verifying the remote fingerprint.
`connected`	DTLS has completed negotiation of a secure connection and verified the remote fingerprint.
`closed`	The transport has been closed intentionally as the result of receipt of a close_notify alert, or calling `close()`.

Enum value	Description
`failed`	The transport has failed as the result of an error (such as receipt of an error alert or failure to validate the remote fingerprint).

§ 5.5.2 `RTCDtlsFingerprint` Dictionary

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

WebIDL

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

§ *Dictionary RTCDtlsFingerprint Members*

***`algorithm`* of type DOMString**
: One of the the hash function algorithms defined in the 'Hash function Textual Names' registry [IANA-HASH-FUNCTION].

***`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.

§ 5.6 `RTCIceTransport` Interface

The `RTCIceTransport` interface allows an application access to information about the ICE transport over which packets are sent and received. In particular, ICE manages peer-to-peer connections which involve state which the application may want to access. `RTCIceTransport` objects are constructed as a result of calls to `setLocalDescription()` and `setRemoteDescription()`. The underlying ICE state is managed by the ICE agent; as such, the state of an `RTCIceTransport` changes when the ICE Agent provides indications to the user agent as described below. Each `RTCIceTransport` object represents the ICE transport layer for the RTP or RTCP `component` of a specific `RTCRtpTransceiver`, or a group of `RTCRtpTransceiver`s if such a group has been negotiated via [RFC8843].

> ***NOTE***
> ***An ICE restart for an existing `RTCRtpTransceiver` will be represented by an existing `RTCIceTransport` object, whose `state` will be updated accordingly, as opposed to being represented by a new object.***

CANDIDATE CORRECTION 24: Queue two tasks upon finishing ICE gathering, and fire gatheringstatechange & icegatheringstatechange in same task (PR #2894)

◉ Show Current and Future ○ Show Current ○ Show Future

When the ICE Agent indicates that it began gathering a generation of candidates for an `RTCIceTransport` *transport* associated with an `RTCPeerConnection` *connection*, the user agent *MUST* queue a task that runs the following steps:

1. ~~Let *connection* be the `RTCPeerConnection` object associated with this ICE Agent.~~
2. If *connection*.`[[IsClosed]]` is `true`, abort these steps.
3. ~~Let *transport* be the `RTCIceTransport` for which candidate gathering began.~~

4. Set *transport*.`[[IceGathererState]]` to `gathering`.

~~.~~

5. Set *connection*.`[[IceGatheringState]]` to the value of deriving a new state value as described by the `RTCIceGatheringState` enum.

6. Let *connectionIceGatheringStateChanged* be `true` if *connection*.`[[IceGatheringState]]` changed in the previous step, otherwise `false`.

7. Do not read or modify state beyond this point.

8. Fire an event named `gatheringstatechange` at *transport*.

9. ~~Update the ICE gathering state of *connection*.~~

10. If *connectionIceGatheringStateChanged* is `true`, fire an event named `icegatheringstatechange` at *connection*.

When the ICE Agent is finished gathering a generation of candidates for an `RTCIceTransport` *transport* associated with an `RTCPeerConnection` *connection*, and those candidates have been surfaced to the application, the user agent *MUST* queue a task ~~that runs~~ to run the ~~following~~ following steps:

1. ~~Let *connection* be the `RTCPeerConnection` object associated with this ICE Agent.~~

2. If *connection*.`[[IsClosed]]` is `true`, abort these steps.

3. ~~Let *transport* be the `RTCIceTransport` for which candidate gathering finished.~~

4. If *connection*.`[[PendingLocalDescription]]` is not `null`, and represents the ICE generation for which gathering finished, add `a=end-of-candidates` to *connection*.`[[PendingLocalDescription]]`.sdp.

5. If *connection*.`[[CurrentLocalDescription]]` is not `null`, and represents the ICE generation for which gathering finished, add `a=end-of-candidates` to *connection*.`[[CurrentLocalDescription]]`.sdp.

6. Let ~~*newCandidate*~~*endOfGatheringCandidate* be the result of creating ~~an~~ an RTCIceCandidate with a new dictionary whose `sdpMid` and `sdpMLineIndex` are set to the values associated with this `RTCIceTransport`, `usernameFragment` ~~is~~ is set to the username fragment of the generation of candidates for which gathering finished, and `candidate` is ~~set~~ set to ~~an empty string~~`""`.

7. Fire an event named `icecandidate` using the `RTCPeerConnectionIceEvent` interface with the candidate attribute set to ~~*newCandidate*~~*endOfGatheringCandidate* at *connection*.

1. ~~If another generation of candidates is still being gathered, abort these steps.~~

> ~~***NOTE***~~
> ~~***This may occur if an ICE restart is initiated while the ICE agent is still gathering the previous generation of candidates.***~~

2. ~~Set *transport*.[[IceGathererState]] to `complete`.~~

3. ~~Fire an event named `gatheringstatechange` at *transport*.~~

4. ~~Update the ICE gathering state of *connection*.~~

When the ICE Agent has queued the above task, and no other generations of candidates is being gathered, the user agent *MUST* also queue a second task to run the following steps:

> ***NOTE***
> ***Other generations of candidates might still be gathering if an ICE restart was initiated while the ICE agent is still gathering the previous generation of candidates.***

1. If *connection*.`[[IsClosed]]` is `true`, abort these steps.

2. Set *transport*.`[[IceGathererState]]` to `complete`.

3. Set *connection*.`[[IceGatheringState]]` to the value of deriving a new state value as described by

the RTCIceGatheringState enum.

4. Let *connectionIceGatheringStateChanged* be true if *connection*.[[IceGatheringState]] changed in the previous step, otherwise false.

5. Do not read or modify state beyond this point.

6. Fire an event named gatheringstatechange at *transport*.

7. If *connectionIceGatheringStateChanged* is true, fire an event named icegatheringstatechange at *connection*.

8. Fire an event named icecandidate using the RTCPeerConnectionIceEvent interface with the candidate attribute set to null at *connection*.

> ***NOTE***
> ***The null candidate event is fired to ensure legacy compatibility. New code should monitor the gathering state of RTCIceTransport and/or RTCPeerConnection.***

When the ICE Agent indicates that a new ICE candidate is available for an RTCIceTransport, either by taking one from the ICE candidate pool or gathering it from scratch, the user agent *MUST* queue a task that runs the following steps:

1. Let *candidate* be the available ICE candidate.

2. Let *connection* be the RTCPeerConnection object associated with this ICE Agent.

3. If *connection*.[[IsClosed]] is true, abort these steps.

4. If either *connection*.[[PendingLocalDescription]] or *connection*.[[CurrentLocalDescription]] are not null, and represent the ICE generation for which *candidate* was gathered, surface the candidate with *candidate* and *connection*, and abort these steps.

5. Otherwise, append *candidate* to *connection*.[[EarlyCandidates]].

When the ICE Agent signals that the ICE role has changed due to an ICE binding request with a role collision per [RFC8445] section 7.3.1.1, the UA will queue a task to set the value of [[IceRole]] to the new value.

To **release early candidates** of a *connection*, run the following steps:

1. For each candidate, *candidate*, in *connection*.[[EarlyCandidates]], queue a task to surface the candidate with *candidate* and *connection*.

2. Set *connection*.[[EarlyCandidates]] to an empty list.

To **surface a candidate** with *candidate* and *connection*, run the following steps:

1. If *connection*.[[IsClosed]] is true, abort these steps.

2. Let *transport* be the RTCIceTransport for which *candidate* is being made available.

3. If *connection*.[[PendingLocalDescription]] is not null, and represents the ICE generation for which *candidate* was gathered, add *candidate* to *connection*.[[PendingLocalDescription]].sdp.

4. If *connection*.[[CurrentLocalDescription]] is not null, and represents the ICE generation for which *candidate* was gathered, add *candidate* to *connection*.[[CurrentLocalDescription]].sdp.

5. Let *newCandidate* be the result of creating an RTCIceCandidate with a new dictionary whose sdpMid and sdpMLineIndex are set to the values associated with this RTCIceTransport, usernameFragment is set to the username fragment of the candidate, and candidate is set to a string encoded using the candidate-attribute grammar to represent *candidate*.

6. Add *newCandidate* to *transport*'s set of local candidates.

7. Fire an event named icecandidate using the RTCPeerConnectionIceEvent interface with the candidate attribute set to *newCandidate* at *connection*.

The RTCIceTransportState of an RTCIceTransport may change because a candidate pair with a usable connection was found and selected or it may change without the selected candidate pair changing. The selected pair and `RTCIceTransportState` are related and are handled in the same task.

When the ICE Agent indicates that an `RTCIceTransport` has changed either the selected candidate pair, the `RTCIceTransportState` or both, the user agent *MUST* queue a task that runs the steps to **change the selected candidate pair and state**:

1. Let *connection* be the `RTCPeerConnection` object associated with this ICE Agent.
2. If *connection*.`[[IsClosed]]` is `true`, abort these steps.
3. Let *transport* be the `RTCIceTransport` whose state is changing.
4. Let *selectedCandidatePairChanged* be `false`.
5. Let *transportIceConnectionStateChanged* be `false`.
6. Let *connectionIceConnectionStateChanged* be `false`.
7. Let *connectionStateChanged* be `false`.
8. If *transport*'s selected candidate pair was changed, run the following steps:
    1. Let *newCandidatePair* be the result of creating an RTCIceCandidatePair with *local* and *remote*, representing the local and remote candidates of the indicated pair if one is selected, and `null` otherwise.
    2. Set *transport*.`[[SelectedCandidatePair]]` to *newCandidatePair*.
    3. Set *selectedCandidatePairChanged* to `true`.
9. If *transport*'s `RTCIceTransportState` was changed, run the following steps:
    1. Set *transport*.`[[IceTransportState]]` to the new indicated `RTCIceTransportState`.
    2. Set *transportIceConnectionStateChanged* to `true`.
    3. Set *connection*.`[[IceConnectionState]]` to the value of deriving a new state value as described by the `RTCIceConnectionState` enum.
    4. If *connection*.`[[IceConnectionState]]` changed in the previous step, set *connectionIceConnectionStateChanged* to `true`.
    5. Set *connection*.`[[ConnectionState]]` to the value of deriving a new state value as described by the `RTCPeerConnectionState` enum.
    6. If *connection*.`[[ConnectionState]]` changed in the previous step, set *connectionStateChanged* to `true`.
10. If *selectedCandidatePairChanged* is `true`, fire an event named `selectedcandidatepairchange` at *transport*.
11. If *transportIceConnectionStateChanged* is `true`, fire an event named `statechange` at *transport*.
12. If *connectionIceConnectionStateChanged* is `true`, fire an event named `iceconnectionstatechange` at *connection*.
13. If *connectionStateChanged* is `true`, fire an event named `connectionstatechange` at *connection*.

An `RTCIceTransport` object has the following internal slots:

- **[[IceTransportState]]** initialized to "`new`"
- **[[IceGathererState]]** initialized to "`new`"
- **[[SelectedCandidatePair]]** initialized to `null`
- **[[IceRole]]** initialized to "`unknown`"

WebIDL

```
[Exposed=Window]
interface RTCIceTransport : EventTarget {
```

```
  readonly attribute RTCIceRole role;
  readonly attribute RTCIceComponent component;
  readonly attribute RTCIceTransportState state;
  readonly attribute RTCIceGathererState gatheringState;
  sequence<RTCIceCandidate> getLocalCandidates();
  sequence<RTCIceCandidate> getRemoteCandidates();
  RTCIceCandidatePair? getSelectedCandidatePair();
  RTCIceParameters? getLocalParameters();
  RTCIceParameters? getRemoteParameters();
  attribute EventHandler onstatechange;
  attribute EventHandler ongatheringstatechange;
  attribute EventHandler onselectedcandidatepairchange;
};
```

§ Attributes

`role` **of type `RTCIceRole`, readonly**
: The `role` attribute *MUST*, on getting, return the value of the [[IceRole]] internal slot.

`component` **of type `RTCIceComponent`, readonly**
: The `component` attribute *MUST* return the ICE component of the transport. When RTCP mux is used, a single `RTCIceTransport` transports both RTP and RTCP and `component` is set to "`rtp`".

`state` **of type `RTCIceTransportState`, readonly**
: The `state` attribute *MUST*, on getting, return the value of the `[[IceTransportState]]` slot.

`gatheringState` **of type `RTCIceGathererState`, readonly**
: The `gatheringState` attribute *MUST*, on getting, return the value of the `[[IceGathererState]]` slot.

`onstatechange` **of type EventHandler**
: This event handler, of event handler event type `statechange`, *MUST* be fired any time the `RTCIceTransport` `state` changes.

`ongatheringstatechange` **of type EventHandler**
: This event handler, of event handler event type `gatheringstatechange`, *MUST* be fired any time the `RTCIceTransport`'s `[[IceGathererState]]` changes.

`onselectedcandidatepairchange` **of type EventHandler**
: This event handler, of event handler event type `selectedcandidatepairchange`, *MUST* be fired any time the `RTCIceTransport`'s selected candidate pair changes.

§ Methods

`getLocalCandidates`
: Returns a sequence describing the local ICE candidates gathered for this `RTCIceTransport` and sent in `onicecandidate`.

`getRemoteCandidates`
: Returns a sequence describing the remote ICE candidates received by this `RTCIceTransport` via `addIceCandidate()`.

> ***NOTE***
> ***`getRemoteCandidates` will not expose peer reflexive candidates since they are not received via `addIceCandidate()`.***

`getSelectedCandidatePair`
: Returns the selected candidate pair on which packets are sent. This method *MUST* return the value of the `[[SelectedCandidatePair]]` slot. When `RTCIceTransport.state` is "`new`" or "`closed`" `getSelectedCandidatePair` returns `null`.

`getLocalParameters`
: Returns the local ICE parameters received by this `RTCIceTransport` via `setLocalDescription`, or `null` if the parameters have not yet been received.

getRemoteParameters
Returns the remote ICE parameters received by this `RTCIceTransport` via `setRemoteDescription` or `null` if the parameters have not yet been received.

§ 5.6.1 `RTCIceParameters` Dictionary

WebIDL

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

§ *Dictionary `RTCIceParameters` Members*

usernameFragment **of type DOMString**
The ICE username fragment as defined in [RFC5245], Section 7.1.2.3.

password **of type DOMString**
The ICE password as defined in [RFC5245], Section 7.1.2.3.

CANDIDATE ADDITION 45: Convert RTCIceCandidatePair dictionary to an interface (PR #2961)

Show Current and Future Show Current Show Future

~~5.6.2 `RTCIceCandidatePair` Dictionary~~

§ 5.6.2 `RTCIceCandidatePair` Interface

This interface represents an ICE candidate pair, described in Section 4 in [RFC8445]. An `RTCIceCandidatePair` is a pairing of a local and a remote `RTCIceCandidate`.

To **create an RTCIceCandidatePair** with `RTCIceCandidate` objects, *local* and *remote*, run the following steps:

1. Let *candidatePair* be a newly created `RTCIceCandidatePair` object.
2. Let *candidatePair* have a **[[Local]]** internal slot, initialized to *local*.
3. Let *candidatePair* have a **[[Remote]]** internal slot, initialized to *remote*.
4. Return *candidatePair*.

```
dictionary [Exposed=Window]
interface RTCIceCandidatePair {
  [SameObject] readonly attribute RTCIceCandidate local;
  [SameObject] readonly attribute RTCIceCandidate remote;
};
```

~~*Dictionary `RTCIceCandidatePair` Members*~~

§ *Attributes*

local **of type RTCIceCandidate~~,~~ , readonly**

~~The local ICE candidate.~~

The `local` attribute *MUST*, on getting, return the value of the `[[Local]]` internal slot.

remote **of type RTCIceCandidate~~,~~ , readonly**

~~The remote ICE candidate.~~

The `remote` attribute *MUST*, on getting, return the value of the `[[Remote]]` internal slot.

§ 5.6.3 `RTCIceGathererState` Enum

WebIDL

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

`RTCIceGathererState` Enumeration description

Enum value	Description
`new`	The `RTCIceTransport` was just created, and has not started gathering candidates yet.
`gathering`	The `RTCIceTransport` is in the process of gathering candidates.
`complete`	The `RTCIceTransport` has completed gathering and the end-of-candidates indication for this transport has been sent. It will not gather candidates again until an ICE restart causes it to restart.

§ 5.6.4 `RTCIceTransportState` Enum

WebIDL

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

`RTCIceTransportState` Enumeration description

Enum value	Description
`closed`	The `RTCIceTransport` has shut down and is no longer responding to STUN requests.
`failed`	**PROPOSED CORRECTION 8:** Put ICE transport connection in `failed` state when no candidates are received (PR #2704) ◉ Show Current and Future ○ Show Current ○ Show Future The `RTCIceTransport` has finished gathering, received an indication that there are no more remote candidates, finished checking all candidate pairs, and all pairs have either failed connectivity checks or ~~have~~ lost consent, and either zero local candidates were

Enum value	Description
	gathered or the PAC timer has expired [RFC8863]. This is a terminal state until ICE is restarted. Since an ICE restart may cause connectivity to resume, entering the "failed" state does not cause DTLS transports, SCTP associations or the data channels that run over them to close, or tracks to mute.
disconnected	The ICE Agent has determined that connectivity is currently lost for this RTCIceTransport. This is a transient state that may trigger intermittently (and resolve itself without action) on a flaky network. The way this state is determined is implementation dependent. Examples include: • Losing the network interface for the connection in use. • Repeatedly failing to receive a response to STUN requests. Alternatively, the RTCIceTransport has finished checking all existing candidates pairs and not found a connection (or consent checks [RFC7675] once successful, have now failed), but it is still gathering and/or waiting for additional remote candidates.
new	The RTCIceTransport is gathering candidates and/or waiting for remote candidates to be supplied, and has not yet started checking.
checking	The RTCIceTransport has received at least one remote candidate (by means of addIceCandidate() or discovered as a peer-reflexive candidate when receiving a STUN binding request) and is checking candidate pairs and has either not yet found a connection or consent checks [RFC7675] have failed on all previously successful candidate pairs. In addition to checking, it may also still be gathering.
completed	The RTCIceTransport has finished gathering, received an indication that there are no more remote candidates, finished checking all candidate pairs and found a connection. If consent checks [RFC7675] subsequently fail on all successful candidate pairs, the state transitions to "failed".
connected	The RTCIceTransport has found a usable connection, but is still checking other candidate pairs to see if there is a better connection. It may also still be gathering and/or waiting for additional remote candidates. If consent checks [RFC7675] fail on the connection in use, and there are no other successful candidate pairs available, then the state transitions to "checking" (if there are candidate pairs remaining to be checked) or "disconnected" (if there are no candidate pairs to check, but the peer is still gathering and/or waiting for additional remote candidates).

NOTE

The most common transitions for a successful call will be new -> checking -> connected -> completed, but under specific circumstances (only the last checked candidate succeeds, and gathering and the no-more candidates indication both occur prior to success), the state can transition directly from "checking" to "completed".

An ICE restart causes candidate gathering and connectivity checks to begin anew, causing a transition to "connected" if begun in the "completed" state. If begun in the transient "disconnected" state, it causes a transition to "checking", effectively forgetting that connectivity was previously lost.

The "failed" and "completed" states require an indication that there are no additional remote candidates. This can be indicated by calling addIceCandidate with a candidate value whose candidate property is set to an empty string or by canTrickleIceCandidates being set to false.

Some example state transitions are:

- (RTCIceTransport first created, as a result of setLocalDescription or setRemoteDescription): "new"
- ("new", remote candidates received): "checking"
- ("checking", found usable connection): "connected"
- ("checking", checks fail but gathering still in progress): "disconnected"
- ("checking", gave up): "failed"
- ("disconnected", new local candidates): "checking"
- ("connected", finished all checks): "completed"

- ("completed", lost connectivity): "disconnected"
- ("disconnected" or "failed", ICE restart occurs): "checking"
- ("completed", ICE restart occurs): "connected"
- RTCPeerConnection.close(): "closed"

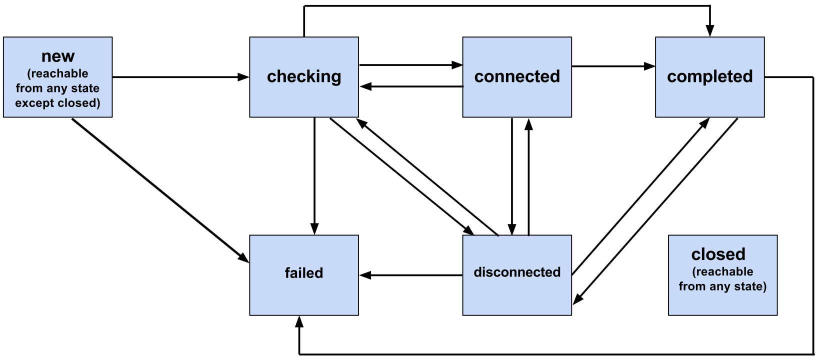


Figure 2 Non-normative ICE transport state transition diagram

§ 5.6.5 `RTCIceRole` Enum

WebIDL

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

`RTCIceRole` Enumeration description

Enum value	Description
`unknown`	An agent whose role as defined by [RFC5245], Section 3, has not yet been determined.
`controlling`	A controlling agent as defined by [RFC5245], Section 3.
`controlled`	A controlled agent as defined by [RFC5245], Section 3.

§ 5.6.6 `RTCIceComponent` Enum

WebIDL

```
enum RTCIceComponent {
  "rtp",
  "rtcp"
};
```

`RTCIceComponent` Enumeration description

Enum value	Description
`rtp`	The ICE Transport is used for RTP (or RTCP multiplexing), as defined in [RFC5245], Section 4.1.1.1. Protocols multiplexed with RTP (e.g. data channel) share its component ID. This represents the `component-id` value `1` when encoded in `candidate-attribute`.
`rtcp`	The ICE Transport is used for RTCP as defined by [RFC5245], Section 4.1.1.1. This represents the `component-id` value `2` when encoded in `candidate-attribute`.

§ 5.7 RTCTrackEvent

The track event uses the RTCTrackEvent interface.

WebIDL

```
[Exposed=Window]
interface RTCTrackEvent : Event {
  constructor(DOMString type, RTCTrackEventInit eventInitDict);
  readonly attribute RTCRtpReceiver receiver;
  readonly attribute MediaStreamTrack track;
  [SameObject] readonly attribute FrozenArray<MediaStream> streams;
  readonly attribute RTCRtpTransceiver transceiver;
};
```

§ Constructors

RTCTrackEvent.constructor()

§ Attributes

receiver **of type** `RTCRtpReceiver`**, readonly**
: The `receiver` attribute represents the `RTCRtpReceiver` object associated with the event.

track **of type** `MediaStreamTrack`**, readonly**
: The `track` attribute represents the `MediaStreamTrack` object that is associated with the `RTCRtpReceiver` identified by `receiver`.

streams **of type FrozenArray<**`MediaStream`**>, readonly**
: The `streams` attribute returns an array of `MediaStream` objects representing the `MediaStreams` that this event's `track` is a part of.

transceiver **of type** `RTCRtpTransceiver`**, readonly**
: The `transceiver` attribute represents the `RTCRtpTransceiver` object associated with the event.

WebIDL

```
dictionary RTCTrackEventInit : EventInit {
  required RTCRtpReceiver receiver;
  required MediaStreamTrack track;
  sequence<MediaStream> streams = [];
  required RTCRtpTransceiver transceiver;
};
```

§ Dictionary `RTCTrackEventInit` Members

receiver **of type** `RTCRtpReceiver`**, required**
: The `receiver` member represents the `RTCRtpReceiver` object associated with the event.

track **of type** `MediaStreamTrack`**, required**
: The `track` member represents the `MediaStreamTrack` object that is associated with the `RTCRtpReceiver` identified by `receiver`.

streams **of type sequence<**`MediaStream`**>, defaulting to** `[]`
: The `streams` member is an array of `MediaStream` objects representing the `MediaStreams` that this event's `track` is a part of.

transceiver **of type** `RTCRtpTransceiver`**, required**

The `transceiver` attribute represents the `RTCRtpTransceiver` object associated with the event.

§ 6. Peer-to-peer Data API

The Peer-to-peer Data API lets a web application send and receive generic application data peer-to-peer. The API for sending and receiving data models the behavior of Web Sockets.

§ 6.1 RTCPeerConnection Interface Extensions

The Peer-to-peer data API extends the `RTCPeerConnection` interface as described below.

WebIDL

```
        partial interface RTCPeerConnection {
  readonly attribute RTCSctpTransport? sctp;
  RTCDataChannel createDataChannel(USVString label,
                                   optional RTCDataChannelInit dataChannelDict = {});
  attribute EventHandler ondatachannel;
};
```

§ Attributes

***`sctp`* of type `RTCSctpTransport`, readonly, nullable**

The SCTP transport over which SCTP data is sent and received. If SCTP has not been negotiated, the value is null. This attribute *MUST* return the `RTCSctpTransport` object stored in the `[[SctpTransport]]` internal slot.

***`ondatachannel`* of type EventHandler**

The event type of this event handler is `datachannel`.

§ Methods

`createDataChannel`

Creates a new `RTCDataChannel` object with the given label. The `RTCDataChannelInit` dictionary can be used to configure properties of the underlying channel such as data reliability.

When the `createDataChannel` method is invoked, the user agent *MUST* run the following steps.

1. Let *connection* be the `RTCPeerConnection` object on which the method is invoked.
2. If *connection*.`[[IsClosed]]` is `true`, throw an `InvalidStateError`.
3. Create an RTCDataChannel, *channel*.
4. Initialize *channel*.`[[DataChannelLabel]]` to the value of the first argument.
5. If the UTF-8 representation of `[[DataChannelLabel]]` is longer than 65535 bytes, throw a `TypeError`.
6. Let *options* be the second argument.
7. Initialize *channel*.`[[MaxPacketLifeTime]]` to *option*.`maxPacketLifeTime`, if present, otherwise `null`.
8. Initialize *channel*.`[[MaxRetransmits]]` to *option*.`maxRetransmits`, if present, otherwise `null`.
9. Initialize *channel*.`[[Ordered]]` to *option*.`ordered`.

10. Initialize *channel*.`[[DataChannelProtocol]]` to *option*.`protocol`.

11. If the UTF-8 representation of `[[DataChannelProtocol]]` is longer than 65535 bytes, throw a `TypeError`.

12. Initialize *channel*.`[[Negotiated]]` to *option*.`negotiated`.

13. Initialize *channel*.`[[DataChannelId]]` to the value of *option*.`id`, if it is present and `[[Negotiated]]` is true, otherwise `null`.

> ***NOTE***
> ***This means the `id` member will be ignored if the data channel is negotiated in-band; this is intentional. Data channels negotiated in-band should have IDs selected based on the DTLS role, as specified in [RFC8832].***

14. If `[[Negotiated]]` is `true` and `[[DataChannelId]]` is `null`, throw a `TypeError`.

15. If both `[[MaxPacketLifeTime]]` and `[[MaxRetransmits]]` attributes are set (not null), throw a `TypeError`.

16. If a setting, either `[[MaxPacketLifeTime]]` or `[[MaxRetransmits]]`, has been set to indicate unreliable mode, and that value exceeds the maximum value supported by the user agent, the value *MUST* be set to the user agents maximum value.

17. If `[[DataChannelId]]` is equal to 65535, which is greater than the maximum allowed ID of 65534 but still qualifies as an unsigned short, throw a `TypeError`.

18. If the `[[DataChannelId]]` slot is `null` (due to no ID being passed into `createDataChannel`, or `[[Negotiated]]` being false), and the DTLS role of the SCTP transport has already been negotiated, then initialize `[[DataChannelId]]` to a value generated by the user agent, according to [RFC8832], and skip to the next step. If no available ID could be generated, or if the value of the `[[DataChannelId]]` slot is being used by an existing `RTCDataChannel`, throw an `OperationError` exception.

> ***NOTE***
> ***If the `[[DataChannelId]]` slot is `null` after this step, it will be populated during the RTCSctpTransport connected procedure.***

19. Let *transport* be *connection*.`[[SctpTransport]]`.

If the `[[DataChannelId]]` slot is not `null`, *transport* is in the "`connected`" state and `[[DataChannelId]]` is greater or equal to *transport*.`[[MaxChannels]]`, throw an `OperationError`.

20. If *channel* is the first `RTCDataChannel` created on *connection*, update the negotiation-needed flag for *connection*.

21. Append *channel* to *connection*.`[[DataChannels]]`.

22. Return *channel* and continue the following steps in parallel.

23. Create *channel*'s associated underlying data transport and configure it according to the relevant properties of *channel*.

§ 6.1.1 `RTCSctpTransport` Interface

The `RTCSctpTransport` interface allows an application access to information about the SCTP data channels tied to a particular SCTP association.

§ *6.1.1.1 Create an instance*

To **create an `RTCSctpTransport`** with an initial state, *initialState*, run the following steps:

1. Let *transport* be a new `RTCSctpTransport` object.
2. Let *transport* have a **[[SctpTransportState]]** internal slot initialized to *initialState*.
3. Let *transport* have a **[[MaxMessageSize]]** internal slot and run the steps labeled update the data max message size to initialize it.
4. Let *transport* have a **[[MaxChannels]]** internal slot initialized to `null`.
5. Return *transport*.

§ *6.1.1.2 Update max message size*

To **update the data max message size** of an `RTCSctpTransport` run the following steps:

1. Let *transport* be the `RTCSctpTransport` object to be updated.
2. Let *remoteMaxMessageSize* be the value of the `max-message-size` SDP attribute read from the remote description, as described in [RFC8841] (section 6), or 65536 if the attribute is missing.
3. Let *canSendSize* be the number of bytes that this client can send (i.e. the size of the local send buffer) or 0 if the implementation can handle messages of any size.
4. If both *remoteMaxMessageSize* and *canSendSize* are 0, set `[[MaxMessageSize]]` to the positive Infinity value.
5. Else, if either *remoteMaxMessageSize* or *canSendSize* is 0, set `[[MaxMessageSize]]` to the larger of the two.
6. Else, set `[[MaxMessageSize]]` to the smaller of *remoteMaxMessageSize* or *canSendSize*.

§ *6.1.1.3 Connected procedure*

Once an **SCTP transport is connected**, meaning the SCTP association of an `RTCSctpTransport` has been established, run the following steps:

1. Let *transport* be the `RTCSctpTransport` object.
2. Let *connection* be the `RTCPeerConnection` object associated with *transport*.
3. Set `[[MaxChannels]]` to the minimum of the negotiated amount of incoming and outgoing SCTP streams.
4. For each of *connection*'s `RTCDataChannel`:
    1. Let *channel* be the `RTCDataChannel` object.
    2. If *channel*.`[[DataChannelId]]` is `null`, initialize `[[DataChannelId]]` to the value generated by the underlying sctp data channel, according to [RFC8832].
    3. If *channel*.`[[DataChannelId]]` is greater or equal to *transport*.`[[MaxChannels]]`, or the previous step failed to assign an id, close the *channel* due to a failure. Otherwise, announce the channel as open.
5. Fire an event named `statechange` at *transport*.

> ***NOTE***
>
> ***This event is fired before the `open` events fired by announcing the channel as open; the `open` events are fired from a queued task.***

WebIDL

```
[Exposed=Window]
```

```
interface RTCSctpTransport : EventTarget {
  readonly attribute RTCDtlsTransport transport;
  readonly attribute RTCSctpTransportState state;
  readonly attribute unrestricted double maxMessageSize;
  readonly attribute unsigned short? maxChannels;
  attribute EventHandler onstatechange;
};
```

§ *Attributes*

***`transport`* of type `RTCDtlsTransport`, readonly**
The transport over which all SCTP packets for data channels will be sent and received.

***`state`* of type `RTCSctpTransportState`, readonly**
The current state of the SCTP transport. On getting, this attribute *MUST* return the value of the `[[SctpTransportState]]` slot.

***`maxMessageSize`* of type unrestricted double, readonly**
The maximum size of data that can be passed to `RTCDataChannel`'s `send()` method. The attribute *MUST*, on getting, return the value of the `[[MaxMessageSize]]` slot.

***`maxChannels`* of type unsigned short , readonly, nullable**
The maximum amount of `RTCDataChannel`'s that can be used simultaneously. The attribute *MUST*, on getting, return the value of the `[[MaxChannels]]` slot.

> ***NOTE***
> ***This attribute's value will be `null` until the SCTP transport goes into the "`connected`" state.***

***`onstatechange`* of type EventHandler**
The event type of this event handler is `statechange`.

§ 6.1.2 `RTCSctpTransportState` Enum

`RTCSctpTransportState` indicates the state of the SCTP transport.

WebIDL

```
enum RTCSctpTransportState {
  "connecting",
  "connected",
  "closed"
};
```

`RTCSctpTransportState` Enumeration description

Enum value	Description
`connecting`	The `RTCSctpTransport` is in the process of negotiating an association. This is the initial state of the [[SctpTransportState]] slot when an `RTCSctpTransport` is created.
`connected`	When the negotiation of an association is completed, a task is queued to update the [[SctpTransportState]] slot to "`connected`".
`closed`	A task is queued to update the [[SctpTransportState]] slot to "`closed`" when: • a SHUTDOWN or ABORT chunk is received. • the SCTP association has been closed intentionally, such as by closing the peer connection or applying a remote description that rejects data or changes the SCTP port.

Enum value	Description
	• the underlying DTLS association has transitioned to "closed" state. Note that the last transition is logical due to the fact that an SCTP association requires an established DTLS connection - [RFC8261] section 6.1 specifies that SCTP over DTLS is single-homed - and that no way of of switching to an alternate transport is defined in this API.

§ 6.2 RTCDataChannel

The RTCDataChannel interface represents a bi-directional data channel between two peers. An RTCDataChannel is created via a factory method on an RTCPeerConnection object. The messages sent between the browsers are described in [RFC8831] and [RFC8832].

There are two ways to establish a connection with RTCDataChannel. The first way is to simply create an RTCDataChannel at one of the peers with the negotiated RTCDataChannelInit dictionary member unset or set to its default value false. This will announce the new channel in-band and trigger an RTCDataChannelEvent 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 negotiated RTCDataChannelInit 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 negotiated RTCDataChannelInit 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. In the case of SCTP data channels utilizing an RTCSctpTransport (which represents the state of the SCTP association), the underlying data transport is the SCTP stream pair. 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. The actual wire protocol between the peers is specified by the WebRTC DataChannel Protocol specification [RFC8831].

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.

An RTCDataChannel, created with createDataChannel or dispatched via an RTCDataChannelEvent, *MUST* initially be in the "connecting" state. When the RTCDataChannel object's underlying data transport is ready, the user agent *MUST* announce the RTCDataChannel as open.

§ 6.2.1 Creating a data channel

To **create an RTCDataChannel**, run the following steps:

1. Let *channel* be a newly created RTCDataChannel object.
2. Let *channel* have a **[[ReadyState]]** internal slot initialized to "connecting".
3. Let *channel* have a **[[BufferedAmount]]** internal slot initialized to 0.
4. Let *channel* have internal slots named **[[DataChannelLabel]]**, **[[Ordered]]**, **[[MaxPacketLifeTime]]**, **[[MaxRetransmits]]**, **[[DataChannelProtocol]]**, **[[Negotiated]]**, and **[[DataChannelId]]**.
5. CANDIDATE ADDITION 48: Make RTCDataChannel transferable to DedicatedWorker (PR #2988)

Show Current and Future Show Current Show Future

Let *channel* have a **[[IsTransferable]]** internal slot initialized to `true`.

6. CANDIDATE ADDITION 48: Make RTCDataChannel transferable to DedicatedWorker (PR #2988)

Show Current and Future Show Current Show Future

Queue a task to run the following step:

1. Set *channel*.`[[IsTransferable]]` to `false`.

This task needs to run before any task enqueued by the receiving messages on a data channel algorithm for *channel*. This ensures that no message is lost during the transfer of a `RTCDataChannel`.

7. Return *channel*.

§ 6.2.2 Announcing a data channel as open

When the user agent is to **announce an `RTCDataChannel` as open**, the user agent *MUST* queue a task to run the following steps:

1. If the associated `RTCPeerConnection` object's `[[IsClosed]]` slot is `true`, abort these steps.
2. Let *channel* be the `RTCDataChannel` object to be announced.
3. If *channel*.`[[ReadyState]]` is "`closing`" or "`closed`", abort these steps.
4. Set *channel*.`[[ReadyState]]` to "`open`".
5. Fire an event named `open` at *channel*.

§ 6.2.3 Announcing a data channel instance

When an underlying data transport is to be announced (the other peer created a channel with `negotiated` unset or set to false), the user agent of the peer that did not initiate the creation process *MUST* queue a task to run the following steps:

1. Let *connection* be the `RTCPeerConnection` object associated with the underlying data transport.
2. If *connection*.`[[IsClosed]]` is `true`, abort these steps.
3. Create an RTCDataChannel, *channel*.
4. Let *configuration* be an information bundle received from the other peer as a part of the process to establish the underlying data transport described by the WebRTC DataChannel Protocol specification [RFC8832].
5. Initialize *channel*.`[[DataChannelLabel]]`, `[[Ordered]]`, `[[MaxPacketLifeTime]]`, `[[MaxRetransmits]]`, `[[DataChannelProtocol]]`, and `[[DataChannelId]]` internal slots to the corresponding values in *configuration*.
6. Initialize *channel*.`[[Negotiated]]` to `false`.
7. Append *channel* to *connection*.`[[DataChannels]]`.
8. Set *channel*.`[[ReadyState]]` to "`open`" (but do not fire the `open` event, yet).

> ***NOTE***
> ***This allows to start sending messages inside of the `datachannel` event handler prior to the `open` event being fired.***

9. Fire an event named `datachannel` using the `RTCDataChannelEvent` interface with the `channel` attribute set to *channel* at *connection*.

10. Announce the data channel as open.

CANDIDATE CORRECTION 38: Prevent GC of non-closed RTCDataChannels (PR #2902)

Show Current and Future Show Current Show Future

~~6.2.4 Closing procedure~~

§ 6.2.4 Closing procedure

An RTCDataChannel object's underlying data transport may be torn down in a non-abrupt manner by running the **closing procedure**. When that happens the user agent *MUST* queue a task to run the following steps:

1. Let *channel* be the RTCDataChannel object whose underlying data transport was closed.
2. Let *connection* be the RTCPeerConnection object associated with *channel*.
3. Remove *channel* from *connection*.[[DataChannels]].
4. Unless the procedure was initiated by *channel*.close, set *channel*.[[ReadyState]] to "closing" and fire an event named closing at *channel*.
5. Run the following steps ~~in parallel~~in parallel:
    1. Finish sending all currently pending messages of the *channel*.
    2. Follow the closing procedure defined for the *channel*'s underlying data transport :
        1. In the case of an SCTP-based transport, follow [RFC8831], section 6.7.
    3. ~~Render~~ Close the *channel*'s data transport ~~closed~~ by following the associated procedure.

§ 6.2.5 Announcing a data channel as closed

When an RTCDataChannel object's underlying data transport has been **closed**, the user agent *MUST* queue a task to run the following steps:

1. Let *channel* be the RTCDataChannel object whose underlying data transport was closed.
2. If *channel*.[[ReadyState]] is "closed", abort these steps.
3. Set *channel*.[[ReadyState]] to "closed".
4. Remove *channel* from *connection*.[[DataChannels]] if it is still there.
5. If the transport was closed with an error, fire an event named error using the RTCErrorEvent interface with its errorDetail attribute set to "sctp-failure" at *channel*.
6. Fire an event named close at *channel*.

CANDIDATE ADDITION 48: Make RTCDataChannel transferable to DedicatedWorker (PR #2988)

Show Current and Future Show Current Show Future

§ 6.2.6 Transfering data channel

The RTCDataChannel transfer steps, given *value* and *dataHolder*, are:

1. If *value*.`[[IsTransferable]]` is `false`, throw a `DataCloneError` DOMException.
2. Set *dataHolder*.`[[ReadyState]]` to *value*.`[[ReadyState]]`.
3. Set *dataHolder*.`[[DataChannelLabel]]` to *value*.`[[DataChannelLabel]]`.
4. Set *dataHolder*.`[[Ordered]]` to *value*.`[[Ordered]]`.
5. Set *dataHolder*.`[[MaxPacketLifeTime]]` to *value*..`[[MaxPacketLifeTime]]`
6. Set *dataHolder*.`[[MaxRetransmits]]` to *value*.`[[MaxRetransmits]]`.
7. Set *dataHolder*.`[[DataChannelProtocol]]` to *value*.`[[DataChannelProtocol]]`.
8. Set *dataHolder*.`[[Negotiated]]` to *value*.`[[Negotiated]]`.
9. Set *dataHolder*.`[[DataChannelId]]` to *value*.`[[DataChannelId]]`.
10. Set *dataHolder*'s underlying data transport to *value* underlying data transport.
11. Set *value*.`[[IsTransferable]]` to `false`.
12. Set *value*.`[[ReadyState]]` to "closed".

The `RTCDataChannel` transfer-receiving steps, given *dataHolder* and *channel*, are:

1. Initialize *channel*.`[[ReadyState]]` to *dataHolder*.`[[ReadyState]]`.
2. Initialize *channel*.`[[DataChannelLabel]]` to *dataHolder*.`[[DataChannelLabel]]`.
3. Initialize *channel*.`[[Ordered]]` to *dataHolder*.`[[Ordered]]`.
4. Initialize *channel*.`[[MaxPacketLifeTime]]` to *dataHolder*.`[[MaxPacketLifeTime]]`.
5. Initialize *channel*.`[[MaxRetransmits]]` to *dataHolder*.`[[MaxRetransmits]]`.
6. Initialize *channel*.`[[DataChannelProtocol]]` to *dataHolder*.`[[DataChannelProtocol]]`.
7. Initialize *channel*.`[[Negotiated]]` to *dataHolder*.`[[Negotiated]]`.
8. Initialize *channel*.`[[DataChannelId]]` to *dataHolder*.`[[DataChannelId]]`.
9. Initialize *channel*'s underlying data transport to *dataHolder*'s underlying data transport.

The above steps do not need to transfer `[[BufferedAmount]]` as its value will always be equal to `0`. The reason is an `RTCDataChannel` can be transferred only if its send() algorithm was not called prior the transfer.

If the underlying data transport is closed at the time of the transfer-receiving steps, the `RTCDataChannel` object will be closed by running the announcing a data channel as closed algorithm immediately after the transfer-receiving steps.

§ 6.2.7 Error on creating data channels

In some cases, the user agent may be **unable to create an `RTCDataChannel` 's** underlying data transport. For example, the data channel's `id` may be outside the range negotiated by the [RFC8831] implementations in the SCTP handshake. When the user agent determines that an `RTCDataChannel`'s underlying data transport cannot be created, the user agent *MUST* queue a task to run the following steps:

1. Let *channel* be the `RTCDataChannel` object for which the user agent could not create an underlying data transport.
2. Set *channel*.`[[ReadyState]]` to "`closed`".
3. Fire an event named `error` using the `RTCErrorEvent` interface with the `errorDetail` attribute set to "`data-channel-failure`" at *channel*.
4. Fire an event named close at *channel*.

§ 6.2.8 Receiving messages on a data channel

When an `RTCDataChannel` message has been received via the underlying data transport with type *type* and data *rawData*, the user agent *MUST* queue a task to run the following steps:

1. Let *channel* be the `RTCDataChannel` object for which the user agent has received a message.
2. Let *connection* be the `RTCPeerConnection` object associated with *channel*.
3. If *channel*.`[[ReadyState]]` is not "`open`", abort these steps and discard *rawData*.
4. Execute the sub step by switching on *type* and *channel*.`binaryType`:
   - If *type* indicates that *rawData* is a `string`:

     Let *data* be a DOMString that represents the result of decoding *rawData* as UTF-8.
   - If *type* indicates that *rawData* is binary and `binaryType` is `"blob"`:

     Let *data* be a new `Blob` object containing *rawData* as its raw data source.
   - If *type* indicates that *rawData* is binary and `binaryType` is `"arraybuffer"`:

     Let *data* be a new `ArrayBuffer` object containing *rawData* as its raw data source.
5. Fire an event named `message` using the `MessageEvent` interface with its `origin` attribute initialized to the serialization of an origin of *connection*.`[[DocumentOrigin]]`, and the `data` attribute initialized to *data* at *channel*.

PROPOSED ADDITION 48: Make RTCDataChannel transferable to DedicatedWorker (PR #2988)

◉ Show Current and Future ○ Show Current ○ Show Future

[~~Exposed=Window~~Exposed=(Window,DedicatedWorker), Transferable]

```
interface RTCDataChannel : EventTarget {
  readonly attribute USVString label;
  readonly attribute boolean ordered;
  readonly attribute unsigned short? maxPacketLifeTime;
  readonly attribute unsigned short? maxRetransmits;
  readonly attribute USVString protocol;
  readonly attribute boolean negotiated;
  readonly attribute unsigned short? id;
  readonly attribute RTCDataChannelState readyState;
  readonly attribute unsigned long bufferedAmount;
  [EnforceRange] attribute unsigned long bufferedAmountLowThreshold;
  attribute EventHandler onopen;
  attribute EventHandler onbufferedamountlow;
  attribute EventHandler onerror;
  attribute EventHandler onclosing;
  attribute EventHandler onclose;
  undefined close();
  attribute EventHandler onmessage;
  attribute BinaryType binaryType;
  undefined send(USVString data);
  undefined send(Blob data);
  undefined send(ArrayBuffer data);
  undefined send(ArrayBufferView data);
};
```

§ Attributes

`label` **of type USVString, readonly**

The `label` attribute represents a label that can be used to distinguish this `RTCDataChannel` object from other `RTCDataChannel` objects. Scripts are allowed to create multiple `RTCDataChannel` objects with the same label. On getting, the attribute *MUST* return the value of the `[[DataChannelLabel]]` slot.

***`ordered`* of type boolean, readonly**
The `ordered` attribute returns true if the `RTCDataChannel` is ordered, and false if out of order delivery is allowed. On getting, the attribute *MUST* return the value of the `[[Ordered]]` slot.

***`maxPacketLifeTime`* of type unsigned short, readonly, nullable**
The `maxPacketLifeTime` attribute returns the length of the time window (in milliseconds) during which transmissions and retransmissions may occur in unreliable mode. On getting, the attribute *MUST* return the value of the `[[MaxPacketLifeTime]]` slot.

***`maxRetransmits`* of type unsigned short, readonly, nullable**
The `maxRetransmits` attribute returns the maximum number of retransmissions that are attempted in unreliable mode. On getting, the attribute *MUST* return the value of the `[[MaxRetransmits]]` slot.

***`protocol`* of type USVString, readonly**
The `protocol` attribute returns the name of the sub-protocol used with this `RTCDataChannel`. On getting, the attribute *MUST* return the value of the `[[DataChannelProtocol]]` slot.

***`negotiated`* of type boolean, readonly**
The `negotiated` attribute returns true if this `RTCDataChannel` was negotiated by the application, or false otherwise. On getting, the attribute *MUST* return the value of the `[[Negotiated]]` slot.

***`id`* of type unsigned short, readonly, nullable**
The `id` attribute returns the ID for this `RTCDataChannel`. The value is initially null, which is what will be returned if the ID was not provided at channel creation time, and the DTLS role of the SCTP transport has not yet been negotiated. Otherwise, it will return the ID that was either selected by the script or generated by the user agent according to [RFC8832]. After the ID is set to a non-null value, it will not change. On getting, the attribute *MUST* return the value of the `[[DataChannelId]]` slot.

***`readyState`* of type `RTCDataChannelState`, readonly**
The `readyState` attribute represents the state of the `RTCDataChannel` object. On getting, the attribute *MUST* return the value of the `[[ReadyState]]` slot.

***`bufferedAmount`* of type unsigned long, readonly**
The `bufferedAmount` attribute *MUST*, on getting, return the value of the `[[BufferedAmount]]` slot. The attribute exposes the number of bytes of application data (UTF-8 text and binary data) that have been queued using `send()`. Even though the data transmission can occur in parallel, the returned value *MUST NOT* be decreased before the current task yielded back to the event loop to prevent race conditions. The value does not include framing overhead incurred by the protocol, or buffering done by the operating system or network hardware. The value of the `[[BufferedAmount]]` slot will only increase with each call to the `send()` method as long as the `[[ReadyState]]` slot is "`open`"; however, the slot does not reset to zero once the channel closes. When the underlying data transport sends data from its queue, the user agent *MUST* queue a task that reduces `[[BufferedAmount]]` with the number of bytes that was sent.

***`bufferedAmountLowThreshold`* of type unsigned long**
The `bufferedAmountLowThreshold` attribute sets the threshold at which the `bufferedAmount` is considered to be low. When the `bufferedAmount` decreases from above this threshold to equal or below it, the `bufferedamountlow` event fires. The `bufferedAmountLowThreshold` is initially zero on each new `RTCDataChannel`, but the application may change its value at any time.

***`onopen`* of type EventHandler**
The event type of this event handler is `open`.

***`onbufferedamountlow`* of type EventHandler**
The event type of this event handler is `bufferedamountlow`.

***`onerror`* of type EventHandler**
The event type of this event handler is `RTCErrorEvent`. `errorDetail` contains "sctp-failure", `sctpCauseCode` contains the SCTP Cause Code value, and `message` contains the SCTP Cause-Specific-Information, possibly with additional text.

***`onclosing`* of type EventHandler**
The event type of this event handler is `closing`.

***`onclose`* of type EventHandler**

The event type of this event handler is close.

***onmessage* of type EventHandler**
The event type of this event handler is `message`.

***binaryType* of type BinaryType**

PROPOSED CORRECTION 39: Fix binaryType setter requirements (PR #2909)

PROPOSED CORRECTION 40: Change the default value of binaryType (PR #2913)

Show Current and Future Show Current Show Future

The `binaryType` attribute ~~*MUST*, on getting, return~~ returns the value to which it was last set. ~~On setting, if the new value is either the string `"blob"` or the string `"arraybuffer"`, then set the IDL attribute to this new value. Otherwise, throw a `SyntaxError`.~~ When an `RTCDataChannel` object is created, the `binaryType` attribute *MUST* be initialized to the string ~~`"blob"`~~`"arraybuffer"`.

This attribute controls how binary data is exposed to scripts. See Web Socket's `binaryType`.

§ 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*.`[[ReadyState]]` is "`closing`" or "`closed`", then abort these steps.
3. Set *channel*.`[[ReadyState]]` to "`closing`".
4. If the closing procedure has not started yet, start it.

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

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

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

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

The `send()` method is overloaded to handle different data argument types. When any version of the method is called, the user agent *MUST* **run the following steps**:

1. Let *channel* be the `RTCDataChannel` object on which data is to be sent.
2. CANDIDATE ADDITION 48: Make RTCDataChannel transferable to DedicatedWorker (PR #2988)

Show Current and Future Show Current Show Future

Set *channel*.`[[IsTransferable]]` to `false`.

3. If *channel*.`[[ReadyState]]` is not "`open`", throw an `InvalidStateError`.
4. Execute the sub step that corresponds to the type of the methods argument:
   - `string` object:

Let *data* be a byte buffer that represents the result of encoding the method's argument as UTF-8.

- `Blob` object:

  Let *data* be the raw data represented by the `Blob` object.

  > ***NOTE***
  > ***Although the actual retrieval of data from a `Blob` object can happen asynchronously, the user agent will make sure to queue the data on the channel's underlying data transport in the same order as the send method is called. The byte size of data needs to be known synchronously.***

- `ArrayBuffer` object:

  Let *data* be the data stored in the buffer described by the `ArrayBuffer` object.

- `ArrayBufferView` object:

  Let *data* be the data stored in the section of the buffer described by the `ArrayBuffer` object that the `ArrayBufferView` object references.

> ***NOTE***
> ***Any data argument type this method has not been overloaded with will result in a `TypeError`. This includes `null` and `undefined`.***

5. If the byte size of *data* exceeds the value of `maxMessageSize` on *channel*'s associated `RTCSctpTransport`, throw a `TypeError`.

6. Queue *data* for transmission on *channel*'s underlying data transport. If queuing *data* is not possible because not enough buffer space is available, throw an `OperationError`.

> ***NOTE***
> ***The actual transmission of data occurs in parallel. If sending data leads to an SCTP-level error, the application will be notified asynchronously through `onerror`.***

7. Increase the value of the `[[BufferedAmount]]` slot by the byte size of *data*.

WebIDL

```
dictionary RTCDataChannelInit {
  boolean ordered = true;
  [EnforceRange] unsigned short maxPacketLifeTime;
  [EnforceRange] unsigned short maxRetransmits;
  USVString protocol = "";
  boolean negotiated = false;
  [EnforceRange] unsigned short id;
};
```

§ Dictionary `RTCDataChannelInit` Members

`ordered` **of type boolean, defaulting to `true`**
If set to false, data is allowed to be delivered out of order. The default value of true, guarantees that data will be delivered in order.

`maxPacketLifeTime` **of type unsigned short**
Limits the time (in milliseconds) during which the channel will transmit or retransmit data if not acknowledged. This value may be clamped if it exceeds the maximum value supported by the user agent.

`maxRetransmits` **of type unsigned short**
Limits the number of times a channel will retransmit data if not successfully delivered. This value may be clamped if it exceeds the maximum value supported by the user agent.

`protocol` **of type USVString, defaulting to `""`**

Subprotocol name used for this channel.

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

The default value of false tells the user agent to announce the channel in-band and instruct the other peer to dispatch a corresponding `RTCDataChannel` object. If set to true, it is up to the application to negotiate the channel and create an `RTCDataChannel` object with the same `id` at the other peer.

> ***NOTE***
> ***If set to true, the application must also take care to not send a message until the other peer has created a data channel to receive it. Receiving a message on an SCTP stream with no associated data channel is undefined behavior, and it may be silently dropped. This will not be possible as long as both endpoints create their data channel before the first offer/answer exchange is complete.***

id **of type unsigned short**

Sets the channel ID when `negotiated` is true. Ignored when `negotiated` is false.

WebIDL

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

`RTCDataChannelState` *Enumeration description*

Enum value	Description
`connecting`	The user agent is attempting to establish the underlying data transport. This is the initial state of an `RTCDataChannel` object, whether created with `createDataChannel`, or dispatched as a part of an `RTCDataChannelEvent`.
`open`	The underlying data transport is established and communication is possible.
`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.

§ 6.3 `RTCDataChannelEvent`

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

WebIDL

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

§ Constructors

`RTCDataChannelEvent.constructor()`

§ **Attributes**

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

WebIDL

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

§ **Dictionary `RTCDataChannelEventInit` Members**

***`channel`* of type `RTCDataChannel`, required**
The `RTCDataChannel` object to be announced by the event.

§ 6.4 Garbage Collection

An `RTCDataChannel` object *MUST* not be garbage collected if its

- `[[ReadyState]]` slot is "`connecting`" and at least one event listener is registered for `open` events, `message` events, `error` events, `closing` events, or `close` events.
- `[[ReadyState]]` slot is "`open`" and at least one event listener is registered for `message` events, `error` events, `closing` events, or `close` events.
- `[[ReadyState]]` slot is "`closing`" and at least one event listener is registered for `error` events, or `close` events.
- underlying data transport is established and data is queued to be transmitted.

§ 7. Peer-to-peer DTMF

This section describes an interface on `RTCRtpSender` to send DTMF (phone keypad) values across an `RTCPeerConnection`. Details of how DTMF is sent to the other peer are described in [RFC7874].

§ 7.1 RTCRtpSender Interface Extensions

The Peer-to-peer DTMF API extends the `RTCRtpSender` interface as described below.

WebIDL

```
          partial interface RTCRtpSender {
  readonly attribute RTCDTMFSender? dtmf;
};
```

§ **Attributes**

***`dtmf`* of type `RTCDTMFSender`, readonly, nullable**
On getting, the `dtmf` attribute returns the value of the `[[Dtmf]]` internal slot, which represents a `RTCDTMFSender` which can be used to send DTMF, or `null` if unset. The `[[Dtmf]]` internal slot is set when the kind of an `RTCRtpSender`'s `[[SenderTrack]]` is `"audio"`.

§ 7.2 RTCDTMFSender

To **create an RTCDTMFSender**, the user agent *MUST* run the following steps:

1. Let *dtmf* be a newly created RTCDTMFSender object.
2. Let *dtmf* have a **[[Duration]]** internal slot.
3. Let *dtmf* have a **[[InterToneGap]]** internal slot.
4. Let *dtmf* have a **[[ToneBuffer]]** internal slot.

WebIDL

```
[Exposed=Window]
interface RTCDTMFSender : EventTarget {
  undefined insertDTMF(DOMString tones, optional unsigned long duration = 100, optional
unsigned long interToneGap = 70);
  attribute EventHandler ontonechange;
  readonly attribute boolean canInsertDTMF;
  readonly attribute DOMString toneBuffer;
};
```

§ Attributes

ontonechange **of type EventHandler**
: The event type of this event handler is tonechange.

canInsertDTMF **of type boolean, readonly**
: Whether the RTCDTMFSender *dtmfSender* is capable of sending DTMF. On getting, the user agent *MUST* return the result of running determine if DTMF can be sent for *dtmfSender*.

toneBuffer **of type DOMString, readonly**
: The toneBuffer attribute *MUST* return a list of the tones remaining to be played out. For the syntax, content, and interpretation of this list, see insertDTMF.

§ Methods

insertDTMF

An RTCDTMFSender object's insertDTMF method is used to send DTMF tones.

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 *MUST* be normalized to uppercase on entry and are equivalent to A to D. As noted in [RTCWEB-AUDIO] Section 3, support for the characters 0 through 9, A through D, #, and * are required. The character ',' *MUST* be supported, and indicates a delay of 2 seconds before processing the next character in the tones parameter. All other characters (and only those 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 in ms. The user agent clamps it to at least 30 ms and at most 6000 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. Let *sender* be the `RTCRtpSender` used to send DTMF.
2. Let *transceiver* be the `RTCRtpTransceiver` object associated with *sender*.
3. Let *dtmf* be the `RTCDTMFSender` associated with *sender*.
4. If determine if DTMF can be sent for *dtmf* returns `false`, throw an `InvalidStateError`.
5. Let *tones* be the method's first argument.
6. Let *duration* be the method's second argument.
7. Let *interToneGap* be the method's third argument.
8. If *tones* contains any `unrecognized` characters, throw an `InvalidCharacterError`.
9. Set the object's `[[ToneBuffer]]` slot to *tones*.
10. Set *dtmf*.`[[Duration]]` to the value of *duration*.
11. Set *dtmf*.`[[InterToneGap]]` to the value of *interToneGap*.
12. If the value of *duration* is less than 40 ms, set *dtmf*.`[[Duration]]` to 40 ms.
13. If the value of *duration* parameter is greater than 6000 ms, set *dtmf*.`[[Duration]]` to 6000 ms.
14. If the value of *interToneGap* is less than 30 ms, set *dtmf*.`[[InterToneGap]]` to 30 ms.
15. If the value of *interToneGap* is greater than 6000 ms, set *dtmf*.`[[InterToneGap]]` to 6000 ms.
16. If [[ToneBuffer]] slot is an empty string, abort these steps.
17. If a task to run the DTMF playout task steps is scheduled to be run, abort these steps; otherwise queue a task that runs the following **DTMF playout task steps**:

> CANDIDATE CORRECTION 33: Determine if DTMF can be sent inside queued playout task (PR #2861)
>
> ◉ Show Current and Future ○ Show Current ○ Show Future
>
> 1. ~~If *transceiver*.[[CurrentDirection]]~~If determine if DTMF can be sent ~~is neither "`sendrecv`" nor "`sendonly`"~~for *dtmf* returns `false`, abort these steps.
> 2. If the `[[ToneBuffer]]` slot contains the empty string, fire an event named `tonechange` using the `RTCDTMFToneChangeEvent` interface with the `tone` attribute set to an empty string at the `RTCDTMFSender` object and abort these steps.
> 3. Remove the first character from the `[[ToneBuffer]]` slot and let that character be *tone*.
> 4. If *tone* is `","` delay sending tones for `2000` ms on the associated RTP media stream, and queue a task to be executed in `2000` ms from now that runs the DTMF playout task steps.
> 5. If *tone* is not `","` start playout of *tone* for `[[Duration]]` ms on the associated RTP media stream, using the appropriate codec, then queue a task to be executed in `[[Duration]]` + `[[InterToneGap]]` ms from now that runs the DTMF playout task steps.
> 6. Fire an event named `tonechange` using the `RTCDTMFToneChangeEvent` interface with the `tone` attribute set to *tone* at the `RTCDTMFSender` object.

Since `insertDTMF` replaces the tone buffer, in order to add to the DTMF tones being played, it is necessary to call `insertDTMF` with a string containing both the remaining tones (stored in the `[[ToneBuffer]]` slot) and the new tones appended together. Calling `insertDTMF` with an empty tones parameter can be used to cancel all tones queued to play after the currently playing tone.

§ 7.3 canInsertDTMF algorithm

PROPOSED CORRECTION 9: No longer queue a task in the determine DTMF algorithm (PR #2742)

◉ Show Current and Future ○ Show Current ○ Show Future

To **determine if DTMF can be sent** for an `RTCDTMFSender` instance *dtmfSender*, the user agent *MUST* ~~queue a task that runs~~ run the ~~following~~ following steps:

1. Let *sender* be the `RTCRtpSender` associated with *dtmfSender*.
2. Let *transceiver* be the `RTCRtpTransceiver` associated with *sender*.
3. Let *connection* be the `RTCPeerConnection` associated with *transceiver*.
4. If *connection*'s `RTCPeerConnectionState` is not "`connected`" return `false`.
5. **PROPOSED CORRECTION 32:** Reject setParameters(), replaceTrack(), & insertDTMF() after stop() (PR #2829)

   Show Current and Future Show Current Show Future

   If *transceiver*.`[[Stopping]]` is `true` return `false`.
6. If *sender*.`[[SenderTrack]]` is `null` return `false`.
7. If *transceiver*.`[[CurrentDirection]]` is neither "`sendrecv`" nor "`sendonly`" return `false`.
8. If *sender*.`[[SendEncodings]][0].active` is `false` return `false`.
9. If no codec with mimetype `"audio/telephone-event"` has been negotiated for sending with this *sender*, return `false`.
10. Otherwise, return `true`.

§ 7.4 `RTCDTMFToneChangeEvent`

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

WebIDL

```
[Exposed=Window]
interface RTCDTMFToneChangeEvent : Event {
  constructor(DOMString type, optional RTCDTMFToneChangeEventInit eventInitDict = {});
  readonly attribute DOMString tone;
};
```

§ Constructors

`RTCDTMFToneChangeEvent.constructor()`

§ Attributes

***tone* of type DOMString, readonly**

The `tone` attribute contains the character for the tone (including `","`) that has just begun playout (see `insertDTMF`). If the value is the empty string, it indicates that the `[[ToneBuffer]]` slot is an empty string and that the previous tones have completed playback.

WebIDL

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

§ Dictionary `RTCDTMFToneChangeEventInit` Members

***tone* of type DOMString, defaulting to `""`**

The `tone` attribute contains the character for the tone (including `","`) that has just begun playout (see `insertDTMF`). If the value is the empty string, it indicates that the `[[ToneBuffer]]` slot is an empty string and that the previous tones have completed playback.

§ 8. Statistics Model

§ 8.1 Introduction

The basic statistics model is that the browser maintains a set of statistics for monitored objects, in the form of stats objects.

A group of related objects may be referenced by a **selector**. The selector may, for example, be a `MediaStreamTrack`. For a track to be a valid selector, it *MUST* be a `MediaStreamTrack` that is sent or received by the `RTCPeerConnection` object on which the stats request was issued. The calling Web application provides the selector to the `getStats()` method and the browser emits (in the JavaScript) a set of statistics that are relevant to the selector, according to the stats selection algorithm. Note that that algorithm takes the sender or receiver of a selector.

The statistics returned in stats objects are designed in such a way that repeated queries can be linked by the `RTCStats` `id` dictionary member. Thus, a Web application can make measurements over a given time period by requesting measurements at the beginning and end of that period.

With a few exceptions, monitored objects, once created, exist for the duration of their associated `RTCPeerConnection`. This ensures statistics from them are available in the result from `getStats()` even past the associated peer connection being `close`d.

Only a few monitored objects have shorter lifetimes. Statistics from these objects are no longer available in subsequent getStats() results. The object descriptions in [WEBRTC-STATS] describe when these monitored objects are deleted.

§ 8.2 RTCPeerConnection Interface Extensions

The Statistics API extends the `RTCPeerConnection` interface as described below.

WebIDL

```
        partial interface RTCPeerConnection {
  Promise<RTCStatsReport> getStats(optional MediaStreamTrack? selector = null);
};
```

§ Methods

`getStats`

Gathers stats for the given selector and reports the result asynchronously.

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

1. Let *selectorArg* be the method's first argument.
2. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.
3. If *selectorArg* is `null`, let *selector* be `null`.
4. If *selectorArg* is a `MediaStreamTrack` let *selector* be an `RTCRtpSender` or `RTCRtpReceiver` on *connection* which `track` attribute matches *selectorArg*. If no such sender or receiver exists, or if more than one sender or receiver fit this criteria, return a promise rejected with a newly created `InvalidAccessError`.
5. Let *p* be a new promise.

6. Run the following steps in parallel:
    1. Gather the stats indicated by *selector* according to the stats selection algorithm.
    2. Resolve *p* with the resulting `RTCStatsReport` object, containing the gathered stats.
7. Return *p*.

§ 8.3 `RTCStatsReport` Object

The `getStats()` method delivers a successful result in the form of an `RTCStatsReport` object. An `RTCStatsReport` object is a map between strings that identify the inspected objects (`id` attribute in `RTCStats` instances), and their corresponding `RTCStats`-derived dictionaries.

An `RTCStatsReport` may be composed of several `RTCStats`-derived dictionaries, each reporting stats for one underlying object that the implementation thinks is relevant for the selector. One achieves the total for the selector by summing over all the stats of a certain type; for instance, if an `RTCRtpSender` uses multiple SSRCs to carry its track over the network, the `RTCStatsReport` may contain one `RTCStats`-derived dictionary per SSRC (which can be distinguished by the value of the `ssrc` stats attribute).

WebIDL

```
[Exposed=Window]
interface RTCStatsReport {
  readonly maplike<DOMString, object>;
};
```

Use these to retrieve the various dictionaries descended from `RTCStats` that this stats report is composed of. The set of supported property names [WEBIDL] is defined as the ids of all the `RTCStats`-derived dictionaries that have been generated for this stats report.

§ 8.4 `RTCStats` Dictionary

An `RTCStats` dictionary represents the stats object constructed by inspecting a specific monitored 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 an `RTCStats`-derived dictionary.

WebIDL

```
dictionary RTCStats {
  required DOMHighResTimeStamp timestamp;
  required RTCStatsType type;
  required DOMString id;
};
```

§ Dictionary `RTCStats` Members

`timestamp` **of type DOMHighResTimeStamp**

CANDIDATE CORRECTION 50: Use Performance.timeOrigin + Performance.now() for stats timestamps (PR #3005)

◉ Show Current and Future ○ Show Current ○ Show Future

~~The timestamp, of type~~ Timestamps are expressed with DOMHighResTimeStamp~~, associated with this object. The time is relative to the UNIX epoch (Jan 1~~ [HIGHRES-TIME], ~~1970,~~ and are defined as Performance.timeOrigin ~~UTC)~~ + Performance.now() at the time the information is collected. For statistics that came from a remote source (e.g., from received RTCP packets), timestamp represents the time at which the information arrived at the local endpoint. The remote timestamp can be found in an additional field in an RTCStats-derived dictionary, if applicable.

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.

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.

Stats ids *MUST NOT* be predictable by an application. This prevents applications from depending on a particular user agent's way of generating ids, since this prevents an application from getting stats objects by their id unless they have already read the id of that specific stats object.

User agents are free to pick any format for the id as long as it meets the requirements above.

> ***NOTE***
>
> ***A user agent can turn a predictably generated string into an unpredictable string using a hash function, as long as it uses a salt that is unique to the peer connection. This allows an implementation to have predictable ids internally, which may make it easier to guarantee that stats objects have stable ids across getStats() calls.***

The set of valid values for RTCStatsType, and the dictionaries derived from RTCStats that they indicate, are documented in [WEBRTC-STATS].

§ 8.5 The stats selection algorithm

The **stats selection algorithm** is as follows:

1. Let *result* be an empty RTCStatsReport.
2. If *selector* is null, gather stats for the whole *connection*, add them to *result*, return *result*, and abort these steps.
3. If *selector* is an RTCRtpSender, gather stats for and add the following objects to *result*:
   - All RTCOutboundRtpStreamStats objects representing RTP streams being sent by *selector*.
   - All stats objects referenced directly or indirectly by the RTCOutboundRtpStreamStats objects added.
4. If *selector* is an RTCRtpReceiver, gather stats for and add the following objects to *result*:
   - All RTCInboundRtpStreamStats objects representing RTP streams being received by *selector*.
   - All stats objects referenced directly or indirectly by the RTCInboundRtpStreamStats added.
5. Return *result*.

§ 8.6 Mandatory To Implement Stats

The stats listed in [WEBRTC-STATS] are intended to cover a wide range of use cases. Not all of them have to be implemented by every WebRTC implementation.

An implementation *MUST* support generating statistics of the following `type`s when the corresponding objects exist on a `RTCPeerConnection`, with the fields that are listed when they are valid for that object in addition to the generic fields defined in the `RTCStats` dictionary:

PROPOSED CORRECTION 10: Align MTI stats with implementations (PR #2744, PR #2748, PR #2832)

Show Current and Future Show Current Show Future

RTCStatsType	Dictionary	Fields
"codec"	RTCCodecStats	payloadType~~, codecType,~~ mimeType, clockRate, channels, sdpFmtpLine
"inbound-rtp"	RTCRtpStreamStats	ssrc, kind, transportId, codecId
	RTCReceivedRtpStreamStats	packetsReceived, packetsLost, jitter, ~~packetsDiscarded,~~ ~~framesDropped~~
	RTCInboundRtpStreamStats	~~receiverId~~trackIdentifier, remoteId, framesDecoded, framesDropped nackCount, framesReceived, bytesReceived, totalAudioEnergy, totalSamplesDuration packetsDiscarded,
"outbound-rtp"	RTCRtpStreamStats	ssrc, kind, transportId, codecId
	RTCSentRtpStreamStats	packetsSent, bytesSent
	RTCOutboundRtpStreamStats	~~senderId,~~ remoteId, framesEncoded, nackCount, framesSent
"remote-inbound-rtp"	RTCRtpStreamStats	ssrc, kind, transportId, codecId
	RTCReceivedRtpStreamStats	packetsReceived, packetsLost, jitter~~,~~ ~~packetsDiscarded,~~ ~~framesDropped~~
	RTCRemoteInboundRtpStreamStats	localId, roundTripTime
"remote-outbound-rtp"	RTCRtpStreamStats	ssrc, kind, transportId, codecId
	RTCSentRtpStreamStats	packetsSent, bytesSent
	RTCRemoteOutboundRtpStreamStats	localId, remoteTimestamp
"media-source"	RTCMediaSourceStats	trackIdentifier, kind
	RTCAudioSourceStats	totalAudioEnergy, totalSamplesDuration (for audio tracks attached to senders)
	RTCVideoSourceStats	width, height, framesPerSecond (for video tracks attached to senders)

RTCStatsType	Dictionary	Fields
"peer-connection"	RTCPeerConnectionStats	dataChannelsOpened, dataChannelsClosed
"data-channel"	RTCDataChannelStats	label , protocol, dataChannelIdentifier, state, messagesSent, bytesSent, messagesReceived, bytesReceived
~~"sender"~~	~~RTCMediaHandlerStats~~	~~trackIdentifier~~
"~~receiver~~" ~~"~~transport"	RTCTransportStats	bytesSent, bytes selectedCandida localCertificat remoteCertifica
"candidate-pair"	RTCIceCandidatePairStats	transportId, localCandidateId, remoteCandidateId, state, nominated, bytesSent, bytesReceived, totalRoundTripTime, responsesReceived, currentRoundTripTime
"local-candidate"	RTCIceCandidateStats	address, port, protocol, candidateType, url
"remote-candidate"		
"certificate"	RTCCertificateStats	fingerprint, fingerprintAlgorithm, base64Certificate, issuerCertificateId

An implementation *MAY* support generating any other statistic defined in [WEBRTC-STATS], and *MAY* generate statistics that are not documented.

§ 8.7 GetStats 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 8

```
async function gatherStats(pc) {
  try {
    const [sender] = pc.getSenders();
    const baselineReport = await sender.getStats();
    await new Promise(resolve => setTimeout(resolve, aBit)); // wait a bit
    const currentReport = await sender.getStats();

    // compare the elements from the current report with the baseline
    for (const now of currentReport.values()) {
      if (now.type != 'outbound-rtp') continue;

      // get the corresponding stats from the baseline report
      const base = baselineReport.get(now.id);
      if (!base) continue;

      const remoteNow = currentReport.get(now.remoteId);
      const remoteBase = baselineReport.get(base.remoteId);

      const packetsSent = now.packetsSent - base.packetsSent;
      const packetsReceived = remoteNow.packetsReceived -
                              remoteBase.packetsReceived;

      const fractionLost = (packetsSent - packetsReceived) / packetsSent;
      if (fractionLost > 0.3) {
        // if fractionLost is > 0.3, we have probably found the culprit
      }
    }
  } catch (err) {
    console.error(err);
  }
}
```

§ 9. Media Stream API Extensions for Network Use

§ 9.1 Introduction

The `MediaStreamTrack` interface, as defined in the [GETUSERMEDIA] specification, typically represents a stream of data of audio or video. One or more `MediaStreamTrack`s can be collected in a `MediaStream` (strictly speaking, a `MediaStream` as defined in [GETUSERMEDIA] may contain zero or more `MediaStreamTrack` objects).

A `MediaStreamTrack` may be extended to represent a media flow that either comes from or is sent to a remote peer (and not just the local camera, for instance). The extensions required to enable this capability on the `MediaStreamTrack` object will be described in this section. How the media is transmitted to the peer is described in [RFC8834], [RFC7874], and [RFC8835].

A `MediaStreamTrack` sent to another peer will appear as one and only one `MediaStreamTrack` to the recipient. A peer is defined as a user agent that supports this specification. In addition, the sending side application can indicate what `MediaStream` object(s) the `MediaStreamTrack` is a member of. The corresponding `MediaStream` object(s) on the receiver side will be created (if not already present) and populated accordingly.

As also described earlier in this document, the objects `RTCRtpSender` and `RTCRtpReceiver` can be used by the application to get more fine grained control over the transmission and reception of `MediaStreamTrack`s.

Channels are the smallest unit considered in the Media Capture and Streams specification. Channels are intended to be encoded together for transmission as, for instance, an RTP payload type. All of the channels that a codec needs to encode jointly *MUST* be in the same `MediaStreamTrack` and the codecs

SHOULD be able to encode, or discard, all the channels in the track.

The concepts of an input and output to a given `MediaStreamTrack` apply in the case of `MediaStreamTrack` objects transmitted over the network as well. A `MediaStreamTrack` created by an `RTCPeerConnection` object (as described previously in this document) will take as input the data received from a remote peer. Similarly, a `MediaStreamTrack` from a local source, for instance a camera via [GETUSERMEDIA], will have an output that represents what is transmitted to a remote peer if the object is used with an `RTCPeerConnection` object.

The concept of duplicating `MediaStream` and `MediaStreamTrack` objects as described in [GETUSERMEDIA] is also applicable here. This feature can be used, for instance, in a video-conferencing scenario to display the local video from the user's camera and microphone in a local monitor, while only transmitting the audio to the remote peer (e.g. in response to the user using a "video mute" feature). Combining different `MediaStreamTrack` objects into new `MediaStream` objects is useful in certain situations.

> ***NOTE***
>
> ***In this document, we only specify aspects of the following objects that are relevant when used along with an `RTCPeerConnection`. Please refer to the original definitions of the objects in the [GETUSERMEDIA] document for general information on using `MediaStream` and `MediaStreamTrack`.***

§ 9.2 MediaStream

§ 9.2.1 id

The `id` attribute specified in `MediaStream` returns an id that is unique to this stream, so that streams can be recognized at the remote end of the `RTCPeerConnection` API.

When a `MediaStream` is created to represent a stream obtained from a remote peer, the `id` attribute is initialized from information provided by the remote source.

> ***NOTE***
>
> ***The `id` of a `MediaStream` object is unique to the source of the stream, but that does not mean it is not possible to end up with duplicates. For example, the tracks of a locally generated stream could be sent from one user agent to a remote peer using `RTCPeerConnection` and then sent back to the original user agent in the same manner, in which case the original user agent will have multiple streams with the same id (the locally-generated one and the one received from the remote peer).***

§ 9.3 MediaStreamTrack

A `MediaStreamTrack` object's reference to its `MediaStream` in the non-local media source case (an RTP source, as is the case for each `MediaStreamTrack` associated with an `RTCRtpReceiver`) is always strong.

Whenever an `RTCRtpReceiver` receives data on an RTP source whose corresponding `MediaStreamTrack` is muted, but not ended, and the `[[Receptive]]` slot of the `RTCRtpTransceiver` object the `RTCRtpReceiver` is a member of is `true`, it *MUST* queue a task to set the muted state of the corresponding `MediaStreamTrack` to `false`.

When one of the SSRCs for RTP source media streams received by an `RTCRtpReceiver` is removed either due to reception of a BYE or via timeout, it *MUST* queue a task to set the muted state of the corresponding `MediaStreamTrack` to `true`. Note that `setRemoteDescription` can also lead to the setting of the muted state of the `track` to the value `true`.

The procedures **add a track**, **remove a track** and **set a track's muted state** are specified in [GETUSERMEDIA].

When a `MediaStreamTrack` track produced by an `RTCRtpReceiver` *receiver* has ended [GETUSERMEDIA] (such as via a call to *receiver*.`track`.`stop`), the user agent *MAY* choose to free resources allocated for the incoming stream, by for instance turning off the decoder of *receiver*.

§ 9.3.1 MediaTrackSupportedConstraints, MediaTrackCapabilities, MediaTrackConstraints and MediaTrackSettings

The concept of constraints and constrainable properties, including `MediaTrackConstraints` (`MediaStreamTrack.getConstraints()`, `MediaStreamTrack.applyConstraints()`), and `MediaTrackSettings` (`MediaStreamTrack.getSettings()`) are outlined in [GETUSERMEDIA]. However, the constrainable properties of tracks sourced from a peer connection are different than those sourced by `getUserMedia()`; the constraints and settings applicable to `MediaStreamTrack`s sourced from a remote source are defined here. The settings of a remote track represent the latest frame received.

`MediaStreamTrack.getCapabilities()` *MUST* always return the empty set and `MediaStreamTrack.applyConstraints()` *MUST* always reject with `OverconstrainedError` on remote tracks for constraints defined here.

The following constrainable properties are defined to apply to video `MediaStreamTrack`s sourced from a remote source:

Property Name	Values	Notes
width	`ConstrainULong`	As a setting, this is the width, in pixels, of the latest frame received.
height	`ConstrainULong`	As a setting, this is the height, in pixels, of the latest frame received.
frameRate	`ConstrainDouble`	As a setting, this is an estimate of the frame rate based on recently received frames.
aspectRatio	`ConstrainDouble`	As a setting, this is the aspect ratio of the latest frame; this is the width in pixels divided by height in pixels as a double rounded to the tenth decimal place.

This document does not define any constrainable properties to apply to audio `MediaStreamTrack`s sourced from a remote source.

§ 10. Examples and Call Flows

This section is non-normative.

§ 10.1 Simple Peer-to-peer Example

When two peers decide they are going to set up a connection to each other, they both go through these steps. The STUN/TURN server configuration describes a server they can use to get things like their public IP address or to set up NAT traversal. They also have to send data for the signaling channel to each other using the same out-of-band mechanism they used to establish that they were going to communicate in the first place.

EXAMPLE 9

```
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const constraints = {audio: true, video: true};
const configuration = {iceServers: [{urls: 'stun:stun.example.org'}]};
const pc = new RTCPeerConnection(configuration);

// send any ice candidates to the other peer
pc.onicecandidate = ({candidate}) => signaling.send({candidate});

// let the "negotiationneeded" event trigger offer generation
pc.onnegotiationneeded = async () => {
  try {
    await pc.setLocalDescription();
    // send the offer to the other peer
    signaling.send({description: pc.localDescription});
  } catch (err) {
    console.error(err);
  }
};

pc.ontrack = ({track, streams}) => {
  // once media for a remote track arrives, show it in the remote video element
  track.onunmute = () => {
    // don't set srcObject again if it is already set.
    if (remoteView.srcObject) return;
    remoteView.srcObject = streams[0];
  };
};

// call start() to initiate
function start() {
  addCameraMic();
}

// add camera and microphone to connection
async function addCameraMic() {
  try {
    // get a local stream, show it in a self-view and add it to be sent
    const stream = await navigator.mediaDevices.getUserMedia(constraints);
    for (const track of stream.getTracks()) {
      pc.addTrack(track, stream);
    }
    selfView.srcObject = stream;
  } catch (err) {
    console.error(err);
  }
}

signaling.onmessage = async ({data: {description, candidate}}) => {
  try {
    if (description) {
      await pc.setRemoteDescription(description);
      // if we got an offer, we need to reply with an answer
      if (description.type == 'offer') {
        if (!selfView.srcObject) {
          // blocks negotiation on permission (not recommended in production code)
          await addCameraMic();
        }
        await pc.setLocalDescription();
        signaling.send({description: pc.localDescription});
      }
    } else if (candidate) {
      await pc.addIceCandidate(candidate);
    }
  } catch (err) {
    console.error(err);
  }
```

```
};
```

§ 10.2 Advanced Peer-to-peer Example with Warm-up

When two peers decide they are going to set up a connection to each other and want to have the ICE, DTLS, and media connections "warmed up" such that they are ready to send and receive media immediately, they both go through these steps.

EXAMPLE 10

```
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const constraints = {audio: true, video: true};
const configuration = {iceServers: [{urls: 'stun:stun.example.org'}]};
let pc;
let audio;
let video;
let started = false;

// Call warmup() before media is ready, to warm-up ICE, DTLS, and media.
async function warmup(isAnswerer) {
  pc = new RTCPeerConnection(configuration);
  if (!isAnswerer) {
    audio = pc.addTransceiver('audio');
    video = pc.addTransceiver('video');
  }

  // send any ice candidates to the other peer
  pc.onicecandidate = ({candidate}) => signaling.send({candidate});

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription();
      // send the offer to the other peer
      signaling.send({description: pc.localDescription});
    } catch (err) {
      console.error(err);
    }
  };

  pc.ontrack = async ({track, transceiver}) => {
    try {
      // once media for the remote track arrives, show it in the video element
      event.track.onunmute = () => {
        // don't set srcObject again if it is already set.
        if (!remoteView.srcObject) {
          remoteView.srcObject = new MediaStream();
        }
        remoteView.srcObject.addTrack(track);
      }

      if (isAnswerer) {
        if (track.kind == 'audio') {
          audio = transceiver;
        } else if (track.kind == 'video') {
          video = transceiver;
        }
        if (started) await addCameraMicWarmedUp();
      }
    } catch (err) {
      console.error(err);
    }
  };

  try {
    // get a local stream, show it in a self-view and add it to be sent
    selfView.srcObject = await navigator.mediaDevices.getUserMedia(constraints);
    if (started) await addCameraMicWarmedUp();
  } catch (err) {
    console.error(err);
  }
}

// call start() after warmup() to begin transmitting media from both ends
function start() {
  signaling.send({start: true});
```

```
    signaling.onmessage({data: {start: true}});
  }

  // add camera and microphone to already warmed-up connection
  async function addCameraMicWarmedUp() {
    const stream = selfView.srcObject;
    if (audio && video && stream) {
      await Promise.all([
        audio.sender.replaceTrack(stream.getAudioTracks()[0]),
        video.sender.replaceTrack(stream.getVideoTracks()[0]),
      ]);
    }
  }

  signaling.onmessage = async ({data: {start, description, candidate}}) => {
    if (!pc) warmup(true);

    try {
      if (start) {
        started = true;
        await addCameraMicWarmedUp();
      } else if (description) {
        await pc.setRemoteDescription(description);
        // if we got an offer, we need to reply with an answer
        if (description.type == 'offer') {
          await pc.setLocalDescription();
          signaling.send({description: pc.localDescription});
        }
      } else {
        await pc.addIceCandidate(candidate);
      }
    } catch (err) {
      console.error(err);
    }
  };
```

§ 10.3 Simulcast Example

A client wants to send multiple RTP encodings (simulcast) to a server.

EXAMPLE 11

```
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const constraints = {audio: true, video: true};
const configuration = {'iceServers': [{'urls': 'stun:stun.example.org'}]};
let pc;

// call start() to initiate
async function start() {
  pc = new RTCPeerConnection(configuration);

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription();
      // send the offer to the other peer
      signaling.send({description: pc.localDescription});
    } catch (err) {
      console.error(err);
    }
  };

  try {
    // get a local stream, show it in a self-view and add it to be sent
    const stream = await navigator.mediaDevices.getUserMedia(constraints);
    selfView.srcObject = stream;
    pc.addTransceiver(stream.getAudioTracks()[0], {direction: 'sendonly'});
    pc.addTransceiver(stream.getVideoTracks()[0], {
      direction: 'sendonly',
      sendEncodings: [
        {rid: 'q', scaleResolutionDownBy: 4.0}
        {rid: 'h', scaleResolutionDownBy: 2.0},
        {rid: 'f'},
      ]
    });
  } catch (err) {
    console.error(err);
  }
}

signaling.onmessage = async ({data: {description, candidate}}) => {
  try {
    if (description) {
      await pc.setRemoteDescription(description);
      // if we got an offer, we need to reply with an answer
      if (description.type == 'offer') {
        await pc.setLocalDescription();
        signaling.send({description: pc.localDescription});
      }
    } else if (candidate) {
      await pc.addIceCandidate(candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

§ 10.4 Peer-to-peer Data Example

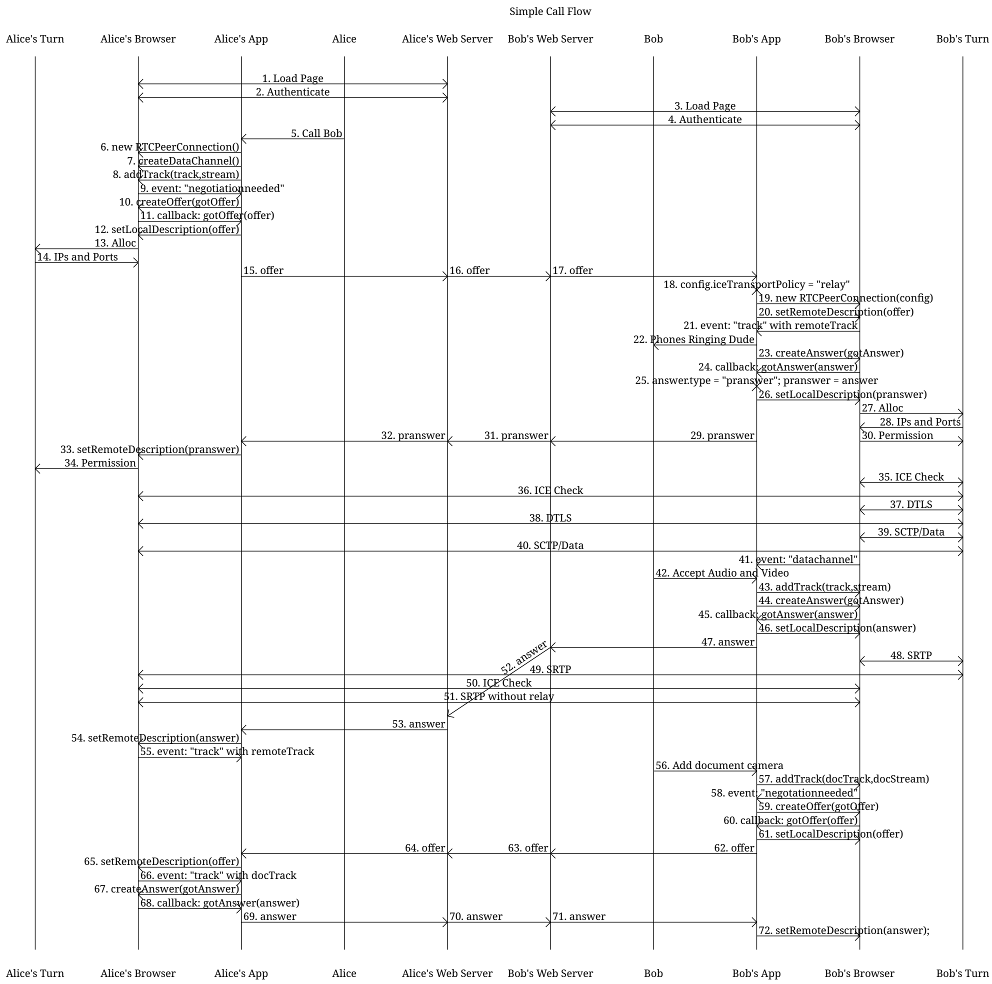
This example shows how to create an `RTCDataChannel` object and perform the offer/answer exchange required to connect the channel to the other peer. The `RTCDataChannel` is used in the context of a simple chat application using an `input` field for user input.

EXAMPLE 12

```
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const configuration = {iceServers: [{urls: 'stun:stun.example.org'}]};
let pc, channel;

// call start() to initiate
function start() {
  pc = new RTCPeerConnection(configuration);

  // send any ice candidates to the other peer
  pc.onicecandidate = ({candidate}) => signaling.send({candidate});

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription();
      // send the offer to the other peer
      signaling.send({description: pc.localDescription});
    } catch (err) {
      console.error(err);
    }
  };

  // create data channel and setup chat using "negotiated" pattern
  channel = pc.createDataChannel('chat', {negotiated: true, id: 0});
  channel.onopen = () => input.disabled = false;
  channel.onmessage = ({data}) => showChatMessage(data);

  input.onkeydown = ({key}) => {
    if (key != 'Enter') return;
    channel.send(input.value);
  }
}

signaling.onmessage = async ({data: {description, candidate}}) => {
  if (!pc) start();

  try {
    if (description) {
      await pc.setRemoteDescription(description);
      // if we got an offer, we need to reply with an answer
      if (description.type == 'offer') {
        await pc.setLocalDescription();
        signaling.send({description: pc.localDescription});
      }
    } else if (candidate) {
      await pc.addIceCandidate(candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

§ 10.5 Call Flow Browser to Browser

This shows an example of one possible call flow between two browsers. This does not show the procedure to get access to local media or every callback that gets fired but instead tries to reduce it down to only show the key events and messages.



§ 10.6 DTMF Example

Examples assume that *sender* is an `RTCRtpSender`.

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

EXAMPLE 13

```
if (sender.dtmf.canInsertDTMF) {
  const duration = 500;
  sender.dtmf.insertDTMF('1234', duration);
} else {
  console.log('DTMF function not available');
}
```

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

EXAMPLE 14

```
async function sendDTMF() {
  if (sender.dtmf.canInsertDTMF) {
    sender.dtmf.insertDTMF('123');
    await new Promise(r => sender.dtmf.ontonechange = e => e.tone == '2' && r());
    // empty the buffer to not play any tone after "2"
    sender.dtmf.insertDTMF('');
  } else {
    console.log('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 15

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

if (sender.dtmf.canInsertDTMF) {
  const duration = 500; // ms
  sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '1234', duration);
  sender.dtmf.ontonechange = async ({tone}) => {
    if (!tone) return;
    lightKey(tone); // light up the key when playout starts
    await wait(duration);
    lightKey(''); // turn off the light after tone duration
  };
} else {
  console.log('DTMF function not available');
}
```

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 16

```
if (sender.dtmf.canInsertDTMF) {
  sender.dtmf.insertDTMF('123');
  // append more tones to the tone buffer before playout has begun
  sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '456');

  sender.dtmf.ontonechange = ({tone}) => {
    // append more tones when playout has begun
    if (tone != '1') return;
    sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '789');
  };
} else {
  console.log('DTMF function not available');
}
```

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

EXAMPLE 17

```
if (sender.dtmf.canInsertDTMF) {
  sender.dtmf.ontonechange = ({tone}) => {
    if (tone == '1') {
      sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '2', 2000);
    }
  };
  sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '1', 1000);
} else {
  console.log('DTMF function not available');
}
```

§ 10.7 Perfect Negotiation Example

Perfect negotiation is a recommended pattern to manage negotiation transparently, abstracting this asymmetric task away from the rest of an application. This pattern has advantages over one side always being the offerer, as it lets applications operate on both peer connection objects simultaneously without risk of glare (an offer coming in outside of "`stable`" state). The rest of the application may use any and all modification methods and attributes, without worrying about signaling state races.

It designates different roles to the two peers, with behavior to resolve signaling collisions between them:

1. The polite peer uses rollback to avoid collision with an incoming offer.
2. The impolite peer ignores an incoming offer when this would collide with its own.

Together, they manage signaling for the rest of the application in a manner that doesn't deadlock. The example assumes a *polite* boolean variable indicating the designated role:

EXAMPLE 18

```
const signaling = new SignalingChannel(); // handles JSON.stringify/parse
const constraints = {audio: true, video: true};
const configuration = {iceServers: [{urls: 'stun:stun.example.org'}]};
const pc = new RTCPeerConnection(configuration);

// call start() anytime on either end to add camera and microphone to connection
async function start() {
  try {
    const stream = await navigator.mediaDevices.getUserMedia(constraints);
    for (const track of stream.getTracks()) {
      pc.addTrack(track, stream);
    }
    selfView.srcObject = stream;
  } catch (err) {
    console.error(err);
  }
}

pc.ontrack = ({track, streams}) => {
  // once media for a remote track arrives, show it in the remote video element
  track.onunmute = () => {
    // don't set srcObject again if it is already set.
    if (remoteView.srcObject) return;
    remoteView.srcObject = streams[0];
  };
};

// - The perfect negotiation logic, separated from the rest of the application ---

// keep track of some negotiation state to prevent races and errors
let makingOffer = false;
let ignoreOffer = false;
let isSettingRemoteAnswerPending = false;

// send any ice candidates to the other peer
pc.onicecandidate = ({candidate}) => signaling.send({candidate});

// let the "negotiationneeded" event trigger offer generation
pc.onnegotiationneeded = async () => {
  try {
    makingOffer = true;
    await pc.setLocalDescription();
    signaling.send({description: pc.localDescription});
  } catch (err) {
     console.error(err);
  } finally {
    makingOffer = false;
  }
};

signaling.onmessage = async ({data: {description, candidate}}) => {
  try {
    if (description) {
      // An offer may come in while we are busy processing SRD(answer).
      // In this case, we will be in "stable" by the time the offer is processed
      // so it is safe to chain it on our Operations Chain now.
      const readyForOffer =
          !makingOffer &&
          (pc.signalingState == "stable" || isSettingRemoteAnswerPending);
      const offerCollision = description.type == "offer" && !readyForOffer;

      ignoreOffer = !polite && offerCollision;
      if (ignoreOffer) {
        return;
      }
      isSettingRemoteAnswerPending = description.type == "answer";
```

```
          await pc.setRemoteDescription(description); // SRD rolls back as needed
          isSettingRemoteAnswerPending = false;
          if (description.type == "offer") {
            await pc.setLocalDescription();
            signaling.send({description: pc.localDescription});
          }
        } else if (candidate) {
          try {
            await pc.addIceCandidate(candidate);
          } catch (err) {
            if (!ignoreOffer) throw err; // Suppress ignored offer's candidates
          }
        }
      } catch (err) {
        console.error(err);
      }
    }
```

Note that this is timing sensitive, and deliberately uses versions of `setLocalDescription` (without arguments) and `setRemoteDescription` (with implicit rollback) to avoid races with other signaling messages being serviced.

The *ignoreOffer* variable is needed, because the `RTCPeerConnection` object on the impolite side is never told about ignored offers. We must therefore suppress errors from incoming candidates belonging to such offers.

§ 11. Error Handling

Some operations throw or fire `RTCError`. This is an extension of `DOMException` that carries additional WebRTC-specific information.

§ 11.1 `RTCError` Interface

WebIDL

```
[Exposed=Window]
interface RTCError : DOMException {
  constructor(RTCErrorInit init, optional DOMString message = "");
  readonly attribute RTCErrorDetailType errorDetail;
  readonly attribute long? sdpLineNumber;
  readonly attribute long? sctpCauseCode;
  readonly attribute unsigned long? receivedAlert;
  readonly attribute unsigned long? sentAlert;
};
```

§ 11.1.1 Constructors

`constructor()`

Run the following steps:

1. Let *init* be the constructor's first argument.
2. Let *message* be the constructor's second argument.
3. Let *e* be a new `RTCError` object.
4. Invoke the `DOMException` constructor of *e* with the `message` argument set to *message* and the `name` argument set to `"OperationError"`.

> ***NOTE***
>
> ***This name does not have a mapping to a legacy code so `e.code` will return 0.***

5. Set all `RTCError` attributes of *e* to the value of the corresponding attribute in *init* if it is present, otherwise set it to `null`.

6. Return *e*.

§ 11.1.2 Attributes

`errorDetail` **of type RTCErrorDetailType, readonly**
: The WebRTC-specific error code for the type of error that occurred.

`sdpLineNumber` **of type long, readonly, nullable**
: If `errorDetail` is "`sdp-syntax-error`" this is the line number where the error was detected (the first line has line number 1).

`sctpCauseCode` **of type long, readonly, nullable**
: If `errorDetail` is "`sctp-failure`" this is the SCTP cause code of the failed SCTP negotiation.

`receivedAlert` **of type unsigned long, readonly, nullable**
: If `errorDetail` is "`dtls-failure`" and a fatal DTLS alert was received, this is the value of the DTLS alert received.

`sentAlert` **of type unsigned long, readonly, nullable**
: If `errorDetail` is "`dtls-failure`" and a fatal DTLS alert was sent, this is the value of the DTLS alert sent.

> (FEATURE AT RISK) ISSUE 1
>
> All attributes defined in `RTCError` are marked at risk due to lack of implementation (`errorDetail`, `sdpLineNumber`, `sctpCauseCode`, `receivedAlert` and `sentAlert`). This does not include attributes inherited from `DOMException`.

§ 11.1.3 `RTCErrorInit` Dictionary

WebIDL

```
dictionary RTCErrorInit {
  required RTCErrorDetailType errorDetail;
  long sdpLineNumber;
  long sctpCauseCode;
  unsigned long receivedAlert;
  unsigned long sentAlert;
};
```

The **`errorDetail`**, **`sdpLineNumber`**, **`sctpCauseCode`**, **`receivedAlert`** and **`sentAlert`** members of `RTCErrorInit` have the same definitions as the attributes of the same name of `RTCError`.

§ 11.2 `RTCErrorDetailType` Enum

WebIDL

```
enum RTCErrorDetailType {
  "data-channel-failure",
  "dtls-failure",
  "fingerprint-failure",
  "sctp-failure",
```

```
  "sdp-syntax-error",
  "hardware-encoder-not-available",
  "hardware-encoder-error"
};
```

RTCErrorDetailType Enumeration description

Enum value	Description
`data-channel-failure`	The data channel has failed.
`dtls-failure`	The DTLS negotiation has failed or the connection has been terminated with a fatal error. The `message` contains information relating to the nature of error. If a fatal DTLS alert was received, the `receivedAlert` attribute is set to the value of the DTLS alert received. If a fatal DTLS alert was sent, the `sentAlert` attribute is set to the value of the DTLS alert sent.
`fingerprint-failure`	The `RTCDtlsTransport`'s remote certificate did not match any of the fingerprints provided in the SDP. If the remote peer cannot match the local certificate against the provided fingerprints, this error is not generated. Instead a "bad_certificate" (42) DTLS alert might be received from the remote peer, resulting in a "`dtls-failure`".
`sctp-failure`	The SCTP negotiation has failed or the connection has been terminated with a fatal error. The `sctpCauseCode` attribute is set to the SCTP cause code.
`sdp-syntax-error`	The SDP syntax is not valid. The `sdpLineNumber` attribute is set to the line number in the SDP where the syntax error was detected.
`hardware-encoder-not-available`	The hardware encoder resources required for the requested operation are not available.
`hardware-encoder-error`	The hardware encoder does not support the provided parameters.

§ 11.3 `RTCErrorEvent` Interface

The `RTCErrorEvent` interface is defined for cases when an `RTCError` is raised as an event:

WebIDL

```
[Exposed=Window]
interface RTCErrorEvent : Event {
  constructor(DOMString type, RTCErrorEventInit eventInitDict);
  [SameObject] readonly attribute RTCError error;
};
```

§ 11.3.1 Constructors

`constructor()`

Constructs a new `RTCErrorEvent`.

§ 11.3.2 Attributes

`error` **of type `RTCError`, readonly**

The `RTCError` describing the error that triggered the event.

§ 11.4 `RTCErrorEventInit` Dictionary

WebIDL

```
        dictionary RTCErrorEventInit : EventInit {
  required RTCError error;
};
```

§ 11.4.1 Dictionary RTCErrorEventInit Members

***error* of type RTCError**
The RTCError describing the error associated with the event (if any).

§ 12. Event summary

This section is non-normative.

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).
message	MessageEvent [html]	A message was successfully received.
bufferedamountlow	Event	The RTCDataChannel object's bufferedAmount decreases from above its bufferedAmountLowThreshold to less than or equal to its bufferedAmountLowThreshold.
error	RTCErrorEvent	An error occurred on the data channel.
closing	Event	The RTCDataChannel object transitions to the "closing" state
close	Event	The RTCDataChannel object's underlying data transport has been closed.

The following events fire on RTCPeerConnection objects:

Event name	Interface	Fired when...
track	RTCTrackEvent	New incoming media has been negotiated for a specific RTCRtpReceiver, and that receiver's track has been added to any associated remote MediaStreams.
negotiationneeded	Event	The browser wishes to inform the application that session negotiation needs to be done (i.e. a createOffer call followed by setLocalDescription).
signalingstatechange	Event	The connection's [[SignalingState]] has changed. This state change is the result of either setLocalDescription or setRemoteDescription being invoked.
iceconnectionstatechange	Event	The RTCPeerConnection's [[IceConnectionState]] has changed.
icegatheringstatechange	Event	The RTCPeerConnection's [[IceGatheringState]] has changed.
icecandidate	RTCPeerConnectionIceEvent	A new RTCIceCandidate is made available to the script.
connectionstatechange	Event	The RTCPeerConnection.connectionState

Event name	Interface	Fired when...
		has changed.
icecandidateerror	RTCPeerConnectionIceErrorEvent	A failure occured when gathering ICE candidates.
datachannel	RTCDataChannelEvent	A new RTCDataChannel is dispatched to the script in response to the other peer creating a channel.

The following events fire on RTCDTMFSender objects:

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

The following events fire on RTCIceTransport objects:

Event name	Interface	Fired when...
statechange	Event	The RTCIceTransport state changes.
gatheringstatechange	Event	The RTCIceTransport gathering state changes.
selectedcandidatepairchange	Event	The RTCIceTransport's selected candidate pair changes.

The following events fire on RTCDtlsTransport objects:

Event name	Interface	Fired when...
statechange	Event	The RTCDtlsTransport state changes.
error	RTCErrorEvent	An error occurred on the RTCDtlsTransport (either "dtls-failure" or "fingerprint-failure").

The following events fire on RTCSctpTransport objects:

Event name	Interface	Fired when...
statechange	Event	The RTCSctpTransport state changes.

§ 13. Privacy and Security Considerations

This section is non-normative.

This section is non-normative; it specifies no new behaviour, but instead summarizes information already present in other parts of the specification. The overall security considerations of the general set of APIs and protocols used in WebRTC are described in [RFC8827].

§ 13.1 Impact on same origin policy

This document extends the Web platform with the ability to set up real-time, direct communication between browsers and other devices, including other browsers.

This means that data and media can be shared between applications running in different browsers, or between an application running in the same browser and something that is not a browser, something that is an extension to the usual barriers in the Web model against sending data between entities with different origins.

The WebRTC specification provides no user prompts or chrome indicators for communication; it assumes that once the Web page has been allowed to access media, it is free to share that media with other entities as it chooses. Peer-to-peer exchanges of data view WebRTC datachannels can thus occur without any user explicit consent or involvement, similarly as a server-mediated exchange (e.g. via Web Sockets) could occur without user involvement.

§ 13.2 Revealing IP addresses

Even without WebRTC, the Web server providing a Web application will know the public IP address to which the application is delivered. Setting up communications exposes additional information about the browser's network context to the web application, and may include the set of (possibly private) IP addresses available to the browser for WebRTC use. Some of this information has to be passed to the corresponding party to enable the establishment of a communication session.

Revealing IP addresses can leak location and means of connection; this can be sensitive. Depending on the network environment, it can also increase the fingerprinting surface and create persistent cross-origin state that cannot easily be cleared by the user.

A connection will always reveal the IP addresses proposed for communication to the corresponding party. The application can limit this exposure by choosing not to use certain addresses using the settings exposed by the `RTCIceTransportPolicy` dictionary, and by using relays (for instance TURN servers) rather than direct connections between participants. One will normally assume that the IP address of TURN servers is not sensitive information. These choices can for instance be made by the application based on whether the user has indicated consent to start a media connection with the other party.

Mitigating the exposure of IP addresses to the application itself requires limiting the IP addresses that can be used, which will impact the ability to communicate on the most direct path between endpoints. Browsers are encouraged to provide appropriate controls for deciding which IP addresses are made available to applications, based on the security posture desired by the user. The choice of which addresses to expose is controlled by local policy (see [RFC8828] for details).

§ 13.3 Impact on local network

Since the browser is an active platform executing in a trusted network environment (inside the firewall), it is important to limit the damage that the browser can do to other elements on the local network, and it is important to protect data from interception, manipulation and modification by untrusted participants.

Mitigations include:

- A user agent will always request permission from the correspondent user agent to communicate using ICE. This ensures that the user agent can only send to partners who you have shared credentials with.
- A user agent will always request ongoing permission to continue sending using ICE continued consent. This enables a receiver to withdraw consent to receive.
- A user agent will always encrypt data, with strong per-session keying (DTLS-SRTP).
- A user agent will always use congestion control. This ensures that WebRTC cannot be used to flood the network.

These measures are specified in the relevant IETF documents.

§ 13.4 Confidentiality of Communications

The fact that communication is taking place cannot be hidden from adversaries that can observe the network, so this has to be regarded as public information.

Communication certificates may be opaquely shared using `postMessage(`*`message, options`*`)` in anticipation of future needs. User agents are strongly encouraged to isolate the private keying material these objects hold a handle to, from the processes that have access to the `RTCCertificate` objects, to reduce memory attack surface.

§ 13.5 Persistent information exposed by WebRTC

As described above, the list of IP addresses exposed by the WebRTC API can be used as a persistent cross-origin state.

Beyond IP addresses, the WebRTC API exposes information about the underlying media system via the `RTCRtpSender.getCapabilities` and `RTCRtpReceiver.getCapabilities` methods, including detailed and ordered information about the codecs that the system is able to produce and consume. A subset of that information is likely to be represented in the SDP session descriptions generated, exposed and transmitted during session negotiation. That information is in most cases persistent across time and origins, and increases the fingerprint surface of a given device.

When establishing DTLS connections, the WebRTC API can generate certificates that can be persisted by the application (e.g. in IndexedDB). These certificates are not shared across origins, and get cleared when persistent storage is cleared for the origin.

§ 13.6 Setting SDP from remote endpoints

`setRemoteDescription` guards against malformed and invalid SDP by throwing exceptions, but makes no attempt to guard against SDP that might be unexpected by the application. Setting the remote description can cause significant resources to be allocated (including image buffers and network ports), media to start flowing (which may have privacy and bandwidth implications) among other things. An application that does not guard against malicious SDP could be at risk of resource deprivation, unintentionally allowing incoming media or at risk of not having certain events fire like `ontrack` if the other endpoint does not negotiate sending. Applications need to be on guard against malevolent SDP.

§ 14. Accessibility Considerations

This section is non-normative.

The WebRTC 1.0 specification exposes an API to control protocols (defined within the IETF) necessary to establish real-time audio, video and data exchange.

The Telecommunications Device for the Deaf (TDD/TTY) enables individuals who are hearing or speech impaired (among others) to communicate over telephone lines. Real-Time Text, defined in [RFC4103], utilizes T.140 encapsulated in RTP to enable the transition from TDD/TTY devices to IP-based communications, including emergency communication with Public Safety Access Points (PSAP).

Since Real-Time Text requires the ability to send and receive data in near real time, it can be best supported via the WebRTC 1.0 data channel API. As defined by the IETF, the data channel protocol utilizes the SCTP/DTLS/UDP protocol stack, which supports both reliable and unreliable data channels. The IETF chose to standardize SCTP/DTLS/UDP over proposals for an RTP data channel which relied on SRTP key management and were focused on unreliable communications.

Since the IETF chose a different approach than the RTP data channel as part of the WebRTC suite of protocols, as of the time of this publication there is no standardized way for the WebRTC APIs to directly support Real-Time Text as defined at IETF and implemented in U.S. (FCC) regulations. The WebRTC working Group will evaluate whether the developing IETF protocols in this space warrant direct exposure in the browser APIs and is looking for input from the relevant user communities on this potential gap.

Within the IETF MMUSIC Working Group, work is ongoing to enable Real-time text to be sent over the WebRTC data channel, allowing gateways to be deployed to translate between the SCTP data channel protocol and RFC 4103 Real-Time Text. This work, once completed, is expected to enable a unified and interoperable approach for integrating real-time text in WebRTC user-agents (including browsers) - through a gateway or otherwise.

At the time of this publication, gateways that enable effective RTT support in WebRTC clients can be developed e.g. through a custom WebRTC data channel. This is deemed sufficient until such time as future standardized gateways are enabled via IETF protocols such as the SCTP data channel protocol and RFC 4103 Real-Time Text. This will need to be defined at IETF in conjunction with related work at W3C groups to effectively and consistently standardise RTT support internationally.

§ A. Proposed Amendments

Since its publication as a W3C Recommendation in January 2021, the following proposed amendments have been integrated in this document.

- Proposed Correction 1:
  - Set default values of the `RTCConfiguration` dictionary, aligning it with current implementations - section 4.2.1 RTCConfiguration Dictionary (PR #2691) - Changes to Web Platform Tests: #43166
  - Set default values of the `RTCConfiguration` dictionary, aligning it with current implementations - section 4.4.1.6 Set the configuration (PR #2691) - Changes to Web Platform Tests: #43166
- Proposed Correction 2:
  - Allow an implementation-defined limited to the number of configured ICE Servers - section 4.2.1 RTCConfiguration Dictionary (PR #2679) (not testable)
- Proposed Correction 3:
  - Update `RTCIceGatheringState` to clarify the relevant transport it represents - section 4.3.2 RTCIceGatheringState Enum (PR #2680) (no change needed in tests)
  - Update `RTCPeerConnectionState` to clarify the relevant transport it represents - section 4.3.3 RTCPeerConnectionState Enum (PR #2680) (no change needed in tests)
  - Update `RTCIceConnectionState` to clarify the relevant transport it represents - section 4.3.4 RTCIceConnectionState Enum (PR #2680) (no change needed in tests)
- Proposed Correction 4:
  - Ensure the `connecting` state happens whenever a ICE or DTLS transport is new - section 4.3.3 RTCPeerConnectionState Enum (PR #2687) - Changes to Web Platform Tests: #43171
- Proposed Correction 6:
  - Validate ICE transport settings upfront when setting a configuration - section 4.4.1.6 Set the configuration (PR #2689) - Changes to Web Platform Tests: #43167
- Proposed Correction 7:
  - Replace `DOMTimeStamp` in the definition of the `RTCCertificateExpiration.expires` and of `RTCCertificate.expires`, and change its origin to certificate creation time - section 4.9.1 RTCCertificateExpiration Dictionary (PR #2686, PR #2700) (not testable)
  - Replace `DOMTimeStamp` in the definition of the `RTCCertificateExpiration.expires` and of `RTCCertificate.expires`, and change its origin to certificate creation time - section 4.9.2 RTCCertificate Interface (PR #2686, PR #2700) (not testable)
- Proposed Correction 8:
  - Put ICE transport connection in `failed` state when no candidates are received - section 5.6.4 RTCIceTransportState Enum (PR #2704) (not testable)
- Proposed Correction 9:
  - No longer queue a task in the determine DTMF algorithm - section 7.3 canInsertDTMF algorithm (PR #2742) (no change needed in tests)
- Proposed Correction 10:
  - Align MTI stats with implementations - section 8.6 Mandatory To Implement Stats (PR #2744, PR #2748, PR #2832) - Changes to Web Platform Tests: #35703 #43172
- Proposed Correction 11:
  - Remove unused RTCRtpDecodingParameters dictionary - section 5.2.5 RTCRtpDecodingParameters Dictionary (PR #2753) (not testable)
- Proposed Correction 15:
  - Clarify simulcast envelope is determined by negotiation - section 5.4.1 Simulcast functionality (PR #2760)

- Proposed Correction 17:
  - Remove single-value RTCIceCredentialType enum - section 4.2.2 RTCIceCredentialType Enum (PR #2767) (not testable)
  - Remove single-value RTCIceCredentialType enum - section 4.2.3 RTCIceServer Dictionary (PR #2767) (not testable)
- Proposed Addition 19:
  - Add RTCRtpEncodingParameters.maxFramerate - section Methods (PR #2785) - Changes to Web Platform Tests: #43173
  - Add RTCRtpEncodingParameters.maxFramerate - section Methods (PR #2785) - Changes to Web Platform Tests: #43173
  - Add RTCRtpEncodingParameters.maxFramerate - section Methods (PR #2785) - Changes to Web Platform Tests: #43173
  - Add RTCRtpEncodingParameters.maxFramerate - section Methods (PR #2785) - Changes to Web Platform Tests: #43173
  - Add RTCRtpEncodingParameters.maxFramerate - section 5.2.6 RTCRtpEncodingParameters Dictionary (PR #2785) - Changes to Web Platform Tests: #43173
- Proposed Correction 20:
  - Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio - section Methods (PR #2772, PR #2799) - Changes to Web Platform Tests: #37477
  - Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio - section Methods (PR #2772, PR #2799) - Changes to Web Platform Tests: #37477
  - Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio - section Methods (PR #2772, PR #2799) - Changes to Web Platform Tests: #37477
  - Remove RTCRtpEncodingParameters.scaleResolutionDownBy for audio - section Methods (PR #2772, PR #2799) - Changes to Web Platform Tests: #37477
- Proposed Correction 21:
  - Default RTCRtpEncodingParameters.scaleResolutionDownBy to 1 for video - section Methods (PR #2772) - Changes to Web Platform Tests: #37477
  - Default RTCRtpEncodingParameters.scaleResolutionDownBy to 1 for video - section 5.2 RTCRtpSender Interface (PR #2772) - Changes to Web Platform Tests: #37477
- Proposed Correction 28:
  - Update explanation of simulcast envelope. - section 5.4.1 Simulcast functionality (PR #2814) (not testable)
- Proposed Correction 29:
  - Create RTCRtpCodec dictionary and reuse in RTCRtpCodecCapability and RTCRtpCodecParameters definitions - section 5.2.9 RTCRtpCodecParameters Dictionary (PR #2834) (not testable)
- Proposed Correction 30:
  - Make RTCRtpHeaderExtensionCapability.uri required - section 5.2.12 RTCRtpHeaderExtensionCapability Dictionary (PR #2841) (not testable)
- Proposed Correction 31:
  - Fix ambiguities in the setCodecPreferences() algorithm - section Methods (PR #2847) (no change needed in tests)
- Proposed Correction 32:
  - Reject setParameters(), replaceTrack(), & insertDTMF() after stop() - section Methods (PR #2829) (no change needed in tests)
  - Reject setParameters(), replaceTrack(), & insertDTMF() after stop() - section Methods (PR #2829) (no change needed in tests)
  - Reject setParameters(), replaceTrack(), & insertDTMF() after stop() - section canInsertDTMF algorithm (PR #2829) (no change needed in tests)
- Proposed Correction 34:
  - Make removeTrack() a no-op after transceiver.stop() - section Methods (PR #2875) (no change needed in tests)

- Proposed Correction 35:
  - Don't fire connectionstatechange on pc.close() - section Update the connection state (PR #2876) (no change needed in tests)
- Proposed Addition 36:
  - Add empty setParameterOptions as second argument to setParameters for extensibility - section RTCSetParameterOptions Dictionary (PR #2885) (not testable)
- Proposed Correction 39:
  - Fix binaryType setter requirements - section Attributes (PR #2909) - Changes to Web Platform Tests: #41663
- Proposed Correction 40:
  - Change the default value of binaryType - section Attributes (PR #2913) - Changes to Web Platform Tests: #43601
- Proposed Correction 43:
  - Allow encoder resolution alignment in scaleResolutionDownBy. - section 5. RTP Media API (PR #2808) (not testable)
- Proposed Addition 44:
  - Add control for the receiver's jitter buffer - section RTCRtpReceiver Interface (PR #2953) - Changes to Web Platform Tests: #45427
  - Add control for the receiver's jitter buffer - section 5.3 RTCRtpReceiver Interface (PR #2953) - Changes to Web Platform Tests: #45427
  - Add control for the receiver's jitter buffer - section Attributes (PR #2953) - Changes to Web Platform Tests: #45427
- Proposed Correction 47:
  - setCodecPreferences() must use case-insensitive mimeType comparison - section Methods (PR #2975) - Changes to Web Platform Tests: #46526
- Proposed Addition 48:
  - Make RTCDataChannel transferable to DedicatedWorker - section 6.2 RTCDataChannel (PR #2988) - Changes to Web Platform Tests: #47707

§ B. Candidate Amendments

Since its publication as a W3C Recommendation in January 2021, the following candidate amendments have been integrated in this document.

- Candidate Correction 5:
  - Forbid ICE gathering and connectivity checks on administrative prohibited candidates - section Set the session description (PR #2708) - Changes to Web Platform Tests: #21025 #45339
  - Forbid ICE gathering and connectivity checks on administrative prohibited candidates - section Set the session description (PR #2708) - Changes to Web Platform Tests: #21025 #45339
- Candidate Correction 12:
  - Mark RTP Pause/Resume as not supported - section 5.4.1 Simulcast functionality (PR #2755) - Changes to Web Platform Tests: #34912
  - Remove interaction between encoding.active and simulcast ~rid - section 4.4.1.5 Set the session description (PR #2754) - Changes to Web Platform Tests: #34912
- Candidate Correction 13:
  - Rollback restores ridless encoding trounced by sRD(simulcastOffer). - section RTCRtpSender Interface (PR #2797) - Changes to Web Platform Tests: #37477
  - Rollback restores ridless encoding trounced by sRD(simulcastOffer). - section Set the session description (PR #2797) - Changes to Web Platform Tests: #37477
  - Rollback restores ridless encoding trounced by sRD(simulcastOffer). - section Set the session description (PR #2797) - Changes to Web Platform Tests: #37477
- Candidate Correction 14:

  - Make RTCTransceiver.direction reflects local preference in offers and answers - section 4.4.1.5 Set the session description (PR #2759)
- Candidate Addition 16:
  - Add RTCIceCandidate.relayProtocol - section 4.8.1 RTCIceCandidate Interface (PR #2763) - Changes to Web Platform Tests: #36157
  - Add RTCIceCandidate.relayProtocol - section RTCIceServerTransportProtocol Enum (PR #2763) - Changes to Web Platform Tests: #36157
  - Add RTCIceCandidate.relayProtocol - section Attributes (PR #2763) - Changes to Web Platform Tests: #36157
- Candidate Correction 18:
  - TypeError unless all or none of encodings have rids and on duplicate rids - section Methods (PR #2774, PR #2775) - Changes to Web Platform Tests: #37477
- Candidate Correction 22:
  - Allow remote offer rid pruning of encodings through the client answer. - section 4.4.1.5 Set the session description (PR #2758)
- Candidate Addition 23:
  - Add RTCIceCandidate.url - section 4.8.1 RTCIceCandidate Interface (PR #2773) - Changes to Web Platform Tests: #36572
  - Mark RTCPeerConnectionIceEvent.url as deprecated - section Attributes (PR #2773) (not testable)
  - Add RTCIceCandidate.url - section Attributes (PR #2773) - Changes to Web Platform Tests: #36572
- Candidate Correction 24:
  - Queue two tasks upon finishing ICE gathering, and fire gatheringstatechange & icegatheringstatechange in same task - section 5.6 RTCIceTransport Interface (PR #2894) - Changes to Web Platform Tests: #44687
- Candidate Correction 25:
  - Remove duplicate rids in proposedSendEncodings. - section 4.4.1.5 Set the session description (PR #2800) - Changes to Web Platform Tests: #37477
- Candidate Correction 26:
  - Prune createAnswer()'s encodings and SendEncodings in sLD(answer). - section 4.4.1.5 Set the session description (PR #2801)
  - Prune createAnswer()'s encodings and SendEncodings in sLD(answer). - section Methods (PR #2801)
- Candidate Correction 27:
  - Ignore comma-separated rid alternatives. - section 4.4.1.5 Set the session description (PR #2813) - Changes to Web Platform Tests: #36155 #37477
  - Ignore comma-separated rid alternatives. - section Methods (PR #2813) - Changes to Web Platform Tests: #36155 #37477
- Candidate Correction 33:
  - Use the url spec to parse ice server urls - section 4.4.1.6 Set the configuration (PR #2853)
  - Determine if DTMF can be sent inside queued playout task - section Methods (PR #2861)
- Candidate Correction 37:
  - Don't fail sRD(offer) over rid mismatch, just answer with unicast. - section 4.4.1.5 Set the session description (PR #2794)
  - Don't fail sRD(offer) over rid mismatch, just answer with unicast. - section 4.4.1.5 Set the session description (PR #2794)
- Candidate Correction 38:
  - Prevent GC of non-closed RTCDataChannels - section 6.2.4 Closing procedure (PR #2902) - Changes to Web Platform Tests: #43369
- Candidate Correction 42:
  - setCodecPreferences only takes into account receive codecs - section Methods (PR #2926) - Changes to Web Platform Tests: #44318
- Candidate Addition 45:

    - Convert RTCIceCandidatePair dictionary to an interface - section 5.6.2 RTCIceCandidatePair Dictionary (PR #2961) - Changes to Web Platform Tests: #46647 #46655
- Candidate Correction 46:
    - Replace RFC8829 reference with RFC9429 - section B.1 Normative references (PR #2966) (no change needed in tests)
- Candidate Addition 48:
    - Make RTCDataChannel transferable to DedicatedWorker - section Creating a data channel (PR #2988) - Changes to Web Platform Tests: #47707
    - Make RTCDataChannel transferable to DedicatedWorker - section Creating a data channel (PR #2988) - Changes to Web Platform Tests: #47707
    - Make RTCDataChannel transferable to DedicatedWorker - section Methods (PR #2988) - Changes to Web Platform Tests: #47707
    - Make RTCDataChannel transferable to DedicatedWorker - section Transfering data channel (PR #2988) - Changes to Web Platform Tests: #47707
- Candidate Addition 49:
    - Add codec to RTCRtpEncodingParameters - section Methods (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Methods (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Methods (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Methods (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Methods (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Methods (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Methods (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Dictionary `RTCRtpEncodingParameters` Members (PR #2985) - Changes to Web Platform Tests: #47663
    - Add codec to RTCRtpEncodingParameters - section Set the session description (PR #2985) - Changes to Web Platform Tests: #47663
- Candidate Correction 50:
    - Use Performance.timeOrigin + Performance.now() for stats timestamps - section Dictionary RTCStats Members (PR #3005) - Changes to Web Platform Tests: #48361

§ C. Acknowledgements

The editors wish to thank the Working Group chairs and Team Contact, Harald Alvestrand, Stefan Håkansson, Erik Lagerway and Dominique Hazaël-Massieux, for their support. Substantial text in this specification was provided by many people including Martin Thomson, Harald Alvestrand, Justin Uberti, Eric Rescorla, Peter Thatcher, Jan-Ivar Bruaroey and Peter Saint-Andre. Dan Burnett would like to acknowledge the significant support received from Voxeo and Aspect during the development of this specification.

The `RTCRtpSender` and `RTCRtpReceiver` objects were initially described in the W3C ORTC CG, and have been adapted for use in this specification.

§ D. References

CANDIDATE CORRECTION 46: Replace RFC8829 reference with RFC9429 (PR #2966)

Show Current and Future Show Current Show Future

~~B.1 Normative references~~

§ D.1 Normative references

[DOM]
DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMASCRIPT-6.0]
ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. Allen Wirfs-Brock. Ecma International. June 2015. Standard. URL: http://www.ecma-international.org/ecma-262/6.0/index.html

[Fetch]
Fetch Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://fetch.spec.whatwg.org/

[FILEAPI]
File API. Marijn Kruisselbrink~~; Arun Ranganathan~~. W3C. ~~11 September 2019~~24 May 2024. W3C Working Draft. URL: https://www.w3.org/TR/FileAPI/

[FIPS-180-4]
FIPS PUB ~~180-4~~ 180-4: Secure Hash ~~Standard~~Standard (SHS). U.S. Department of Commerce/National Institute of Standards and Technology. August 2015. National Standard. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[GETUSERMEDIA]
Media Capture and Streams. Cullen Jennings; Bernard Aboba; Jan-Ivar Bruaroey; Henrik Boström; youenn fablet~~; Daniel Burnett; Adam Bergkvist; Anant Narayanan~~. W3C. ~~21 January 2021~~3 October 2024. W3C Candidate Recommendation. URL: https://www.w3.org/TR/mediacapture-streams/

[hr-time]
High Resolution ~~Time Level 2~~Time. ~~Ilya Grigorik~~Yoav Weiss. W3C. ~~21 November 2019~~19 July 2023. W3C ~~Recommendation~~Working Draft. URL: https://www.w3.~~org/TR/hr-time-2/~~org/TR/hr-time-3/

[HTML]
HTML Standard. Anne van Kesteren; Domenic Denicola; Dominic Farolino; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[IANA-HASH-FUNCTION]
Hash Function Textual Names. IANA. URL: https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xml

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

[INFRA]
Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: ~~https://tools.ietf.org/html/rfc2119~~https://www.rfc-editor.org/rfc/rfc2119

[RFC3550]
RTP: A Transport Protocol for Real-Time Applications. H. Schulzrinne; S. Casner; R. Frederick; V. Jacobson. IETF. July 2003. Internet Standard. URL: ~~https://tools.ietf.org/html/rfc3550~~https://www.rfc-editor.org/rfc/rfc3550

[RFC3890]
A Transport Independent Bandwidth Modifier for the Session Description Protocol (SDP). M. Westerlund. IETF. September 2004. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc3890~~https://www.rfc-editor.org/rfc/rfc3890

~~**[RFC3986]**~~
~~*Uniform Resource Identifier (URI): Generic Syntax*. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986~~

[RFC4566]

SDP: Session Description Protocol. M. Handley; V. Jacobson; C. Perkins. IETF. July 2006. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc4566~~https://www.rfc-editor.org/rfc/rfc4566

[RFC4572]
Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP). J. Lennox. IETF. July 2006. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc4572~~https://www.rfc-editor.org/rfc/rfc4572

[RFC5245]
Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. J. Rosenberg. IETF. April 2010. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc5245~~https://www.rfc-editor.org/rfc/rfc5245

[RFC5246]
The Transport Layer Security (TLS) Protocol Version 1.2. T. Dierks; E. Rescorla. IETF. August 2008. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc5246~~https://www.rfc-editor.org/rfc/rfc5246

[RFC5285]
A General Mechanism for RTP Header Extensions. D. Singer; H. Desineni. IETF. July 2008. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc5285~~https://www.rfc-editor.org/rfc/rfc5285

[RFC5389]
Session Traversal Utilities for NAT (STUN). J. Rosenberg; R. Mahy; P. Matthews; D. Wing. IETF. October 2008. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc5389~~https://www.rfc-editor.org/rfc/rfc5389

[RFC5506]
Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences. I. Johansson; M. Westerlund. IETF. April 2009. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc5506~~https://www.rfc-editor.org/rfc/rfc5506

[RFC5888]
The Session Description Protocol (SDP) Grouping Framework. G. Camarillo; H. Schulzrinne. IETF. June 2010. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc5888~~https://www.rfc-editor.org/rfc/rfc5888

[RFC6464]
A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication. J. Lennox, Ed.; E. Ivov; E. Marocco. IETF. December 2011. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc6464~~https://www.rfc-editor.org/rfc/rfc6464

[RFC6465]
A Real-time Transport Protocol (RTP) Header Extension for Mixer-to-Client Audio Level Indication. E. Ivov, Ed.; E. Marocco, Ed.; J. Lennox. IETF. December 2011. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc6465~~https://www.rfc-editor.org/rfc/rfc6465

[RFC6544]
TCP Candidates with Interactive Connectivity Establishment (ICE). J. Rosenberg; A. Keranen; B. B. Lowekamp; A. B. Roach. IETF. March 2012. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc6544~~https://www.rfc-editor.org/rfc/rfc6544

[RFC7064]
URI Scheme for the Session Traversal Utilities for NAT (STUN) Protocol. S. Nandakumar; G. Salgueiro; P. Jones; M. Petit-Huguenin. IETF. November 2013. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc7064~~https://www.rfc-editor.org/rfc/rfc7064

[RFC7065]
Traversal Using Relays around NAT (TURN) Uniform Resource Identifiers. M. Petit-Huguenin; S. Nandakumar; G. Salgueiro; P. Jones. IETF. November 2013. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc7065~~https://www.rfc-editor.org/rfc/rfc7065

[RFC7656]
A Taxonomy of Semantics and Mechanisms for Real-Time Transport Protocol (RTP) Sources. J. Lennox; K. Gross; S. Nandakumar; G. Salgueiro; B. Burman, Ed.. IETF. November 2015. Informational. URL: ~~https://tools.ietf.org/html/rfc7656~~https://www.rfc-editor.org/rfc/rfc7656

[RFC7675]
Session Traversal Utilities for NAT (STUN) Usage for Consent Freshness. M. Perumal; D. Wing; R. Ravindranath; T. Reddy; M. Thomson. IETF. October 2015. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc7675~~https://www.rfc-editor.org/rfc/rfc7675

[RFC7874]
WebRTC Audio Codec and Processing Requirements. JM. Valin; C. Bran. IETF. May 2016. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc7874~~https://www.rfc-editor.org/rfc/rfc7874

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: ~~https://tools.ietf.org/html/rfc8174~~https://www.rfc-editor.org/rfc/rfc8174

[RFC8261]
Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets. M. Tuexen; R. Stewart; R. Jesup; S. Loreto. IETF. November 2017. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8261~~https://www.rfc-editor.org/rfc/rfc8261

[RFC8445]
Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. A. Keranen; C. Holmberg; J. Rosenberg. IETF. July 2018. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8445~~https://www.rfc-editor.org/rfc/rfc8445

[RFC8656]
Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). T. Reddy, Ed.; A. Johnston, Ed.; P. Matthews; J. Rosenberg. IETF. February 2020. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8656

[RFC8826]
Security Considerations for WebRTC~~*for WebRTC*~~. E. Rescorla. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8826~~https://www.rfc-editor.org/rfc/rfc8826

~~**[RFC8829]**~~
~~*JavaScript Session Establishment Protocol (JSEP)*. J. Uberti; C. Jennings; E. Rescorla, Ed.. IETF. January 2021. Proposed Standard. URL: https://tools.ietf.org/html/rfc8829~~

[RFC8831]
WebRTC Data Channels. R. Jesup; S. Loreto; M. Tüxen. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8831~~https://www.rfc-editor.org/rfc/rfc8831

[RFC8832]
WebRTC Data Channel Establishment Protocol. R. Jesup; S. Loreto; M. Tüxen. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8832~~https://www.rfc-editor.org/rfc/rfc8832

[RFC8834]
Media Transport and Use of RTP in WebRTC. C. Perkins; M. Westerlund; J. Ott. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8834~~https://www.rfc-editor.org/rfc/rfc8834

[RFC8835]
Transports for WebRTC. H. Alvestrand. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8835~~https://www.rfc-editor.org/rfc/rfc8835

[RFC8838]
Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol. E. Ivov; J. Uberti; P. Saint-Andre. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8838~~https://www.rfc-editor.org/rfc/rfc8838

[RFC8841]
Session Description Protocol (SDP) Offer/Answer Procedures for Stream Control Transmission Protocol (SCTP) over Datagram Transport Layer Security (DTLS) Transport. C. Holmberg; R. Shpount; S. Loreto; G. Camarillo. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8841~~https://www.rfc-editor.org/rfc/rfc8841

[RFC8843]
Negotiating Media Multiplexing Using the Session Description Protocol (SDP). C. Holmberg; H. Alvestrand; C. Jennings. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8843~~https://www.rfc-editor.org/rfc/rfc8843

[RFC8851]
RTP Payload Format Restrictions. A.B. Roach, Ed.. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8851~~https://www.rfc-editor.org/rfc/rfc8851

[RFC8853]
Using Simulcast in Session Description Protocol (SDP) and RTP Sessions. B. Burman; M. Westerlund; S. Nandakumar; M. Zanaty. IETF. January 2021. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc8853~~https://www.rfc-editor.org/rfc/rfc8853

[RFC8863]
Interactive Connectivity Establishment Patiently Awaiting Connectivity (ICE PAC). C. Holmberg; J. Uberti. IETF. January 2021. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8863

[RFC9429]
JavaScript Session Establishment Protocol (JSEP). J. Uberti; C. Jennings; E. Rescorla, Ed.. IETF. April 2024. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9429

[SDP]

An Offer/Answer Model with Session Description Protocol (SDP). J. Rosenberg; H. Schulzrinne. IETF. June 2002. Proposed Standard. URL: ~~https://tools.ietf.org/html/rfc3264~~https://www.rfc-editor.org/rfc/rfc3264

[STUN-PARAMETERS]
STUN Error Codes. IETF. IANA. April 2011. IANA Parameter Assignment. URL: https://www.iana.org/assignments/stun-parameters/stun-parameters.xhtml#stun-parameters-6

[url]
URL Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://url.spec.whatwg.org/

[WebCryptoAPI]
Web Cryptography API. Mark Watson. W3C. 26 January 2017. W3C Recommendation. URL: https://www.w3.org/TR/WebCryptoAPI/

[WEBIDL]
~~*Web IDL*. Boris Zbarsky. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/~~

Web IDL Standard. Edgar Chen; Timothy Gu. WHATWG. Living Standard. URL: https://webidl.spec.whatwg.org/

[WEBRTC-STATS]
Identifiers for WebRTC's Statistics API. Harald Alvestrand; Varun Singh; Henrik Boström. W3C. ~~20 January 2021~~18 July 2024. W3C Candidate Recommendation. URL: https://www.w3.org/TR/webrtc-stats/

[X509V3]
ITU-T Recommendation X.509 version 3 (1997). "Information Technology - Open Systems Interconnection - The Directory Authentication Framework" ISO/IEC 9594-8:1997.. ITU.

[X690]
Recommendation X.690 — Information Technology — ASN.1 Encoding Rules — Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). ITU. URL: https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

§ D.2 Informative references

[API-DESIGN-PRINCIPLES]
Web Platform Design Principles. Lea Verou. W3C. 18 July 2024. W3C Working Group Note. URL: https://www.w3.org/TR/design-principles/

[INDEXEDDB]
Indexed Database API. Nikunj Mehta; Jonas Sicking; Eliot Graff; Andrei Popescu; Jeremy Orlow; Joshua Bell. W3C. 8 January 2015. W3C Recommendation. URL: https://www.w3.org/TR/IndexedDB/

[RFC4103]
RTP Payload for Text Conversation. G. Hellstrom; P. Jones. IETF. June 2005. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc4103

[RFC6236]
Negotiation of Generic Image Attributes in the Session Description Protocol (SDP). I. Johansson; K. Jung. IETF. May 2011. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6236

[RFC7728]
RTP Stream Pause and Resume. B. Burman; A. Akram; R. Even; M. Westerlund. IETF. February 2016. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7728

[RFC8825]
Overview: Real-Time Protocols for Browser-Based Applications. H. Alvestrand. IETF. January 2021. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8825

[RFC8827]
WebRTC Security Architecture. E. Rescorla. IETF. January 2021. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8827

[RFC8828]
WebRTC IP Address Handling Requirements. J. Uberti; G. Shieh. IETF. January 2021. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8828

↑