WebRTC 1.0: Real-time Communication Between Browsers

W3C Candidate Recommendation 27 September 2018

This version:
: https://www.w3.org/TR/2018/CR-webrtc-20180927/

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

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

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

Implementation report:
: https://wpt.fyi/webrtc

Previous version:
: https://www.w3.org/TR/2018/CR-webrtc-20180621/

Editors:
: Adam Bergkvist, Ericsson
: Daniel C. Burnett, Invited Expert
: Cullen Jennings, Cisco
: Anant Narayanan, Mozilla (until November 2012)
: Bernard Aboba, Microsoft Corporation (until March 2017)
: Taylor Brandstetter, Google
: Jan-Ivar Bruaroey, Mozilla

Participate:
: Mailing list
: Browse open issues
: IETF RTCWEB Working Group

Abstract

This document defines a set of ECMAScript APIs in WebIDL to allow media 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 developed by the Media Capture Task Force.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. 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/.

The API is based on preliminary work done in the WHATWG.

While the specification is feature complete and is expected to be stable, there are also a number of known substantive issues on the specification that will be addressed during the Candidate Recommendation period based on implementation experience feedback.

It might also evolve based on feedback gathered as its associated test suite evolves. This test suite will be used to build an implementation report of the API.

The Identity Framework for WebRTC and its associated feature of isolated media streams, previously published as part of this specification, have been moved to a separate Identity for WebRTC 1.0 specification.

To go into Proposed Recommendation status, the group expects to demonstrate implementation of each feature in at least two deployed browsers, and at least one implementation of each optional feature. Mandatory feature with only one implementation may be marked as optional in a revised Candidate Recommendation where applicable.

The following features are marked as at risk:

- The value `negotiate` of RTCRtcpMuxPolicy
- The `encodings` attribute of `RTCRtpReceiveParameters`

This document was published by the Web Real-Time Communications Working Group as a Candidate Recommendation. This document is intended to become a W3C Recommendation. Comments regarding this document are welcome. Please send them to public-webrtc@w3.org (subscribe, archives). W3C publishes a Candidate Recommendation to indicate that the document is believed to be stable and to encourage implementation by the developer community. This Candidate Recommendation is expected to advance to Proposed Recommendation no earlier than 31 December 2018.

Please see the Working Group's implementation report.

Publication as a Candidate Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

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 1 February 2018 W3C Process Document.

Table of Contents

1. Introduction

2. Conformance

3. Terminology

4. Peer-to-peer connections

4.1 Introduction
4.2 Configuration
4.2.1 `RTCConfiguration` Dictionary
4.2.2 `RTCIceCredentialType` Enum
4.2.3 `RTCOAuthCredential` Dictionary
4.2.4 `RTCIceServer` Dictionary
4.2.5 `RTCIceTransportPolicy` Enum
4.2.6 `RTCBundlePolicy` Enum
4.2.7 `RTCRtcpMuxPolicy` Enum
4.2.8 Offer/Answer Options

4.3 State Definitions
4.3.1 RTCSignalingState Enum
4.3.2 RTCIceGatheringState Enum
4.3.3 RTCPeerConnectionState Enum
4.3.4 RTCIceConnectionState Enum
4.4 RTCPeerConnection Interface
4.4.1 Operation
4.4.1.1 Constructor
4.4.1.2 Enqueue an operation
4.4.1.3 Update the connection state
4.4.1.4 Update the ICE gathering state
4.4.1.5 Update the ICE connection state
4.4.1.6 Set the RTCSessionDescription
4.4.1.7 Set the configuration
4.4.2 Interface Definition
4.4.3 Legacy Interface Extensions
4.4.3.1 Method extensions
4.4.3.2 Legacy configuration extensions
4.4.4 Garbage collection
4.5 Error Handling
4.5.1 General Principles
4.6 Session Description Model
4.6.1 RTCSdpType
4.6.2 RTCSessionDescription Class
4.7 Session Negotiation Model
4.7.1 Setting Negotiation-Needed
4.7.2 Clearing Negotiation-Needed
4.7.3 Updating the Negotiation-Needed flag
4.8 Interfaces for Connectivity Establishment
4.8.1 RTCIceCandidate Interface
4.8.1.1 candidate-attribute Grammar
4.8.1.2 RTCIceProtocol Enum
4.8.1.3 RTCIceTcpCandidateType Enum
4.8.1.4 RTCIceCandidateType Enum
4.8.2 RTCPeerConnectionIceEvent
4.8.3 RTCPeerConnectionIceErrorEvent
4.9 Priority and QoS Model
4.9.1 RTCPriorityType Enum
4.10 Certificate Management
4.10.1 RTCCertificateExpiration Dictionary

4.10.2 RTCCertificate Interface

5. **RTP Media API**
5.1 RTCPeerConnection Interface Extensions
5.1.1 Processing Remote MediaStreamTracks
5.2 RTCRtpSender Interface
5.2.1 RTCRtpParameters Dictionary
5.2.2 RTCRtpSendParameters Dictionary
5.2.3 RTCRtpReceiveParameters Dictionary
5.2.4 RTCRtpCodingParameters Dictionary
5.2.5 RTCRtpDecodingParameters Dictionary
5.2.6 RTCRtpEncodingParameters Dictionary
5.2.7 RTCDtxStatus Enum
5.2.8 RTCDegradationPreference Enum
5.2.9 RTCRtcpParameters Dictionary
5.2.10 RTCRtpHeaderExtensionParameters Dictionary
5.2.11 RTCRtpCodecParameters Dictionary
5.2.12 RTCRtpCapabilities Dictionary
5.2.13 RTCRtpCodecCapability Dictionary
5.2.14 RTCRtpHeaderExtensionCapability Dictionary
5.3 RTCRtpReceiver Interface
5.4 RTCRtpTransceiver Interface
5.4.1 Simulcast functionality
5.4.1.1 Encoding Parameter Examples
5.4.2 "Hold" functionality
5.5 RTCDtlsTransport Interface
5.5.1 RTCDtlsFingerprint Dictionary
5.6 RTCIceTransport Interface
5.6.1 RTCIceParameters Dictionary
5.6.2 RTCIceCandidatePair Dictionary
5.6.3 RTCIceGathererState Enum
5.6.4 RTCIceTransportState Enum
5.6.5 RTCIceRole Enum
5.6.6 RTCIceComponent Enum
5.7 RTCTrackEvent

6. **Peer-to-peer Data API**
6.1 RTCPeerConnection Interface Extensions
6.1.1 RTCSctpTransport Interface
6.1.1.1 Create an instance

6.1.1.2 Update max message size

6.1.1.3 Connected procedure

6.1.2 `RTCSctpTransportState` Enum

6.2 `RTCDataChannel`

6.3 `RTCDataChannelEvent`

6.4 Garbage Collection

7. Peer-to-peer DTMF

7.1 RTCRtpSender Interface Extensions

7.2 `RTCDTMFSender`

7.3 canInsertDTMF algorithm

7.4 `RTCDTMFToneChangeEvent`

8. Statistics Model

8.1 Introduction

8.2 RTCPeerConnection Interface Extensions

8.3 `RTCStatsReport` Object

8.4 `RTCStats` Dictionary

8.5 `RTCStatsEvent`

8.6 The stats selection algorithm

8.7 Mandatory To Implement Stats

8.8 GetStats Example

9. Media Stream API Extensions for Network Use

9.1 Introduction

9.2 MediaStream

9.2.1 id

9.3 MediaStreamTrack

9.3.1 MediaTrackSupportedConstraints, MediaTrackCapabilities, MediaTrackConstraints and MediaTrackSettings

10. Examples and Call Flows

10.1 Simple Peer-to-peer Example

10.2 Advanced Peer-to-peer Example with Warm-up

10.3 Peer-to-peer Example with media before signaling

10.4 Simulcast Example

10.5 Peer-to-peer Data Example

10.6 Call Flow Browser to Browser

10.7 DTMF Example

11. Error Handling

11.1 ECMAScript 6 Terminology
11.2 RTCError Object
11.2.1 RTCError Constructor
11.2.1.1 `RTCErrorDetailType` Enum
11.2.1.2 RTCError (errorDetail, message)
11.2.2 Properties of the RTCError Constructor
11.2.2.1 RTCError.prototype
11.2.3 Properties of the RTCError Prototype Object
11.2.3.1 RTCError.prototype.constructor
11.2.3.2 RTCError.prototype.errorDetail
11.2.3.3 RTCError.prototype.sdpLineNumber
11.2.3.4 RTCError.prototype.httpRequestStatusCode
11.2.3.5 RTCError.prototype.sctpCauseCode
11.2.3.6 RTCError.prototype.receivedAlert
11.2.3.7 RTCError.prototype.sentAlert
11.2.3.8 RTCError.prototype.message
11.2.3.9 RTCError.prototype.name
11.2.4 Properties of RTCError Instances

12. Event summary

13. Privacy and Security Considerations
13.1 Impact on same origin policy
13.2 Revealing IP addresses
13.3 Impact on local network
13.4 Confidentiality of Communications
13.5 Persistent information exposed by WebRTC

14. Change Log

A. Acknowledgements

B. References
B.1 Normative references
B.2 Informative references

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 Media Capture Task Force. An overview of the system can be found in [RTCWEB-OVERVIEW] and [RTCWEB-SECURITY].

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*, *SHALL*, and *SHOULD* are to be interpreted as described in [RFC2119].

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-1], as this specification uses that specification and terminology.

3. Terminology

The `EventHandler` interface, representing a callback used for event handlers, and the **`ErrorEvent`** interface are defined in [HTML51].

The concepts **queue a task** and **networking task source** are defined in [HTML51].

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

The terms **event**, **event handlers** and **event handler event types** are defined in [HTML51].

`performance.timeOrigin` and `performance.now()` are defined in [HIGHRES-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 `the MediaStream section` in this document while `MediaStreamTrack` is extended in `the MediaStreamTrack section` 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 [TRICKLE-ICE] Section 2.

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

When referring to exceptions, the terms **throw** and **create** are defined in [WEBIDL-1].

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

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

The terms **bundle**, **bundle-only** and **bundle-policy** are defined in [JSEP].

The **OAuth Client** and **Authorization Server** roles are defined in [RFC6749] Section 1.1.

The terms **isolated stream**, **peeridentity**, **request an identity assertion** and **validate the identity** are defined in [WEBRTC-IDENTITY].

4. Peer-to-peer connections

4.1 Introduction

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 `XMLHttpRequest` [XMLHttpRequest] or Web Sockets [WEBSOCKETS-API].

4.2 Configuration

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.

WebIDL

```
dictionary RTCConfiguration {
    sequence<RTCIceServer>   iceServers;
    RTCIceTransportPolicy    iceTransportPolicy = "all";
    RTCBundlePolicy          bundlePolicy = "balanced";
    RTCRtcpMuxPolicy         rtcpMuxPolicy = "require";
    DOMString                peerIdentity;
    sequence<RTCCertificate> certificates;
    [EnforceRange]
    octet                    iceCandidatePoolSize = 0;
};
```

Dictionary ***`RTCConfiguration`*** *Members*

`iceServers` of type sequence<*RTCIceServer*>
: An array of objects describing servers available to be used by ICE, such as STUN and TURN servers.

`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.

`peerIdentity` of type DOMString

Sets the target peer identity for the `RTCPeerConnection`. The `RTCPeerConnection` will not establish a connection to a remote peer unless it can be successfully authenticated with the provided name.

`certificates` of type sequence<*RTCCertificate*>

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.

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 [JSEP] (section 3.5.4. and section 4.1.1.).

4.2.2 `RTCIceCredentialType` Enum

WebIDL

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

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

oauth

An OAuth 2.0 based authentication method, as described in [RFC7635].

For OAuth Authentication, the ICE Agent requires three pieces of credential information. The credential is composed of a `kid`, which the `RTCIceServer` `username` member is used for, and `macKey` and `accessToken`, which are placed in the `RTCOAuthCredential` dictionary.

> ***NOTE***
>
> ***This specification does not define how an application (acting as the OAuth Client) obtains the `accessToken`, `kid` and `macKey` from the Authorization Server, as WebRTC only handles the interaction between the ICE agent and TURN server. For example, the application may use the OAuth 2.0 Implicit Grant type, with PoP (Proof-of-Possession) Token type, as described in [RFC6749] and [OAUTH-POP-KEY-DISTRIBUTION]; an example of this is provided in [RFC7635], Appendix B.***

The application, acting as the OAuth Client, is responsible for refreshing the credential information and updating the ICE Agent with fresh new credentials before the `accessToken` expires. The OAuth Client can use the `RTCPeerConnection` `setConfiguration` method to periodically refresh the TURN credentials.

The length of the HMAC key (`RTCOAuthCredential.macKey`) *MAY* be any integer number of bytes greater than 20 (160 bits).

> ***NOTE***
>
> ***According to [RFC7635] Section 4.1, the HMAC key MUST be a symmetric key, as asymmetric keys would result in large access tokens which may not fit in a single STUN message.***

> ***NOTE***
>
> ***Currently the STUN/TURN protocols use only SHA-1 and SHA-2 family hash algorithms for Message Integrity Protection, as defined in [RFC5389] Section 15.4, and [STUN-BIS] Section 14.6.***

4.2.3 `RTCOAuthCredential` Dictionary

The `RTCOAuthCredential` dictionary is used to describe the OAuth auth credential information which is used by the STUN/TURN client (inside the ICE Agent) to authenticate against a STUN/TURN server, as described in [RFC7635]. Note that the `kid` parameter is not located in this dictionary, but in `RTCIceServer`'s `username` member.

WebIDL

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

Dictionary `RTCOAuthCredential` Members

`macKey` of type DOMString, required
: The "mac_key", as described in [RFC7635], Section 6.2, in a base64-url encoded format. It is used in STUN message integrity hash calculation (as the password is used in password based authentication). Note that the OAuth response "key" parameter is a JSON Web Key (JWK) or a JWK encrypted with a JWE format. Also note that this is the only OAuth parameter whose value is not used directly, but must be extracted from the "k" parameter value from the JWK, which contains the needed base64-encoded "mac_key".

`accessToken` of type DOMString, required
: The "access_token", as described in [RFC7635], Section 6.2, in a base64-encoded format. This is an encrypted self-contained token that is opaque to the application. Authenticated encryption is used for message encryption and integrity protection. The access token contains a non-encrypted nonce value, which is used by the Authorization Server for unique mac_key generation. The second part of the token is protected by Authenticated Encryption. It contains the mac_key, a timestamp and a lifetime. The

timestamp combined with lifetime provides expiry information; this information describes the time window during which the token credential is valid and accepted by the TURN server.

An example of an RTCOAuthCredential dictionary is:

EXAMPLE 1

```
{
  macKey: 'WmtzanB3ZW9peFhtdm42NzUzNG0=',
  accessToken:
'AAwg3kPHWPfvk9bDFL936wYvkoctMADzQ5VhNDgeMR3+ZlZ35byg972fW8QjpEl7b
x91YLBPFsIhsxloWcXPhA=='
}
```

4.2.4 `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.

WebIDL

```
dictionary RTCIceServer {
    required (DOMString or sequence<DOMString>) urls;
             DOMString                          username;
             (DOMString or RTCOAuthCredential)  credential;
             RTCIceCredentialType               credentialType =
"password";
};
```

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 specifies the username to use with that TURN server.

If this `RTCIceServer` object represents a TURN server, and `credentialType` is `"oauth"`, then this attribute specifies the Key ID (`kid`) of the shared symmetric key, which is shared between the TURN server and the Authorization Server, as described in [RFC7635]. It is an ephemeral and unique key identifier. The `kid` allows the TURN server to select the appropriate keying material for decryption of the Access-Token, so the key identified by this `kid` is used in the Authenticated Encryption of the "access_token". The `kid` value is equal with the OAuth response "kid" parameter, as defined in [RFC7515] Section 4.1.4.

credential of type (DOMString or *RTCOAuthCredential*)

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` is a DOMString, and represents a long-term authentication password, as described in [RFC5389], Section 10.2.

If `credentialType` is `"oauth"`, `credential` is an `RTCOAuthCredential`, which contains the OAuth access token and MAC key.

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 2

```
[
  {urls: 'stun:stun1.example.net'},
  {urls: ['turns:turn.example.org', 'turn:turn.example.net'],
    username: 'user',
    credential: 'myPassword',
    credentialType: 'password'},
  {urls: 'turns:turn2.example.net',
    username: '22BIjxU93h/IgwEb',
    credential: {
      macKey: 'WmtzanB3ZW9peFhtdm42NzUzNG0=',
      accessToken:
'AAwg3kPHWPfvk9bDFL936wYvkoctMADzQ5VhNDgeMR3+ZlZ35byg972fW8QjpEl7b
x91YLBPFsIhsxloWcXPhA=='
    },
    credentialType: 'oauth'}
];
```

4.2.5 `RTCIceTransportPolicy` Enum

As described in [JSEP] (section 4.1.1.), if the `iceTransportPolicy` member of the `RTCConfiguration` is specified, it defines the ICE candidate policy [JSEP] (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"
};
```

Enumeration description (non-normative)	
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`.*

4.2.6 `RTCBundlePolicy` Enum

As described in [JSEP] (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"
};
```

Enumeration description (non-normative)	
`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.

4.2.7 `RTCRtcpMuxPolicy` Enum

As described in [JSEP] (section 4.1.1.), the RtcpMuxPolicy affects what ICE candidates are gathered to support non-multiplexed RTCP.

WebIDL

```
enum RTCRtcpMuxPolicy {
    // At risk due to lack of implementers' interest.
    "negotiate",
    "require"
};
```

Enumeration description (non-normative)	
negotiate	Gather ICE candidates for both RTP and RTCP candidates. If the remote-endpoint is capable of multiplexing RTCP, multiplex RTCP on the RTP candidates. If it is not, use both the RTP and RTCP candidates separately. Note that, as stated in [JSEP] (section 4.1.1.), the user agent *MAY* not implement non-multiplexed RTCP, in which case it will reject attempts to construct an `RTCPeerConnection` with the `negotiate` policy.
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.

FEATURE AT RISK 1

Aspects of this specification supporting non-multiplexed RTP/RTCP are marked as features at risk, since there is no clear commitment from implementers. This includes:

1. The value `negotiate`, since there is no clear commitment from implementers for the behavior associated with this.
2. Support for the `rtcpTransport` attribute within the `RTCRtpSender` and `RTCRtpReceiver`.

4.2.8 Offer/Answer Options

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

WebIDL

```
dictionary RTCOfferAnswerOptions {
    boolean voiceActivityDetection = true;
};
```

Dictionary **RTCOfferAnswerOptions** *Members*

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

Many codecs and systems are capable of detecting "silence" and changing their behavior in this case by doing things such as not transmitting any media. In many cases, such as when dealing with emergency calling or sounds other than spoken voice, it is desirable to be able to turn off this behavior. This option allows the application to provide information about whether it wishes this type of processing enabled or disabled.

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, the generated description will have ICE credentials that are different from the current credentials (as visible in the `localDescription` attribute's SDP). Applying the generated description will restart ICE, as described in section 9.1.1.1 of [ICE].

When the value of this dictionary member is false, and the `localDescription` attribute has valid ICE credentials, the generated description will have the same ICE credentials as the current value from the `localDescription` 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"
};
```

Enumeration 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`.

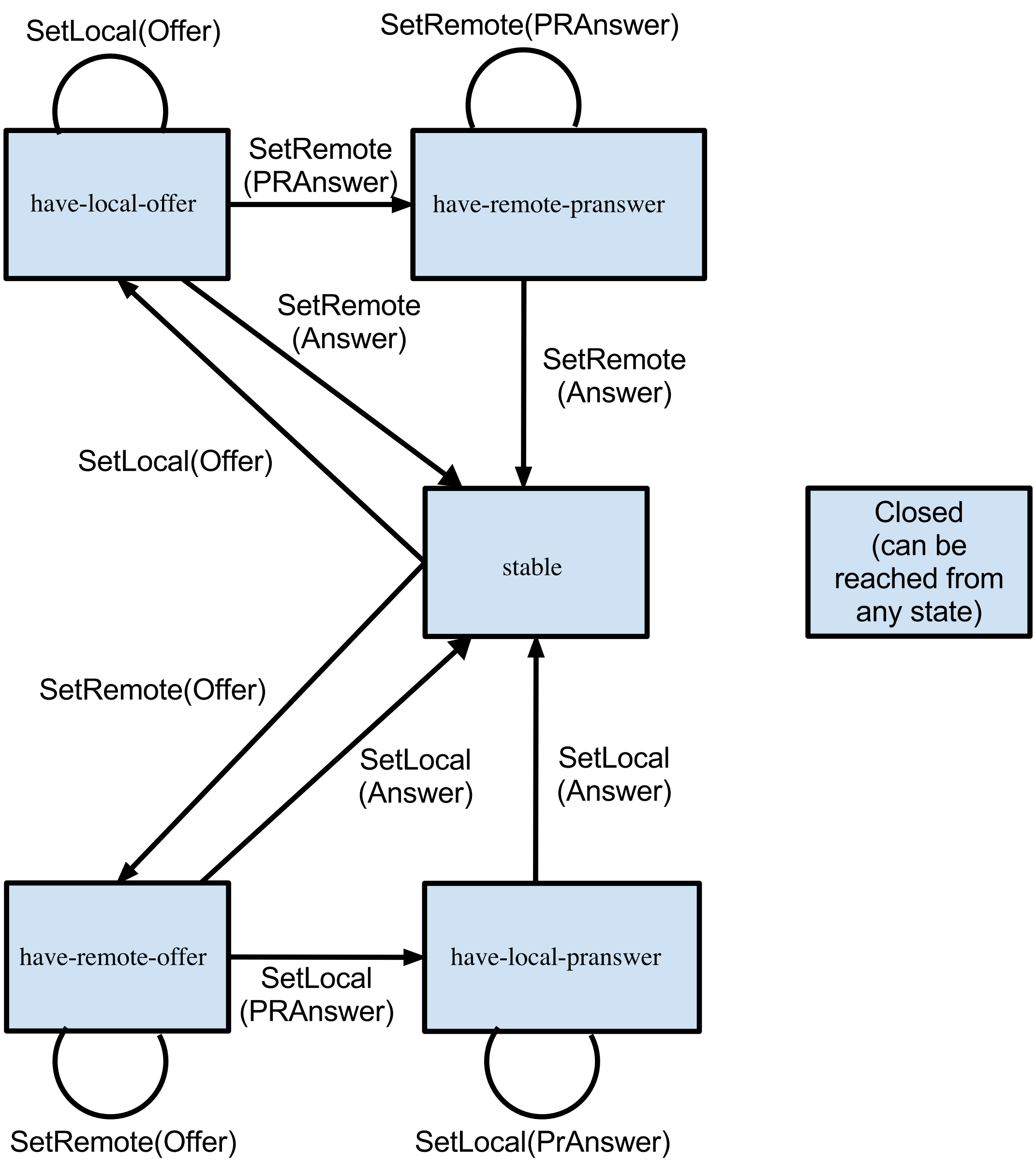


Figure 1 Non-normative signalling state transitions diagram. Method calls abbreviated.

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"
};
```

Enumeration 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 `"completed"` gathering state.

4.3.3 `RTCPeerConnectionState` Enum

WebIDL

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

Enumeration description	
`new`	Any of the `RTCIceTransport`s or `RTCDtlsTransport`s are in the `"new"` state and none of the transports are in the `"connecting"`, `"checking"`, `"failed"` or `"disconnected"` state, or all transports are in the `"closed"` state, or there are no transports.
`connecting`	Any of the `RTCIceTransport`s or `RTCDtlsTransport`s are in the `"connecting"` or `"checking"` state and none of them is in the `"failed"` state.
`connected`	All `RTCIceTransport`s and `RTCDtlsTransport`s are in the `"connected"`, `"completed"` or `"closed"` state and at least one of them is in the `"connected"` or `"completed"` state.
`disconnected`	Any of the `RTCIceTransport`s or `RTCDtlsTransport`s are in the `"disconnected"` state and none of them are in the `"failed"` or `"connecting"` or `"checking"` state.
`failed`	Any of the `RTCIceTransport`s or `RTCDtlsTransport`s are in a `"failed"` state.
`closed`	The `RTCPeerConnection` object's [[IsClosed]] slot is `true`.

4.3.4 `RTCIceConnectionState` Enum

WebIDL

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

Enumeration description	
`new`	Any of the `RTCIceTransport`s are in the `"new"` state and none of them are in the `"checking"`, `"disconnected"` or `"failed"` state, or all `RTCIceTransport`s are in the `"closed"` state, or there are no transports.
`checking`	Any of the `RTCIceTransport`s are in the `"checking"` state and none of them are in the `"disconnected"` or `"failed"` state.
`connected`	All `RTCIceTransport`s are in the `"connected"`, `"completed"` or `"closed"` state and at least one of them is in the `"connected"` state.
`completed`	All `RTCIceTransport`s are in the `"completed"` or `"closed"` state and at least one of them is in the `"completed"` state.
`disconnected`	Any of the `RTCIceTransport`s are in the `"disconnected"` state and none of them are in the `"failed"` state.
`failed`	Any of the `RTCIceTransport`s are in the `"failed"` state.
`closed`	The `RTCPeerConnection` object's [[IsClosed]] slot is `true`.

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

4.4.1 Operation

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

`configuration.servers` 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 **signaling state**, a **connection state**, an **ICE gathering state**, and an **ICE connection state**. These are initialized when the object is created.

The ICE protocol implementation of an `RTCPeerConnection` is represented by an **ICE agent** [ICE]. 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 [JSEP]. 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.

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" field set to an appropriate description.

2. Let *connection* be a newly created `RTCPeerConnection` object.

3. If the `certificates` value in *configuration* is non-empty, check that the `expires` on each value is in the future. If a certificate has expired or a the [[Origin]] internal slot of the certificate does not match the current origin, throw an `InvalidAccessError`; otherwise, store the certificates. If no `certificates` value was specified, one or more new `RTCCertificate` instances are generated for use with this `RTCPeerConnection` instance. This *MAY* happen asynchronously and the value of `certificates` remains undefined for the subsequent steps. As noted in Section 4.3.2.3 of [RTCWEB-SECURITY], 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.

4. Initialize *connection*'s ICE Agent.

5. Let *connection* have a **[[Configuration]]** internal slot. Set the configuration specified by *configuration*.
6. Let *connection* have an **[[IsClosed]]** internal slot, initialized to `false`.
7. Let *connection* have a **[[NegotiationNeeded]]** internal slot, initialized to `false`.
8. Let *connection* have an **[[SctpTransport]]** internal slot, initialized to `null`.
9. Let *connection* have an **[[Operations]]** internal slot, representing an operations queue, initialized to an empty list.
10. Let *connection* have an **[[LastOffer]]** internal slot, initialized to "".
11. Let *connection* have an **[[LastAnswer]]** internal slot, initialized to "".
12. Set *connection*'s signaling state to `"stable"`.
13. Set *connection*'s ICE connection state to `"new"`.
14. Set *connection*'s ICE gathering state to `"new"`.
15. Set *connection*'s connection state to `"new"`.
16. Let *connection* have a **[[PendingLocalDescription]]** internal slot, initialized to `null`.
17. Let *connection* have a **[[CurrentLocalDescription]]** internal slot, initialized to `null`.
18. Let *connection* have a **[[PendingRemoteDescription]]** internal slot, initialized to `null`.
19. Let *connection* have a **[[CurrentRemoteDescription]]** internal slot, initialized to `null`.
20. Return *connection*.

4.4.1.2 Enqueue an operation

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

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

1. Let *connection* be the `RTCPeerConnection` object.

2. If *connection*'s [[IsClosed]] slot is `true`, return a promise rejected with a newly created `InvalidStateError`.
3. Let *operation* be the operation to be enqueued.
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*'s [[IsClosed]] slot 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*'s [[IsClosed]] slot 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]].
8. Return *p*.

4.4.1.3 Update the connection state

An `RTCPeerConnection` object has an aggregated connection state. Whenever the state of an `RTCDtlsTransport` or `RTCIceTransport` changes or when the [[IsClosed]] slot turns `true`, the user agent *MUST* **update the connection state** by queueing a task that runs the following steps:

1. Let *connection* be this `RTCPeerConnection` object.
2. Let *newState* be the value of deriving a new state value as described by the `RTCPeerConnectionState` enum.
3. If *connection*'s connection state is equal to *newState*, abort these steps.

4. Let *connection*'s connection state be *newState*.

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

4.4.1.4 Update the ICE gathering state

To **update the ICE gathering state** of an `RTCPeerConnection` instance *connection*, the user agent *MUST* queue a task that runs the following steps:

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

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

3. If *connection*'s ICE gathering state is equal to *newState*, abort these steps.

4. Set *connection*'s ice gathering state to *newState*.

5. Fire an event named `icegatheringstatechange` at *connection*.

6. If *newState* is `"completed"`, 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`.***

4.4.1.5 Update the ICE connection state

To **update the ICE connection state** of an `RTCPeerConnection` instance *connection*, the user agent *MUST* queue a task that runs the following steps:

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

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

3. If *connection*'s ICE connection state is equal to *newState*, abort these steps.

4. Set *connection*'s ice connection state to *newState*.

5. Fire an event named `iceconnectionstatechange` at *connection*.

4.4.1.6 Set the RTCSessionDescription

To **set an RTCSessionDescription** *description* on an `RTCPeerConnection` object *connection*, enqueue the following steps to *connection*'s operation queue:

1. Let *p* be a new promise.

2. In parallel, start the process to apply *description* as described in [JSEP] (section 5.5. and section 5.6.).

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

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

        2. If the *description*'s `type` is invalid for the current signaling state of *connection* as described in [JSEP] (section 5.5. and section 5.6.), then reject *p* with a newly created `InvalidStateError` and abort these steps.

        3. If *description* is set as a local description, if *`description`*`.type` is `offer` and *`description`*`.sdp` is not equal to *connection*'s [[LastOffer]] slot, then reject *p* with a newly created `InvalidModificationError` and abort these steps.

        4. If *description* is set as a local description, if *`description`*`.type` is `"rollback"` and signaling state is `"stable"` then reject *p* with a newly created `InvalidStateError` and abort these steps.

        5. If *description* is set as a local description, if *`description`*`.type` is `"answer"` or `"pranswer"` and *`description`*`.sdp` is not equal to *connection*'s [[LastAnswer]] slot, then reject *p* with a newly created `InvalidModificationError` and abort these steps.

        6. 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.

        7. If *description* is set as a remote description, the *connection*'s `RTCRtcpMuxPolicy` is `require` and the remote description does not use RTCP mux, then reject *p* with a newly created `InvalidAccessError` and abort these steps.

8. If the content of *description* is invalid, then reject *p* with a newly created `InvalidAccessError` and abort these steps.

9. For all other errors, reject *p* with a newly created `OperationError`.

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

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

   2. If *description* is set as a local description, then run one of the following steps:

      - If *description* is of type `"offer"`, set *connection*.[[PendingLocalDescription]] to a new `RTCSessionDescription` object constructed from *description*, and set signaling state to `"have-local-offer"`.

      - 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`. Finally set *connection*'s signaling state to `"stable"`.

      - If *description* is of type `"rollback"`, then this is a rollback. Set *connection*.[[PendingLocalDescription]] to `null`, and set signaling state to `"stable"`.

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

   3. Otherwise, if *description* is set as a remote description, then run one of the following steps:

      - If *description* is set as a remote description, if *`description`*`.type` is `"rollback"` and signaling state is `"stable"` then reject *p* with a newly created `InvalidStateError` and abort these steps.

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

- 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. Finally set *connection*'s signaling state to "stable".

- If *description* is of type "rollback", then this is a rollback. Set *connection*.[[PendingRemoteDescription]] to null, and set signaling state to "stable".

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

4. If *description* is of type "answer", and it initiates the closure of an existing SCTP association, as defined in [SCTP-SDP], Sections 10.3 and 10.4, set the value of *connection*'s [[SctpTransport]] internal slot to null.

5. 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 [SCTP-SDP], Sections 10.3 and 10.4, create an RTCSctpTransport with an initial state of "connecting" and assign the result to the [[SctpTransport]] slot.

   2. Otherwise, if an SCTP association is established, but the "max-message-size" SDP attribute is updated, update the data max message size of *connection*'s [[SctpTransport]].

   3. If *description* negotiates the DTLS role of the SCTP transport, and there is an RTCDataChannel with a null id, then generate an ID according to [RTCWEB-DATA-PROTOCOL]. If no available ID could be generated, then run the following steps:

      1. Let *channel* be the RTCDataChannel object for which an ID could not be generated.

      2. Set *channel*'s [[ReadyState]] slot 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. Let *trackEventInits*, *muteTracks*, *addList*, and *removeList* be empty lists.

7. If *description* is set as a local description, then run the following steps:

    1. Run the following steps for each media description in *description*:

        1. If the media description is 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*'s `mid` value to the mid of the media description.

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

            4. If the media description is indicated as using an existing media transport according to [BUNDLE], let *transport* and *rtcpTransport* be the `RTCDtlsTransport` objects representing the RTP and RTCP components of that transport, respectively.

            5. Otherwise, let *transport* and *rtcpTransport* be newly created `RTCDtlsTransport` objects, each with a new underlying `RTCIceTransport`. Though if RTCP multiplexing is negotiated according to [RFC5761], or if *connection*'s `RTCRtcpMuxPolicy` is `require`, do not create any RTCP-specific transport objects, and instead let *rtcpTransport* equal *transport*.

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

            7. Set *transceiver*.[[Sender]].[[SenderRtcpTransport]] to *rtcpTransport*.

            8. Set *transceiver*.[[Receiver]].[[ReceiverTransport]] to *transport*.

            9. Set *transceiver*.[[Receiver]].[[ReceiverRtcpTransport]] to *rtcpTransport*.

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

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

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

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

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

   1. If *direction* is `"sendonly"` or `"inactive"`, and *transceiver*'s [[FiredDirection]] slot 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 track for the media description, given *transceiver* and *muteTracks*.

   2. Set *transceiver*'s [[CurrentDirection]] and [[FiredDirection]] slots to *direction*.

8. If *description* is set as a remote description, then run the following steps:

   1. Run the following steps for each media description in *description*:

      1. Let *direction* be an `RTCRtpTransceiverDirection` value representing the direction from the media description, but with the send and receive directions reversed to represent this peer's point of view.

      2. As described by [JSEP] (section 5.10.), attempt to find an existing `RTCRtpTransceiver` object, *transceiver*, to represent the media description.

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

         1. Create an RTCRtpSender, *sender*, from the media description.

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

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

4. Set *transceiver*'s `mid` value to the mid of the corresponding media description. If the media description has no MID, and *transceiver*'s `mid` is unset, generate a random value as described in [JSEP] (section 5.10.).

5. 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.

6. Set the associated remote streams given *transceiver*.[[Receiver]], *msids*, *addList*, and *removeList*.

7. If the previous step increased the length of *addList*, or if *transceiver*'s [[FiredDirection]] slot is neither `"sendrecv"` nor `"recvonly"`, process the addition of a remote track for the media description, given *transceiver* and *trackEventInits*.

8. If *direction* is `"sendonly"` or `"inactive"`, set *transceiver*'s [[Receptive]] slot to `false`.

9. If *direction* is `"sendonly"` or `"inactive"`, and *transceiver*'s [[FiredDirection]] slot is either `"sendrecv"` or `"recvonly"`, process the removal of a remote track for the media description, given *transceiver* and *muteTracks*.

10. Set *transceiver*'s [[FiredDirection]] slot to *direction*.

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

    1. Set *transceiver*'s [[CurrentDirection]] slot to *direction*.

    2. Let *transport* and *rtcpTransport* be the `RTCDtlsTransport` objects representing the RTP and RTCP components of the media transport used by *transceiver*'s associated media description, according to [BUNDLE].

    3. Set *transceiver*.[[Sender]].[[SenderTransport]] to *transport*.

    4. Set *transceiver*.[[Sender]].[[SenderRtcpTransport]] to *rtcpTransport*.

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

6. Set *transceiver*.[[Receiver]].[[ReceiverRtcpTransport]] to *rtcpTransport*.

12. If the media description is rejected, and *transceiver* is not already stopped, stop the RTCRtpTransceiver *transceiver*.

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

   1. If the `mid` value of an `RTCRtpTransceiver` was set to a non-null value by the `RTCSessionDescription` that is being rolled back, set the `mid` value of that transceiver to null, as described by [JSEP] (section 4.1.8.2.).

   2. If an `RTCRtpTransceiver` was created by applying the `RTCSessionDescription` that is being rolled back, and a track has not been attached to it via `addTrack`, remove that transceiver from *connection*'s set of transceivers, as described by [JSEP] (section 4.1.8.2.).

   3. For the `RTCRtpTransceiver`s remaining on *connection*, revert any changes to the [[CurrentDirection]] and [[Receptive]] internal slots made by the application of the `RTCSessionDescription` that is being rolled back.

   4. Restore the value of *connection*'s [[SctpTransport]] internal slot to its value at the last `stable` signaling state.

10. If *connection*'s signaling state changed above, fire an event named `signalingstatechange` at *connection*.

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

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

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

14. 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.

15. If *connection*'s signaling state is now `"stable"`, update the negotiation-needed flag. If *connection*'s [[NegotiationNeeded]] slot was `true` both before and after this update, queue a task that runs the following steps:

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

        2. If *connection*'s [[NegotiationNeeded]] slot is `false`, abort these steps.

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

    16. Resolve *p* with *undefined*.

3. Return *p*.

4.4.1.7 Set the configuration

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

1. Let *configuration* be the `RTCConfiguration` dictionary to be processed.
2. Let *connection* be the target `RTCPeerConnection` object.
3. If *`configuration`*`.peerIdentity` is set and its value differs from the target peer identity, throw an `InvalidModificationError`.
4. If *`configuration`*`.certificates` is set and the set of certificates differs from the ones used when *connection* was constructed, throw an `InvalidModificationError`.
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`. If the value is `"negotiate"` and the user agent does not implement non-muxed RTCP, throw a `NotSupportedError`.
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. Set the ICE Agent's **ICE transports setting** to the value of *`configuration`*`.iceTransportPolicy`. As defined in [JSEP] (section 4.1.16.), if the new 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.

9. Set the ICE Agent's prefetched **ICE candidate pool size** as defined in [JSEP] (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 [JSEP] (section 4.1.16.).

10. Let *validatedServers* be an empty list.

11. 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 following steps:

        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 [RFC7064] fails, throw a SyntaxError. If *scheme name* is stun or stuns, and parsing the *url* using the syntax defined in [RFC7065] fails, throw a SyntaxError.

        2. If *scheme name* is turn or turns, and either of *server*.username or *server*.credential are omitted, then throw an InvalidAccessError.

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

        4. If *scheme name* is turn or turns, and *server*.credentialType is "oauth", and *server*.credential is not an RTCOAuthCredential, then throw an InvalidAccessError.

    6. Append *server* to *validatedServers*.

    Let *validatedServers* be the ICE Agent's **ICE servers list**.

    As defined in [JSEP] (section 4.1.16.), if a new list of servers replaces the ICE Agent's existing 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 candidate pool has a nonzero size, any existing pooled candidates will be discarded, and new candidates will be gathered from the new servers.

12. Store the configuration in the [[Configuration]] internal slot.

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

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

```
                attribute EventHandler
onicegatheringstatechange;
                attribute EventHandler
onconnectionstatechange;
};
```

Constructors

RTCPeerConnection
: See the `RTCPeerConnection constructor algorithm`.

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 signaling state.

`iceGatheringState` of type *RTCIceGatheringState*, readonly

The **`iceGatheringState`** attribute *MUST* return the ICE gathering state of the `RTCPeerConnection` instance.

`iceConnectionState` of type *RTCIceConnectionState*, readonly

The **`iceConnectionState`** attribute *MUST* return the ICE connection state of the `RTCPeerConnection` instance.

`connectionState` of type *RTCPeerConnectionState*, readonly

The **`connectionState`** attribute *MUST* return the connection state of the `RTCPeerConnection` instance.

`canTrickleIceCandidates` of type boolean, readonly, nullable

The **`canTrickleIceCandidates`** attribute indicates whether the remote peer is able to accept trickled ICE candidates [TRICKLE-ICE]. The value is determined based on whether a remote description indicates support for trickle ICE, as defined in [JSEP] (section 4.1.15.). 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 [JSEP]. As an offer, the generated SDP will contain the full set of codec/RTP/RTCP capabilities supported 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 [ICE], 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 and as input to requests for identity assertions.

If the RTCPeerConnection is configured to generate Identity assertions by calling setIdentityProvider, then the session description *SHALL* contain an appropriate assertion.

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*'s [[IsClosed]] slot is true, return a promise rejected with a newly created InvalidStateError.
3. If *connection* is configured with an identity provider, then begin the identity assertion request process if it has not already begun.
4. Return the result of enqueuing the following steps to *connection*'s operation queue:
    1. Let *p* be a new promise.
    2. In parallel, begin the steps to create an offer, given *p*.
    3. Return *p*.

The **steps to create an offer** given 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. Let *provider* be *connection*'s currently configured identity provider if one has been configured, or `null` otherwise.

3. If *provider* is non-null, wait for the identity assertion request process to complete.

4. If *provider* was unable to produce an identity assertion, reject *p* with a newly created `NotReadableError` and abort these steps.

5. Inspect the system state to determine the currently available resources as necessary for generating the offer, as described in [JSEP] (section 4.1.6.).

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

7. Queue a task that runs the final steps to create an offer, given *p*.

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

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

2. If *connection* was modified in such a way that additional inspection of the system state is necessary, or if its configured indentity provider is no longer *provider*, then in parallel begin the steps to create an offer again, given *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 steps to create an offer in parallel, 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, and the identity assertion from *provider* (if non-null), generate an SDP offer, *sdpString*, as described in [JSEP] (section 5.2.).

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 [[LastOffer]] 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 `MediaStreamTrack`s 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 [JSEP].

The generated SDP will also contain the ICE agent's `usernameFragment`, `password` and ICE options (as defined in [ICE], 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 and as input to requests for identity assertions.

An answer can be marked as provisional, as described in [JSEP] (section 4.1.8.1.), by setting the `type` to `"pranswer"`.

If the `RTCPeerConnection` is configured to generate Identity assertions by calling `setIdentityProvider`, then the session description *SHALL* contain an appropriate assertion.

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*'s [[IsClosed]] slot is `true`, return a promise rejected with a newly created `InvalidStateError`.

3. If *connection* is configured with an identity provider, then begin the identity assertion request process if it has not already begun.

4. Return the result of enqueuing the following steps to *connection*'s operation queue:

    1. If *connection*'s signaling state 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 steps to create an answer, given *p*.

    4. Return *p*.

The **steps to create an answer** given 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. Let *provider* be *connection*'s currently configured identity provider if one has been configured, or `null` otherwise.

3. If *provider* is non-null, wait for the identity assertion request process to complete.

4. If *provider* was unable to produce an identity assertion, reject *p* with a newly created `NotReadableError` and abort these steps.

5. Inspect the system state to determine the currently available resources as necessary for generating the answer, as described in [JSEP] (section 4.1.7.).

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

7. 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*'s [[IsClosed]] slot is `true`, then abort these steps.

2. If *connection* was modified in such a way that additional inspection of the system state is necessary, or if its configured indentity provider is no longer *provider*, then

in parallel begin the steps to create an answer again, given *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 steps to create an answer in parallel, 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, and the identity assertion from *provider* (if non-null), generate an SDP answer, *sdpString*, as described in [JSEP] (section 5.3.).

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 [[LastAnswer]] internal slot to *sdpString*.

6. Resolve *p* with *answer*.

`setLocalDescription`

The **`setLocalDescription`** method instructs the `RTCPeerConnection` to apply the supplied `RTCSessionDescriptionInit` 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.

As noted in [JSEP] (section 5.4.) the SDP returned from `createOffer` or `createAnswer` *MUST NOT* be changed before passing it to `setLocalDescription`. As a result, when the method is invoked, the user agent *MUST* run the following steps:

1. Let *description* be the first argument to `setLocalDescription`.

2. If *`description`*`.sdp` is the empty string and *`description`*`.type` is `"answer"` or `"pranswer"`, set *`description`*`.sdp` to the value of *connection*'s

[[LastAnswer]] slot.

3. If `description.sdp` is the empty string and `description.type` is `"offer"`, set `description.sdp` to the value of *connection*'s [[LastOffer]] slot.
4. Return the result of setting the RTCSessionDescription indicated by `description`.

> ***NOTE***
> ***As noted in [JSEP] (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* return the result of setting the RTCSessionDescription indicated by the method's first argument.

In addition, a remote description is processed to determine and verify the identity of the peer.

If an `a=identity` attribute is present in the session description, the browser validates the identity assertion..

If the `peerIdentity` configuration is applied to the `RTCPeerConnection`, this establishes a **target peer identity** of the provided value. Alternatively, if the `RTCPeerConnection` has previously authenticated the identity of the peer (that is, the `peerIdentity` promise is resolved), then this also establishes a target peer identity.

The target peer identity cannot be changed once set.

If a target peer identity is set, then the identity validation *MUST* be completed before the promise returned `setRemoteDescription` is resolved. If identity validation fails, then the promise returned by `setRemoteDescription` is rejected.

If there is no target peer identity, then `setRemoteDescription` does not await the completion of identity validation.

`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 both *sdpMid* and *sdpMLineIndex* are `null`, return a promise rejected with a newly created `TypeError`.

4. Return the result of enqueuing the following steps to *connection*'s operation queue:

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

    2. Let *p* be a new promise.

    3. 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`, reject *p* with a newly created `OperationError` and abort these steps.

    4. 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`, reject *p* with a newly created `OperationError` and abort these steps.

    5. If `candidate.usernameFragment` is neither `undefined` nor `null`, and is not equal to any username fragment present in the corresponding media description of an applied remote description, reject *p* with a newly created `OperationError` and abort these steps.

    6. In parallel, add the ICE candidate *candidate* as described in [JSEP] (section 4.1.17.). 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.

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

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

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

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

        1. If *connection*'s [[IsClosed]] slot 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 the *connection*. [[PendingRemoteDescription]].sdp.

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

        4. Resolve *p* with `undefined`.

7. Return *p*.

`getDefaultIceServers`

Returns a list of ICE servers that are configured into the browser. A browser might be configured to use local or private STUN or TURN servers. This method allows an application to learn about these servers and optionally use them.

This list is likely to be persistent and is the same across origins. It thus increases the fingerprinting surface of the browser. In privacy-sensitive contexts, browsers can consider mitigations such as only providing this data to whitelisted origins (or not providing it at all.)

> ***NOTE***
>
> ***Since the use of this information is left to the discretion of application developers, configuring a user agent with these defaults does not per se increase a user's ability to limit the exposure of their IP addresses.***

`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 [JSEP] (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*'s [[IsClosed]] slot 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. If *connection*'s [[IsClosed]] slot is `true`, abort these steps.
3. Set *connection*'s [[IsClosed]] slot to `true`.
4. Set *connection*'s signaling state to `"closed"`.
5. Let *transceivers* be the result of executing the `CollectTransceivers` algorithm. For every `RTCRtpTransceiver` *transceiver* in *transceivers*, run the following steps:
    1. If *transceiver*'s [[Stopped]] slot is `true`, abort these steps.
    2. Let *sender* be *transceiver*'s [[Sender]].
    3. Let *receiver* be *transceiver*'s [[Receiver]].
    4. Stop sending media with *sender*.
    5. Send an RTCP BYE for each RTP stream that was being sent by *sender*, as specified in [RFC3550].
    6. Stop receiving media with *receiver*.

    7. Set the `readyState` of *receiver*'s [[ReceiverTrack]] to `"ended"`.

    8. Set *transceiver*'s [[Stopped]] slot to `true`.

6. 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.***

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

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

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

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

11. Set *connection*'s ICE connection state to `"closed"`.

12. Set *connection*'s connection state to `"closed"`.

4.4.3 Legacy Interface Extensions

> ***NOTE***
> ***This section is broken out for readability. Consider partial interfaces here to be part of their main counterparts, as they overload existing methods.***

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

WebIDL

```
partial interface RTCPeerConnection {
    Promise<void> createOffer(RTCSessionDescriptionCallback
successCallback,
                              RTCPeerConnectionErrorCallback
failureCallback,
                              optional RTCOfferOptions options);
    Promise<void> setLocalDescription(RTCSessionDescriptionInit
description,
                                      VoidFunction successCallback,
                                      RTCPeerConnectionErrorCallback
failureCallback);
    Promise<void> createAnswer(RTCSessionDescriptionCallback
successCallback,
                               RTCPeerConnectionErrorCallback
failureCallback);
    Promise<void> setRemoteDescription(RTCSessionDescriptionInit
description,
                                       VoidFunction successCallback,
                                       RTCPeerConnectionErrorCallback
failureCallback);
    Promise<void> addIceCandidate(RTCIceCandidateInit candidate,
                                  VoidFunction successCallback,
                                  RTCPeerConnectionErrorCallback
failureCallback);
};
```

`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 = void (DOMException error);
```

Callback `RTCPeerConnectionErrorCallback` Parameters

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

`RTCSessionDescriptionCallback`

WebIDL

```
callback RTCSessionDescriptionCallback = void
(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*'s [[Direction]] slot to "sendonly".
        2. For each non-stopped "recvonly" transceiver of transceiver kind *kind*, set *transceiver*'s [[Direction]] slot 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*'s [[Direction]] slot to "recvonly".
4. Run the steps specified by createOffer to create the offer.

WebIDL

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

`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` or `RTCSessionDescription` instance.

WebIDL

```
enum RTCSdpType {
    "offer",
    "pranswer",
    "answer",
    "rollback"
};
```

Enumeration 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 and answer 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.

4.6.2 `RTCSessionDescription` Class

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

WebIDL

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

Constructors

`RTCSessionDescription`

The `RTCSessionDescription()` constructor takes a dictionary argument, *descriptionInitDict*, 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 RTCSessionDescription.

`sdp` of type DOMString, readonly

The string representation of the SDP [SDP].

Methods

`toJSON()`

When called, run [WEBIDL]'s default toJSON operation.

WebIDL

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

Dictionary `RTCSessionDescriptionInit` *Members*

type of type *RTCSdpType*, required
DOMString sdp

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 an `RTCSessionDescription` of type "answer" is applied, 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* queue a task to update the negotiation-needed flag.

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

1. If *connection*'s [[IsClosed]] slot is `true`, abort these steps.
2. If *connection*'s signaling state 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 an RTCSessionDescription.***

3. If the result of checking if negotiation is needed is `false`, **clear the negotiation-needed flag** by setting *connection*'s [[NegotiationNeeded]] slot to `false`, and abort these steps.
4. If *connection*'s [[NegotiationNeeded]] slot is already `true`, abort these steps.
5. Set *connection*'s [[NegotiationNeeded]] slot to `true`.
6. Queue a task that runs the following steps:
   1. If *connection*'s [[IsClosed]] slot is `true`, abort these steps.
   2. If *connection*'s [[NegotiationNeeded]] slot is `false`, abort these steps.
   3. Fire an event named `negotiationneeded` at *connection*.

   > ***NOTE***
   >
   > ***This queueing prevents `negotiationneeded` from firing prematurely, in the common situation where multiple modifications to connection are being made at once.***

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

3. If *connection* has created any RTCDataChannels, and no m= section in *description* has been negotiated yet for data, return true.

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

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

   2. 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.

      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 [JSEP] (section 5.3.1.)), return true.

   3. 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.

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

4.8 Interfaces for Connectivity Establishment

4.8.1 RTCIceCandidate Interface

This interface describes an ICE candidate, described in [ICE] 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.

WebIDL

```
[Constructor(optional RTCIceCandidateInit candidateInitDict),
 Exposed=Window]
interface RTCIceCandidate {
    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;
    RTCIceCandidateInit toJSON();
};
```

Constructor

RTCIceCandidate

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 dictionary members in *candidateInitDict* are null, throw a TypeError.
2. Let *iceCandidate* be a newly created RTCIceCandidate object.
3. Initialize the following attributes of *iceCandidate* to null: foundation, component, priority, address, protocol, port, type, tcpType, relatedAddress, and relatedPort.
4. Set the candidate, sdpMid, sdpMLineIndex, usernameFragment attributes of *iceCandidate* with the corresponding dictionary member values of *candidateInitDict*.
5. 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 attributes in *iceCandidate* to the field values of the parsed result.

6. 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 [ICE].

`candidate` of type DOMString, readonly
: This carries the `candidate-attribute` as defined in section 15.1 of [ICE]. If this `RTCIceCandidate` represents an end-of-candidates indication, `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`.

> ***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 [RTCWEB-IP-HANDLING].***

`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 [ICE].

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;
};
```

Dictionary **`RTCIceCandidateInit`** *Members*

`candidate` of type DOMString, defaulting to `""`

This carries the `candidate-attribute` as defined in section 15.1 of [ICE]. 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

This carries the `ufrag` as defined in section 15.4 of [ICE].

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 [ICE]. 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"
};
```

Enumeration description	
udp	A UDP candidate, as described in [ICE].
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"
};
```

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

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 [ICE] section 15.1.

WebIDL

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

Enumeration description	
host	A host candidate, as defined in Section 4.1.1.1 of [ICE].
srflx	A server reflexive candidate, as defined in Section 4.1.1.2 of [ICE].
prflx	A peer reflexive candidate, as defined in Section 4.1.1.2 of [ICE].
relay	A relay candidate, as defined in Section 7.1.3.2.1 of [ICE].

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 [TRICKLE-ICE]. 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

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

Constructors

`RTCPeerConnectionIceEvent`

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`.

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

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

Constructors

`RTCPeerConnectionIceErrorEvent`

Attributes

hostCandidate of type DOMString, readonly

The `hostCandidate` attribute is the local IP address and port 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 use of multiple interfaces has been prohibited for privacy reasons, this attribute will be set to 0.0.0.0:0 or [::]:0, as appropriate.

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      hostCandidate;
                DOMString      url;
    required unsigned short errorCode;
                USVString      statusText;
};
```

Dictionary **RTCPeerConnectionIceErrorEventInit** *Members*

hostCandidate of type DOMString
: The local address and port used to communicate with the STUN or TURN server.

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.

statusText of type USVString
: The STUN reason text returned by the STUN or TURN server.

4.9 Priority and QoS Model

Many applications have multiple media flows of the same data type and often some of the flows are more important than others. WebRTC uses the priority and Quality of Service (QoS) framework described in [RTCWEB-TRANSPORT] and [TSVWG-RTCWEB-QOS] to provide priority and DSCP marking for packets that will help provide QoS in some networking environments. The priority setting can be used to indicate the relative priority of various flows. The priority API allows the JavaScript applications to tell the browser whether a particular media flow is high, medium, low or of very low importance to the application by setting the priority property of RTCRtpEncodingParameters objects to one of the following values.

4.9.1 `RTCPriorityType` Enum

WebIDL

```
enum RTCPriorityType {
    "very-low",
    "low",
    "medium",
    "high"
};
```

Enumeration description	
`very-low`	See [RTCWEB-TRANSPORT], Sections 4.1 and 4.2. Corresponds to "below normal" as defined in [RTCWEB-DATA].
`low`	See [RTCWEB-TRANSPORT], Sections 4.1 and 4.2. Corresponds to "normal" as defined in [RTCWEB-DATA].
`medium`	See [RTCWEB-TRANSPORT], Sections 4.1 and 4.2. Corresponds to "high" as defined in [RTCWEB-DATA].
`high`	See [RTCWEB-TRANSPORT], Sections 4.1 and 4.2. Corresponds to "extra high" as defined in [RTCWEB-DATA].

Applications that use this API should be aware that often better overall user experience is obtained by lowering the priority of things that are not as important rather than raising the priority of the things that are.

4.10 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 and store 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 *keygenAlgorithm* value *MUST* be a valid argument to `window.crypto.subtle.generateKey`; that is, the value *MUST* produce a non-error result when normalized according to the WebCrypto algorithm normalization process [WebCryptoAPI] with an operation name of `generateKey` and a [[supportedAlgorithms]] value specific to production of certificates for `RTCPeerConnection`. If the algorithm normalization process produces an error, the call to `generateCertificate` *MUST* be rejected with that error.

Signatures produced by the generated key are used to authenticate the DTLS connection. The identified algorithm (as identified by the `name` of the normalized `AlgorithmIdentifier`) *MUST* be an asymmetric algorithm that can be used to produce a signature.

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.

A user agent *MUST* reject a call to `generateCertificate()` with a `DOMException` of type `NotSupportedError` if the *keygenAlgorithm* parameter identifies an algorithm that the user agent cannot or will not use to generate a certificate for `RTCPeerConnection`.

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.***

4.10.1 `RTCCertificateExpiration` Dictionary

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

WebIDL

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

`expires`

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 that the `RTCCertificate` is valid for relative to the current time.

When `generateCertificate` is called with an `object` argument, the user agent attempts to convert the object into an `RTCCertificateExpiration`. If this is unsuccessful, immediately return a promise that is rejected with a newly created `TypeError` and abort processing.

A user agent generates a certificate that has an expiration date set to the current time plus the value of the `expires` attribute. The `expires` attribute of the returned `RTCCertificate` is set to the expiration time of the certificate. A user agent *MAY* choose to limit the value of the `expires` attribute.

4.10.2 `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 (**[[KeyingMaterial]]**), a certificate (**[[Certificate]]**) that `RTCPeerConnection` uses to authenticate with a peer, and the origin (**[[Origin]]**) that created the object.

WebIDL

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

Attributes

`expires` of type DOMTimeStamp, 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

`getSupportedAlgorithms`

Returns a sequence providing a representative set of supported certificate algorithms. At least one algorithm *MUST* be returned.

> ***NOTE***
> ***For example, the "RSASSA-PKCS1-v1_5" algorithm dictionary, `RsaHashedKeyGenParams`, contains fields for the modulus length, public exponent, and hash algorithm. Implementations are likely to support a wide range of modulus lengths and exponents, but a finite number of hash algorithms. So in this case, it would be reasonable for the implementation to return one `AlgorithmIdentifier` for each supported hash algorithm that can be used with RSA, using default/recommended values for `modulusLength` and `publicExponent` (such as 1024 and 65537, respectively).***

`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 [[KeyingMaterial]] internal slot. Implementations *MUST* support applications storing and retrieving `RTCCertificate` objects from persistent storage. In implementations where an `RTCCertificate` might not directly hold private keying material (it might be stored in a secure module), a reference to the private key can be held in the [[KeyingMaterial]] internal slot, allowing the private key to be stored and used.

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

1. Set *serialized*.[[Expires]] to the value of *value*'s `expires` attribute.
2. Set *serialized*.[[Certificate]] to a copy of the unstructured binary data in *value*'s [[Certificate]] slot.
3. Set *serialized*.[[Origin]] to a copy of the unstructured binary data in *value*'s [[Origin]] slot.
4. Set *serialized*.[[KeyingMaterial]] to a serialization of the private keying material represented by *value*'s [[KeyingMaterial]] slot.

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

1. Initialize *value*'s `expires` attribute to contain *serialized*.[[Expires]].
2. Set *value*'s [[Certificate]] slot to a copy of *serialized*.[[Certificate]].
3. Set *value*'s [[Origin]] slot to a copy of *serialized*.[[Origin]].
4. Set *value*'s [[KeyingMaterial]] slot to the private key material resulting from deserializing *serialized*.[[KeyingMaterial]]

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` [webmessaging]. 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, in many cases the ID of a track sent by one side will not match the ID of the corresponding track received on the other side; whether the IDs end up matching depends on the relative ordering of calls to `addTrack`, `addTransceiver` and `setRemoteDescription`, and which side generates the offer. Also, if `replaceTrack` is called, changing the track sent by an `RTCRtpSender`, no new track will be created on the receiver side; the corresponding `RTCRtpReceiver` will only have a single track, potentially representing multiple sources of media stiched together. 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.

Following the rules in [JSEP] (section 3.6.), the video *MAY* be downscaled in order to fit the SDP constraints. 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.***

When video is rescaled, for example for certain combinations of width or height and `scaleResolutionDownBy` values, situations when the resulting width or height is not an integer may occur. In such situations the user agent *MUST* use the integer part of the result. What to transmit if the integer part of the scaled width or height is zero is implementation-specific.

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 `RTCRtpReceiver`s. 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; volume, 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 `RTCRtpTransceiver`s**, representing the paired senders and receivers with some shared state. This set is initialized to the empty set when the `RTCPeerConnection` object is created. `RTCRtpSender`s and `RTCRtpReceiver`s are always created at the same time as an `RTCRtpTransceiver`, which they will remain attached to for their lifetime. `RTCRtpTransceiver`s 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.

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);
    void                        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*'s [[IsClosed]] slot 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 track added, as defined in [JSEP] (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 transceiver is not `stopped`. More precisely, the [[Stopped]] 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*'s [[SenderTrack]] to *track*.

   2. Set *sender*'s [[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*'s [[Direction]] slot is `recvonly`, set *transceiver*'s [[Direction]] slot to `sendrecv`.

   6. If *transceiver*'s [[Direction]] slot is `inactive`, set *transceiver*'s [[Direction]] slot 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 being marked with a `peerIdentity` option or 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, unless they are also marked with `peerIdentity` and they meet the requirements for sending (see isolated stream).

    All other tracks that are not accessible to the application *MUST NOT* be sent to the peer, with silence (audio), black frames (video) or equivalently absent content being 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 [JSEP] (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 `muted` event is fired at the track.

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

1. Let *sender* be the argument to `removeTrack`.

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

3. If *connection*'s [[IsClosed]] slot is `true`, throw an `InvalidStateError`.
4. If *sender* was not created by *connection*, throw an `InvalidAccessError`.
5. Let *senders* be the result of executing the `CollectSenders` algorithm.
6. If *sender* is not in *senders* (which indicates that it was removed due to setting an RTCSessionDescription of type "rollback"), then abort these steps.
7. If *sender*'s [[SenderTrack]] is null, abort these steps.
8. Set *sender*'s [[SenderTrack]] to null.
9. Let *transceiver* be the `RTCRtpTransceiver` object corresponding to *sender*.
10. If *transceiver*'s [[Direction]] slot is `sendrecv`, set *transceiver*'s [[Direction]] slot to `recvonly`.
11. If *transceiver*'s [[Direction]] slot is `sendonly`, set *transceiver*'s [[Direction]] slot to `inactive`.
12. 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 [JSEP] (section 5.2.2.).

The initial value of `mid` is null. Setting a new `RTCSessionDescription` may change it to a non-null value, as defined in [JSEP] (section 5.5. and section 5.6.).

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*'s `streams` member.
3. Let *sendEncodings* be *init*'s `sendEncodings` member.
4. Let *direction* be *init*'s `direction` member.
5. If the first argument is a string, let it be *kind* and run the following steps:

1. If *kind* is not a legal `MediaStreamTrack` `kind`, throw a `TypeError`.
2. Let *track* be `null`.

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

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

8. Validate *sendEncodings* by running the following steps:
   1. Verify that each `rid` value in *sendEncodings* is composed only of alphanumeric characters (a-z, A-Z, 0-9) up to a maximum of 16 characters. If one of the RIDs does not meet these requirements, throw a `TypeError`.
   2. If any `RTCRtpEncodingParameters` dictionary in *sendEncodings* contains a read-only parameter other than `rid`, throw an `InvalidAccessError`.
   3. Verify that each `scaleResolutionDownBy` value in *sendEncodings* is greater than or equal to 1.0. If one of the `scaleResolutionDownBy` values does not meet this requirement, throw a `RangeError`.
   4. Verify that each `maxFramerate` value in *sendEncodings* is greater than or equal to 0.0. If one of the `maxFramerate` values does not meet this requirement, throw a `RangeError`.
   5. 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.
   6. If the number of `RTCRtpEncodingParameters` stored in *sendEncodings* exceeds *maxN*, then trim *sendEncodings* from the tail until its length is *maxN*.
   7. If the number of `RTCRtpEncodingParameters` 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 [JSEP] (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 [JSEP] (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 PeerConnection'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"
};
```

RTCRtpTransceiverDirection Enumeration 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.

5.1.1 Processing Remote MediaStreamTracks

An application can reject incoming media descriptions by calling RTCRtpTransceiver.stop() to stop both directions, or set the transceiver's direction to "sendonly" to reject only the incoming side.

To **process the addition of a remote track** for an incoming media description [JSEP] (section 5.10.) given RTCRtpTransceiver *transceiver* and *trackEventInits*, the user agent *MUST* run the following steps:

1. Let *receiver* be *transceiver*'s [[Receiver]].
2. Let *track* be *receiver*'s [[ReceiverTrack]].

3. Let *streams* be *receiver*'s [[AssociatedRemoteMediaStreams]] slot.

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** for an incoming media description [JSEP] (section 5.10.) given `RTCRtpTransceiver` *transceiver* and *muteTracks*, the user agent *MUST* run the following steps:

1. Let *receiver* be *transceiver*'s [[Receiver]].

2. Let *track* be *receiver*'s [[ReceiverTrack]].

3. If *track.muted* is `false`, add *track* to *muteTracks*.

To **set the associated remote streams** given `RTCRtpReceiver` *receiver*, *msids*, *addList*, and *removeList*, the user agent *MUST* 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 `id`s corresponding to *msids*.

4. Let *track* be *receiver*'s [[ReceiverTrack]].

5. For each *stream* in *receiver*'s [[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*'s [[AssociatedRemoteMediaStreams]], add *stream* and *track* as a pair to *addList*.

7. Set *receiver*'s [[AssociatedRemoteMediaStreams]] slot 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 **[[Dtmf]]** internal slot initialized to `null`.
5. If *kind* is `"audio"` then create an `RTCDTMFSender` *dtmf* and set the [[Dtmf]] internal slot to *dtmf*.
6. Let *sender* have a **[[SenderRtcpTransport]]** internal slot initialized to `null`.
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 [JSEP] (section 5.2.1.).
8. Set *sender*'s [[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.
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 `RTCRtpEncodingParameters` with `active` set to `true`.
12. Let *sender* have a **[[LastReturnedParameters]]** internal slot, which will be used to match `getParameters` and `setParameters` transactions.
13. Return *sender*.

WebIDL

```
[Exposed=Window]
interface RTCRtpSender {
    readonly attribute MediaStreamTrack? track;
    readonly attribute RTCDtlsTransport? transport;
    readonly attribute RTCDtlsTransport? rtcpTransport;
    static RTCRtpCapabilities? getCapabilities(DOMString kind);
    Promise<void>                 setParameters(RTCRtpSendParameters
parameters);
    RTCRtpSendParameters          getParameters();
    Promise<void>                 replaceTrack(MediaStreamTrack?
withTrack);
    void                          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 silence (audio), black frames (video) or a zero-information-content equivalent. 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.

rtcpTransport of type *RTCDtlsTransport*, readonly, nullable

The `rtcpTransport` attribute is the transport over which RTCP is sent and received. Prior to construction of the `RTCDtlsTransport` object, the `rtcpTransport` attribute will be null. When RTCP mux is used (or bundling, which mandates RTCP mux), `rtcpTransport` will be null, and both RTP and RTCP traffic will flow over the transport described by `transport`.

On getting, the attribute *MUST* return the value of the [[SenderRtcpTransport]] slot.

Methods

`getCapabilities`, static

The **`getCapabilities()`** method returns the most optimistic view of the capabilities of the system for sending media of the given kind. It does not reserve any resources, ports, or other state but is meant to provide a way to discover the types of capabilities of the browser including which codecs may be supported. User agents *MUST* support *kind* values of `"audio"` and `"video"`. If the system has no capabilities corresponding to the value of the *kind* argument, `getCapabilities` returns `null`.

These capabilities provide generally persistent cross-origin information on the device and thus increases the fingerprinting surface of the application. In privacy-sensitive contexts, browsers can consider mitigations such as reporting only a common subset of the capabilities.

`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*'s [[Sender]]).
4. If *transceiver*'s [[Stopped]] slot is `true`, return a promise rejected with a newly created `InvalidStateError`.
5. If *sender*'s [[LastReturnedParameters]] internal slot is `null`, return a promise rejected with a newly created `InvalidStateError`.
6. Validate *parameters* by running the following steps:
    1. Let *encodings* be `parameters.encodings`.
    2. Let *codecs* be `parameters.codecs`.
    3. Let *N* be the number of `RTCRtpEncodingParameters` stored in *sender*'s internal [[SendEncodings]] slot.

4. If any of the following conditions are met, return a promise rejected with a newly created `InvalidModificationError`:
    1. *encodings*`.length` is different from *N*.
    2. *encodings* has been re-ordered.
    3. 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*'s [[LastReturnedParameters]] internal slot. Note that this also applies to *transactionId*.
5. Verify that each `scaleResolutionDownBy` value in *encodings* is greater than or equal to 1.0. If one of the `scaleResolutionDownBy` values does not meet this requirement, return a promise rejected with a newly created `RangeError`.
6. Verify that each `maxFramerate` value in *encodings* 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`.
7. For each value of *i* from 0 to the number of encodings, check whether *encodings*`[i].codecPayloadType` (if set) corresponds to a value of *codecs*`[j].payloadType` where *j* goes from 0 to the number of codecs. If there is no correspondence, or if the MIME subtype portion of *codecs*`[j].mimeType` is equal to "red", "cn", "telephone-event", "rtx" or a forward error correction codec ("ulpfec" [RFC5109] or "flexfec" [FLEXFEC]), reject *p* with a newly created `InvalidAccessError`.

7. Let *p* be a new promise.
8. In parallel, configure the media stack to use *parameters* to transmit *sender*'s [[SenderTrack]].
    1. If the media stack is successfully configured with *parameters*, queue a task to run the following steps:
        1. Set *sender*'s internal [[LastReturnedParameters]] slot to `null`.
        2. Set *sender*'s internal [[SendEncodings]] slot 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*.

If the application selects a codec via `codecPayloadType`, and this codec is removed from a subsequent offer/answer negotiation, `codecPayloadType` will be unset in the next call to `getParameters`, and the implementation will fall back to its default codec selection policy until a new codec is selected.

`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 `RTCRtpSendParameters` dictionary is constructed as follows:

- `transactionId` is set to a new unique identifier, used to match this `getParameters` call to a `setParameters` call that may occur later.
- `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.
- The `codecs` sequence is populated based on the codecs that have been negotiated for sending, and which the user agent is currently capable of sending. Prior to the completion of negotiation, the `codecs` sequence is empty.
- `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.
- `degradationPreference` is set to the last value passed into `setParameters`, or

the default value of "balanced" if `setParameters` hasn't been called.

The returned `RTCRtpSendParameters` dictionary *MUST* be stored in the `RTCRtpSender` object's [[LastReturnedParameters]] internal slot.

`getParameters` may be used with `setParameters` to change the parameters in the following way:

EXAMPLE 3

```
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.

To avoid track identifiers changing on the remote receiving end when a track is replaced, the sender *MUST* retain the original track identifier and stream associations and use these in subsequent negotiations.

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 enqueuing the following steps to *connection*'s operation queue:

    1. If *transceiver*'s [[Stopped]] slot is `true`, return a promise rejected with a newly created `InvalidStateError`.

    2. Let *p* be a new promise.

    3. Let *sending* be `true` if the *transceiver*'s [[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 *transceiver*'s [[Stopped]] slot is `true`, abort these steps.

            2. Set *sender*'s `track` attribute 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*'s [[IsClosed]] slot 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*'s [[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;
             RTCDegradationPreference                     degradationPreference
= "balanced";
};
```

Dictionary `RTCRtpSendParameters` Members

`transactionId` of type DOMString, required

An 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.

`degradationPreference` of type *RTCDegradationPreference*, defaulting to `"balanced"`

When bandwidth is constrained and the `RtpSender` needs to choose between degrading resolution or degrading framerate, `degradationPreference` indicates which is preferred.

5.2.3 `RTCRtpReceiveParameters` Dictionary

WebIDL

```
dictionary RTCRtpReceiveParameters : RTCRtpParameters {
    required sequence<RTCRtpDecodingParameters> encodings;
};
```

Dictionary `RTCRtpReceiveParameters` Members

`encodings` of type sequence<*RTCRtpDecodingParameters*>, required

A sequence containing information about incoming RTP encodings of media.

FEATURE AT RISK 2

Support for the `encodings` attribute of `RTCRtpReceiveParameters` is marked as a feature at risk, since there is no clear commitment from implementers.

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 [JSEP] (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.

5.2.5 `RTCRtpDecodingParameters` Dictionary

WebIDL

```
dictionary RTCRtpDecodingParameters : RTCRtpCodingParameters {
};
```

5.2.6 `RTCRtpEncodingParameters` Dictionary

WebIDL

```
dictionary RTCRtpEncodingParameters : RTCRtpCodingParameters {
    octet             codecPayloadType;
    RTCDtxStatus      dtx;
    boolean           active = true;
    RTCPriorityType priority = "low";
    unsigned long     ptime;
    unsigned long     maxBitrate;
    double            maxFramerate;
    double            scaleResolutionDownBy;
};
```

Dictionary `RTCRtpEncodingParameters` Members

`codecPayloadType` of type octet

Used to select a codec to be sent. Must reference a payload type from the `codecs` member of `RTCRtpParameters`. If left unset, the implementation will select a codec according to its default policy.

`dtx` of type *RTCDtxStatus*

This member is only used if the sender's `kind` is `"audio"`. It indicates whether discontinuous transmission will be used. Setting it to `disabled` causes discontinuous transmission to be turned off. Setting it to `enabled` causes discontinuous transmission to be turned on if it was negotiated (either via a codec-specific parameter or via negotiation of the CN codec); if it was not negotiated (such as when setting `voiceActivityDetection` to `false`), then discontinuous operation will be turned off regardless of the value of `dtx`, and media will be sent even when silence is detected.

`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.

`priority` of type *RTCPriorityType*, defaulting to **`"low"`**

Indicates the priority of this encoding. It is specified in [RTCWEB-TRANSPORT], Section 4.

`ptime` of type unsigned long

When present, indicates the preferred duration of media represented by a packet in milliseconds for this encoding. Typically, this is only relevant for audio encoding. The user agent *MUST* use this duration if possible, and otherwise use the closest available duration. This value *MUST* take precedence over any "ptime" attribute in the remote description, whose processing is described in [JSEP] (section 5.10.). Note that the user agent *MUST* still respect the limit imposed by any "maxptime" attribute, as defined in [RFC4566], Section 6.

`maxBitrate` of type unsigned long
: When present, indicates the maximum bitrate that can be used to send this encoding. The encoding may also be further constrained by other limits (such as maxFramerate or 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.

`maxFramerate` of type double
: When present, indicates the maximum framerate that can be used to send this encoding, in frames per second.

`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, the sender will not apply any scaling, (i.e., `scaleResolutionDownBy` will be 1.0).

5.2.7 `RTCDtxStatus` Enum

WebIDL

```
enum RTCDtxStatus {
    "disabled",
    "enabled"
};
```

`RTCDtxStatus` Enumeration description	
`disabled`	Discontinuous transmission is disabled.
`enabled`	Discontinuous transmission is enabled if negotiated.

5.2.8 `RTCDegradationPreference` Enum

```
WebIDL

enum RTCDegradationPreference {
    "maintain-framerate",
    "maintain-resolution",
    "balanced"
};
```

`RTCDegradationPreference` Enumeration description	
`maintain-framerate`	Degrade resolution in order to maintain framerate.
`maintain-resolution`	Degrade framerate in order to maintain resolution.
`balanced`	Degrade a balance of framerate and resolution.

5.2.9 `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.

5.2.10 `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`.***

5.2.11 RTCRtpCodecParameters Dictionary

WebIDL

```
dictionary RTCRtpCodecParameters {
    required octet          payloadType;
    required DOMString      mimeType;
    required unsigned long  clockRate;
             unsigned short channels;
             DOMString      sdpFmtpLine;
};
```

Dictionary RTCRtpCodecParameters Members

payloadType of type octet
: The RTP payload type used to identify this codec. Read-only parameter.

mimeType of type DOMString
: 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
: 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 [JSEP] (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.

5.2.12 RTCRtpCapabilities Dictionary

WebIDL

```
dictionary RTCRtpCapabilities {
    required sequence<RTCRtpCodecCapability>           codecs;
    required sequence<RTCRtpHeaderExtensionCapability>
headerExtensions;
};
```

Dictionary `RTCRtpCapabilities` Members

`codecs` of type sequence<*RTCRtpCodecCapability*>, required
: Supported media codecs as well as entries for RTX, RED and FEC mechanisms. There will 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.

5.2.13 `RTCRtpCodecCapability` Dictionary

WebIDL

```
dictionary RTCRtpCodecCapability {
    required DOMString      mimeType;
    required unsigned long  clockRate;
             unsigned short channels;
             DOMString      sdpFmtpLine;
};
```

Dictionary `RTCRtpCodecCapability` Members

The `RTCRtpCodecCapability` dictionary provides information about codec capabilities. Only capability combinations that would utilize distinct payload types in a generated SDP offer are 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.

`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.

5.2.14 `RTCRtpHeaderExtensionCapability` Dictionary

WebIDL

```
dictionary RTCRtpHeaderExtensionCapability {
    DOMString uri;
};
```

Dictionary `RTCRtpHeaderExtensionCapability` Members

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

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*, and optionally an id string, *id*, 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*.
3. Initialize *track.kind* to *kind*.

4. If an id string, *id*, was given as input to this algorithm, initialize *track.id* to *id*. (Otherwise the value generated when *track* was created will be used.)

5. Initialize *track.label* to the result of concatenating the string `"remote "` with *kind*.

6. Initialize *track.readyState* to `live`.

7. Initialize *track.muted* to `true`. See the MediaStreamTrack section about how the `muted` attribute reflects if a `MediaStreamTrack` is receiving media data or not.

8. Let *receiver* have a **[[ReceiverTrack]]** internal slot initialized to *track*.

9. Let *receiver* have a **[[ReceiverTransport]]** internal slot initialized to `null`.

10. Let *receiver* have a **[[ReceiverRtcpTransport]]** internal slot initialized to `null`.

11. 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.

12. Return *receiver*.

WebIDL

```
[Exposed=Window]
interface RTCRtpReceiver {
    readonly attribute MediaStreamTrack  track;
    readonly attribute RTCDtlsTransport? transport;
    readonly attribute RTCDtlsTransport? rtcpTransport;
    static RTCRtpCapabilities?            getCapabilities(DOMString
kind);
    RTCRtpReceiveParameters               getParameters();
    sequence<RTCRtpContributingSource>    getContributingSources();
    sequence<RTCRtpSynchronizationSource>
getSynchronizationSources();
    Promise<RTCStatsReport>               getStats();
};
```

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.

`rtcpTransport` of type *RTCDtlsTransport*, readonly, nullable

The **`rtcpTransport`** attribute is the transport over which RTCP is sent and received. Prior to construction of the `RTCDtlsTransport` object, the `rtcpTransport` attribute will be null. When RTCP mux is used (or bundling, which mandates RTCP mux), `rtcpTransport` will be null, and both RTP and RTCP traffic will flow over `transport`.

On getting, the attribute *MUST* return the value of the [[ReceiverRtcpTransport]] slot.

Methods

`getCapabilities`, static

The **`getCapabilities()`** method returns the most optimistic view of the capabilities of the system for receiving media of the given kind. It does not reserve any resources, ports, or other state but is meant to provide a way to discover the types of capabilities of the browser including which codecs may be supported. User agents *MUST* support *kind* values of `"audio"` and `"video"`. If the system has no capabilities corresponding to the value of the *kind* argument, `getCapabilities` returns `null`.

These capabilities provide generally persistent cross-origin information on the device and thus increases the fingerprinting surface of the application. In privacy-sensitive contexts, browsers can consider mitigations such as reporting only a common subset of the capabilities.

`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:

- `encodings` is populated based on the RIDs present in the current remote description.
- The `headerExtensions` sequence is populated based on the header extensions that the receiver is currently prepared to receive.
- The `codecs` sequence is populated based on the codecs that the receiver is currently prepared to receive.

  > ***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.

`getSynchronizationSources`

Returns an `RTCRtpSynchronizationSource` for each unique SSRC identifier received by this RTCRtpReceiver in the last 10 seconds.

`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, including the most recent time a packet that the source contributed to was played out. The browser *MUST* keep information from RTP packets received in the previous 10 seconds. When the first audio frame contained in an RTP packet is delivered to the RTCRtpReceiver's MediaStreamTrack for playout, the user agent *MUST* queue a task to update the relevant information for the RTCRtpContributingSource and RTCRtpSynchronizationSource dictionaries based on the contents of the packet. The information relevant to the RTCRtpSynchronizationSource dictionary corresponding to the SSRC identifier, is updated each time, and if the RTP packet contains CSRC identifiers, then the information relevant to the RTCRtpContributingSource dictionaries corresponding to those CSRC identifiers is also updated.

> ***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 implementaion does not need to literally queue a task for every packet, 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;
};
```

Dictionary RTCRtpContributingSource Members

`timestamp` of type DOMHighResTimeStamp, required

The timestamp of type DOMHighResTimeStamp [HIGHRES-TIME], indicating the most recent time of playout of media that arrived in an RTP packet originating from this source. The timestamp is defined as `performance.timeOrigin` + `performance.now()` at the time of playout.

`source` of type unsigned long, required
The CSRC or SSRC identifier of the contributing or synchronization source.

`audioLevel` of type double
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 present, otherwise the user agent must compute the value from the audio data (the member must never 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)`.

WebIDL

```
dictionary RTCRtpSynchronizationSource : RTCRtpContributingSource {
    boolean voiceActivityFlag;
};
```

Dictionary RTCRtpSynchronizationSource Members

`voiceActivityFlag` of type boolean
Whether the last RTP packet played from this source contains voice activity (true) or not (false). If the RFC 6464 extension header was not present, or if the peer has signaled that it is not using the V bit by setting the "vad" extension attribute to "off", as described in [RFC6464], Section 4, `voiceActivityFlag` will be absent.

5.4 `RTCRtpTransceiver` Interface

The `RTCRtpTransceiver` interface represents a combination of an `RTCRtpSender` and an `RTCRtpReceiver` that share a common `mid`. As defined in [JSEP] (section 3.4.1.), an `RTCRtpTransceiver` is said to be **associated** with a media description if its `mid` property is non-null; otherwise it is said to be disassociated. Conceptually, an associated transceiver is one that's represented in the last applied session description.

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 **[[Stopped]]** internal slot, initialized to `false`.
5. Let *transceiver* have a **[[Direction]]** internal slot, initialized to *direction*.
6. Let *transceiver* have a **[[Receptive]]** internal slot, initialized to `false`.
7. Let *transceiver* have a **[[CurrentDirection]]** internal slot, initialized to `null`.
8. Let *transceiver* have a **[[FiredDirection]]** internal slot, initialized to `null`.
9. 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 an `RTCSessionDescription`.***

WebIDL

```
[Exposed=Window]
interface RTCRtpTransceiver {
    readonly attribute DOMString?                     mid;
    [SameObject]
    readonly attribute RTCRtpSender                   sender;
    [SameObject]
    readonly attribute RTCRtpReceiver                 receiver;
    readonly attribute boolean                        stopped;
             attribute RTCRtpTransceiverDirection    direction;
    readonly attribute RTCRtpTransceiverDirection? currentDirection;
    void stop();
    void setCodecPreferences(sequence<RTCRtpCodecCapability> codecs);
};
```

Attributes

`mid` of type DOMString, readonly, nullable

The **`mid`** attribute is the `mid` negotatiated and present in the local and remote descriptions as defined in [JSEP] (section 5.2.1. and section 5.3.1.). Before negotiation is complete, the `mid` value may be null. After rollbacks, the value may change from a non-null value to null.

`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.

`stopped` of type boolean, readonly

The `stopped` attribute indicates that the sender of this transceiver will no longer send, and that the receiver will no longer receive. It is true if either `stop` has been called or if setting the local or remote description has caused the `RTCRtpTransceiver` to be stopped. On getting, this attribute *MUST* return the value of the [[Stopped]] slot.

`direction` of type *RTCRtpTransceiverDirection*

As defined in [JSEP] (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 [JSEP] (section 5.2.2. and section 5.3.2.)

On getting, this attribute *MUST* 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 *connection*'s [[IsClosed]] slot is true, throw an InvalidStateError.
4. If *transceiver*'s [[Stopped]] slot is true, throw an InvalidStateError.
5. Let *newDirection* be the argument to the setter.
6. If *newDirection* is equal to *transceiver*'s [[Direction]] slot, abort these steps.
7. Set *transceiver*'s [[Direction]] slot to *newDirection*.
8. Update the negotiation-needed flag for *connection*.

currentDirection of type *RTCRtpTransceiverDirection*, readonly, nullable

As defined in [JSEP] (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, or if the transceiver is stopped, the value is null. On getting, this attribute *MUST* return the value of the [[CurrentDirection]] slot.

Methods

stop

Irreversibly stops the RTCRtpTransceiver. The sender of this transceiver will no longer send, the receiver will no longer receive. Calling stop() updates the negotiation-needed flag for the RTCRtpTransceiver's associated RTCPeerConnection.

Stopping a 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 [JSEP] (section 4.2.1.).

> ***NOTE***
>
> ***If this method is called in between applying a remote offer and creating an answer, and the transceiver is associated with the "offerer tagged" media description as defined in [BUNDLE], this will cause all other transceivers in the bundle group to be stopped as well. To avoid this, one could instead stop the transceiver when `signalingState` is "`stable`" and perform a subsequent offer/answer exchange.***

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*'s [[IsClosed]] slot is `true`, throw an `InvalidStateError`.
4. If *transceiver*'s [[Stopped]] slot is `true`, abort these steps.
5. Stop the RTCRtpTransceiver specified by *transceiver*.
6. Update the negotiation-needed flag for *connection*.

The **stop the RTCRtpTransceiver** algorithm given a *transceiver* is as follows:

1. Let *sender* be *transceiver*'s [[Sender]].
2. Let *receiver* be *transceiver*'s [[Receiver]].
3. Stop sending media with *sender*.
4. Send an RTCP BYE for each RTP stream that was being sent by *sender*, as specified in [RFC3550].
5. Stop receiving media with *receiver*.
6. Execute the steps for *receiver*'s [[ReceiverTrack]] to be ended.
7. Set *transceiver*'s [[Stopped]] slot to `true`.
8. Set *transceiver*'s [[Receptive]] slot to `false`.

9. Set *transceiver*'s [[CurrentDirection]] slot to `null`.

`setCodecPreferences`

The `setCodecPreferences` method overrides the default 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. 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.

The `codecs` sequence passed into `setCodecPreferences` can only contain codecs that are returned by `RTCRtpSender.getCapabilities(kind)` or `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 InvalidAccessError.

> ***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, [JSEP] (section 5.3.1.) allows adding codecs that were not in the offer, so implementations can behave differently.*

5.4.1 Simulcast functionality

Simulcast functionality is provided via the `addTransceiver` method of the `RTCPeerConnection` object and the `setParameters` method of the `RTCRtpSender` object.

The `addTransceiver` method establishes the **simulcast envelope** which includes the maximum number of simulcast streams that can be sent, as well as the ordering of the `encodings`. While characteristics of individual simulcast streams can be modified using the `setParameters` method, the simulcast envelope cannot be changed. 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.

While `setParameters` cannot modify the 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` attribute to `false`, or can be reactivated by setting the `active` attribute to `true`. Using `setParameters`, stream characteristics can be changed by modifying attributes such as `maxBitrate` and `maxFramerate`.

This specification does not define how to configure `createOffer` to receive multiple RTP encodings. However when `setRemoteDescription` is called with a corresponding remote description that is able to send multiple RTP encodings as defined in [JSEP], 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 4

```
// Example of 3-layer spatial simulcast with all but the lowest
resolution layer disabled
var encodings = [
  {rid: 'f', active: false},
  {rid: 'h', active: false, scaleResolutionDownBy: 2.0},
  {rid: 'q', active: true, scaleResolutionDownBy: 4.0}
];

// Example of 3-layer framerate simulcast with the middle layer
disabled
var encodings = [
  {rid: 'f', active: true, maxFramerate: 60},
  {rid: 'h', active: false, maxFramerate: 30},
  {rid: 'q', active: true, maxFramerate: 15}
];
```

5.4.2 "Hold" functionality

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 5

```
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 6

```
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 7

```
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 8

```
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 [BUNDLE].

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`.

When the underlying DTLS transport needs to update the state of the corresponding `RTCDtlsTransport` object, 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*'s [[DtlsTransportState]] slot to *newState*.
4. Fire an event named statechange at *transport*.

WebIDL

```
[Exposed=Window]
interface RTCDtlsTransport : EventTarget {
    readonly attribute RTCIceTransport       transport;
    readonly attribute RTCDtlsTransportState state;
    sequence<ArrayBuffer> getRemoteCertificates();
             attribute EventHandler          onstatechange;
             attribute EventHandler          onerror;
};
```

Attributes

`transport` of type *RTCIceTransport*, readonly

The `transport` 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 certificate chain in use by the remote side, with each certificate encoded in binary Distinguished Encoding Rules (DER) [X690]. `getRemoteCertificates()` will return an empty list prior to selection of the remote certificate, which will be completed by the time `RTCDtlsTransportState` transitions to "connected".

`RTCDtlsTransportState` Enum

WebIDL

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

Enumeration 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()`.
`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).

5.5.1 `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 [BUNDLE].

> ***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.***

When the ICE Agent indicates that it began gathering a generation of candidates for an `RTCIceTransport`, 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*'s [[IsClosed]] slot is `true`, abort these steps.
3. Let *transport* be the `RTCIceTransport` for which candidate gathering began.
4. Set *transport*'s [[IceGathererState]] slot to `gathering`.
5. Fire an event named `gatheringstatechange` at *transport*.
6. Update the ICE gathering state of *connection*.

When the ICE Agent indicates that it finished gathering a generation of candidates for an `RTCIceTransport`, 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*'s [[IsClosed]] slot is `true`, abort these steps.

3. Let *transport* be the `RTCIceTransport` for which candidate gathering finished.

4. Create an `RTCIceCandidate` instance *newCandidate*, with `sdpMid` and `sdpMLineIndex` set to the values associated with this `RTCIceTransport`, with `usernameFragment` set to the username fragment of the generation of candidates for which gathering finished, with `candidate` set to an empty string, and with all other nullable members set to null.

5. Fire an event named `icecandidate` using the `RTCPeerConnectionIceEvent` interface with the candidate attribute set to *newCandidate* at *connection*.

6. 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.***

7. Set *transport*'s [[IceGathererState]] slot to `complete`.

8. Fire an event named `gatheringstatechange` at *transport*.

9. Update the ICE gathering state of *connection*.

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 *connection* be the `RTCPeerConnection` object associated with this ICE Agent.

2. If *connection*'s [[IsClosed]] slot is `true`, abort these steps.

3. Let *transport* be the `RTCIceTransport` for which this candidate is being made available.

4. If *connection*.[[PendingLocalDescription]] is not `null`, and represents the ICE generation for which *candidate* was gathered, add *candidate* to the *connection*.[[PendingLocalDescription]].sdp.

5. If *connection*.[[CurrentLocalDescription]] is not `null`, and represents the ICE generation for which *candidate* was gathered, add *candidate* to the *connection*.[[CurrentLocalDescription]].sdp.

6. Create an `RTCIceCandidate` instance to represent the candidate. Let *newCandidate* be that object.

7. Add *newCandidate* to *transport*'s set of local candidates.
8. Fire an event named `icecandidate` using the `RTCPeerConnectionIceEvent` interface with the candidate attribute set to *newCandidate* at *connection*.

When the ICE Agent indicates that the `RTCIceTransportState` for an `RTCIceTransport` has changed, 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*'s [[IsClosed]] slot is `true`, abort these steps.
3. Let *transport* be the `RTCIceTransport` whose state is changing.
4. Let *newState* be the new indicated `RTCIceTransportState`.
5. Set *transport*'s [[IceTransportState]] slot to *newState*.
6. Fire an event named `statechange` at *transport*.
7. Update the ICE connection state of *connection*.
8. Update the connection state of *connection*.

When the ICE Agent indicates that the selected candidate pair for an `RTCIceTransport` has changed, 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*'s [[IsClosed]] slot is `true`, abort these steps.
3. Let *transport* be the `RTCIceTransport` whose selected candidate pair is changing.
4. Let *newCandidatePair* be a newly created `RTCIceCandidatePair` representing the indicated pair if one is selected, and `null` otherwise.
5. Set *transport*'s [[SelectedCandidatePair]] slot to *newCandidatePair*.
6. Fire an event named `selectedcandidatepairchange` at *transport*.

An `RTCIceTransport` object has the following internal slots:

- **[[IceTransportState]]** initialized to `new`
- **[[IceGathererState]]** initialized to `new`
- **[[SelectedCandidatePair]]** initialized to `null`

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* return the ICE role of the transport.

`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 **`gathering state`** 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`gathering state 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()`

`getSelectedCandidatePair`

Returns the selected candidate pair on which packets are sent. This method *MUST* return the value of the [[SelectedCandidatePair]] slot.

`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 [ICE], Section 7.1.2.3.

`password` of type DOMString

The ICE password as defined in [ICE], Section 7.1.2.3.

5.6.2 `RTCIceCandidatePair` Dictionary

WebIDL

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

Dictionary *RTCIceCandidatePair* *Members*

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

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

5.6.3 `RTCIceGathererState` Enum

WebIDL

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

RTCIceGathererState Enumeration 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 {
    "new",
    "checking",
    "connected",
    "completed",
    "disconnected",
    "failed",
    "closed"
};
```

RTCIceTransportState Enumeration description	
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 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.
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).
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".
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.
`failed`	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. This is a terminal state.
`closed`	The `RTCIceTransport` has shut down and is no longer responding to STUN requests.

An ICE restart causes candidate gathering and connectity 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`

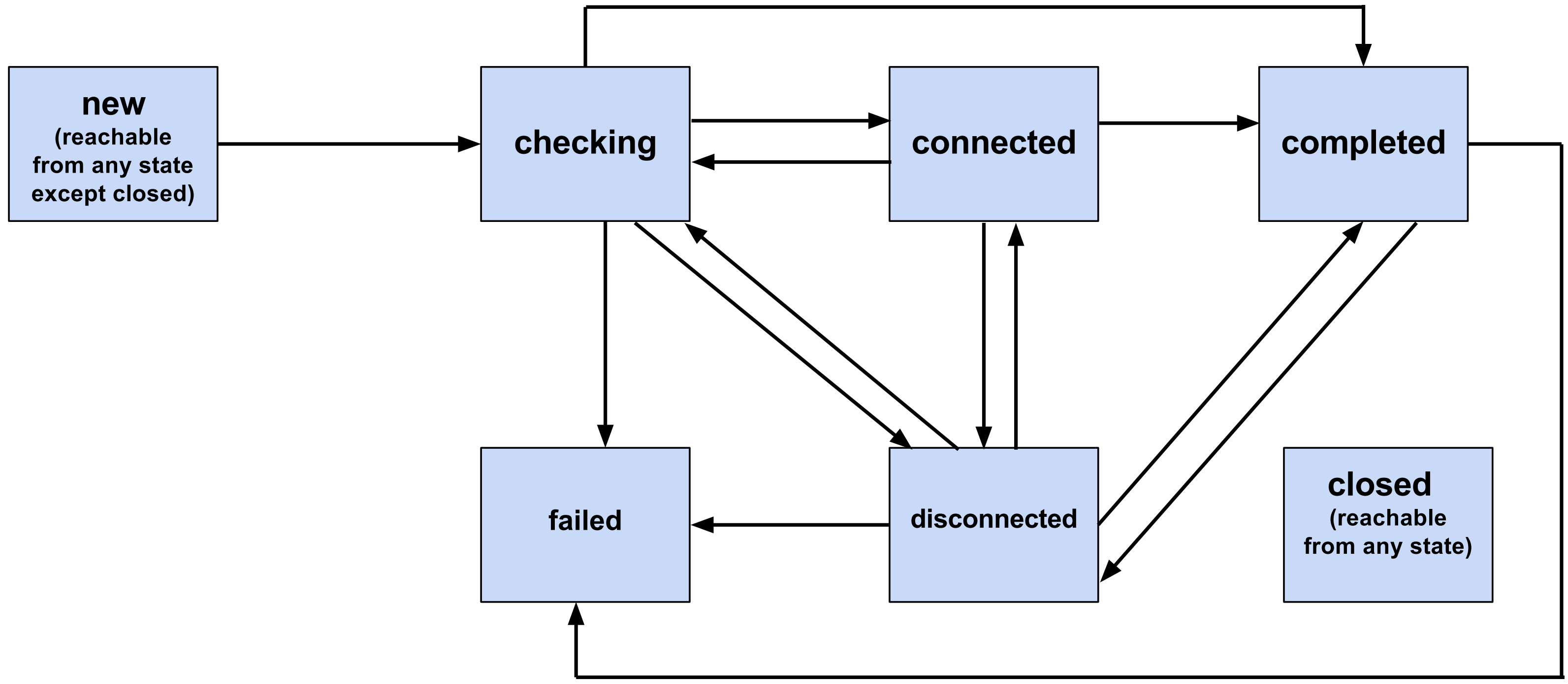


Figure 2 Non-normative ICE transport state transition diagram

5.6.5 `RTCIceRole` Enum

WebIDL

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

`RTCIceRole` Enumeration description	
`controlling`	A controlling agent as defined by [ICE], Section 3.
`controlled`	A controlled agent as defined by [ICE], Section 3.

5.6.6 `RTCIceComponent` Enum

WebIDL

```
enum RTCIceComponent {
    "rtp",
    "rtcp"
};
```

`RTCIceComponent` Enumeration description	
`rtp`	The ICE Transport is used for RTP (or RTCP multiplexing), as defined in [ICE], 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 [ICE], 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

```
[Constructor(DOMString type, RTCTrackEventInit eventInitDict),
 Exposed=Window]
interface RTCTrackEvent : Event {
    readonly attribute RTCRtpReceiver              receiver;
    readonly attribute MediaStreamTrack            track;
    [SameObject]
    readonly attribute FrozenArray<MediaStream> streams;
    readonly attribute RTCRtpTransceiver           transceiver;
};
```

Constructors

`RTCTrackEvent`

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 `MediaStream`s 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` attribute represents the `RTCRtpReceiver` object associated with the event.

`track` of type MediaStreamTrack, required

The `track` attribute represents the `MediaStreamTrack` object that is associated with the `RTCRtpReceiver` identified by `receiver`.

`streams` of type sequence<MediaStream>, defaulting to `[]`

The `streams` attribute returns an array of `MediaStream` objects representing the `MediaStream`s 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 WebSockets [WEBSOCKETS-API].

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*'s [[IsClosed]] slot is `true`, throw an `InvalidStateError`.
3. Create an `RTCDataChannel`, *channel*.
4. Initialize *channel*'s [[DataChannelLabel]] slot to the value of the first argument.
5. If [[DataChannelLabel]] is longer than 65535 bytes, throw a `TypeError`.
6. Let *options* be the second argument.

7. Initialize *channel*'s [[MaxPacketLifeTime]] slot to *option*'s `maxPacketLifeTime` member, if present, otherwise `null`.

8. Initialize *channel*'s [[MaxRetransmits]] slot to *option*'s `maxRetransmits` member, if present, otherwise `null`.

9. Initialize *channel*'s [[Ordered]] slot to *option*'s `ordered` member.

10. Initialize *channel*'s [[DataChannelProtocol]] slot to *option*'s `protocol` member.

11. If [[DataChannelProtocol]] is longer than 65535 bytes long, throw a `TypeError`.

12. Initialize *channel*'s [[Negotiated]] slot to *option*'s `negotiated` member.

13. Initialize *channel*'s [[DataChannelId]] slot to the value of *option*'s `id` member, 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 [RTCWEB-DATA-PROTOCOL].

14. If [[Negotiated]] is `true` and [[DataChannelId]] is `null`, throw a `TypeError`.

15. Initialize *channel*'s [[DataChannelPriority]] slot to *option*'s `priority` member.

16. If both [[MaxPacketLifeTime]] and [[MaxRetransmits]] attributes are set (not null), throw a `TypeError`.

17. 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.

18. 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`.

19. 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 [RTCWEB-DATA-PROTOCOL], 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 once the DTLS role is determined during the process of setting an `RTCSessionDescription`.***

20. Let *transport* be the *connection*'s [[SctpTransport]] slot.

    If the [[DataChannelId]] slot is not `null`, *transport* is in the `connected` state and [[DataChannelId]] is greater or equal to the *transport*'s [[MaxChannels]] slot, throw an `OperationError`.

21. If *channel* is the first `RTCDataChannel` created on *connection*, update the negotiation-needed flag for *connection*.

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 optional 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*, if provided, otherwise `"new"`.

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 [SCTP-SDP] (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. Fire an event named statechange at *transport*.
5. For each of *connection*'s RTCDataChannel:
    1. Let *channel* be the RTCDataChannel object.
    2. If the *channel*'s [[DataChannelId]] slot is greater or equal to *transport*'s [[MaxChannels]] slot, close the *channel* due to a failure . Otherwise, announce the

channel as open.

WebIDL

```
[Exposed=Window]
interface RTCSctpTransport {
    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 went 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"
};
```

Enumeration 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 or when 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.

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 [RTCWEB-DATA] and [RTCWEB-DATA-PROTOCOL].

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 `id`s.

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 [RTCWEB-DATA].

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.

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]]**, **[[DataChannelId]]**, and **[[DataChannelPriority]]**.

5. Return *channel*.

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*'s [[ReadyState]] is `closing` or `closed`, abort these steps.

4. Set *channel*'s [[ReadyState]] slot to `open`.

5. Fire an event named `open` at *channel*.

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. If the associated `RTCPeerConnection` object's [[IsClosed]] slot is `true`, abort these steps.

2. Create an `RTCDataChannel`, *channel*.

3. 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 [RTCWEB-DATA-PROTOCOL].

4. Initialize *channel*'s [[DataChannelLabel]], [[Ordered]], [[MaxPacketLifeTime]], [[MaxRetransmits]], [[DataChannelProtocol]], and [[DataChannelId]] internal slots to the corresponding values in *configuration*.

5. Initialize *channel*'s [[Negotiated]] internal slot to `false`.

6. Initialize *channel*'s [[DataChannelPriority]] internal slot based on the integer priority value in *configuration*, according to the following mapping:

configuration **priority value**	`RTCPriorityType` **value**
0 to 128	`very-low`
129 to 256	`low`
257 to 512	`medium`
513 and greater	`high`

7. Set *channel*'s [[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.***

8. Fire an event named `datachannel` using the `RTCDataChannelEvent` interface with the `channel` attribute set to *channel* at the `RTCPeerConnection` object.

9. Announce the data channel as open.

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 transport was closed.
2. Unless the procedure was initiated by the *channel*'s `close` method, set *channel*'s [[ReadyState]] slot to `closing`.
3. Run the following steps in parallel:
    1. Finish sending all currently pending messages of the *channel*.
    2. Follow the closing procedure defined for the *channel*'s underlying transport:
        1. In the case of an SCTP-based transport, follow [RTCWEB-DATA], section 6.7.
    3. Render the *channel*'s data transport closed by following the associated procedure.

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 transport was closed.
2. Set *channel*'s [[ReadyState]] slot to `closed`.
3. 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*.
4. Fire an event named `close` at *channel*.

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 [RTCWEB-DATA] 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*'s [[ReadyState]] slot 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*.

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. If *channel*'s [[ReadyState]] slot is not `open`, abort these steps and discard *rawData*.
3. Execute the sub step by switching on *type* and the *channel*'s `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.
4. Fire an event named `message` using the `MessageEvent` interface with its `origin` attribute initialized to the origin of the document that created the *channel*'s associated `RTCPeerConnection`, and the `data` attribute initialized to *data* at *channel*.

WebIDL

```
[Exposed=Window]
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 RTCPriorityType    priority;
    readonly attribute RTCDataChannelState readyState;
    readonly attribute unsigned long      bufferedAmount;
             attribute unsigned long
bufferedAmountLowThreshold;
             attribute EventHandler       onopen;
             attribute EventHandler       onbufferedamountlow;
             attribute EventHandler       onerror;
             attribute EventHandler       onclose;
    void close();
             attribute EventHandler       onmessage;
             attribute DOMString          binaryType;
    void send(USVString data);
    void send(Blob data);
    void send(ArrayBuffer data);
    void 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 other 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 initally 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 [RTCWEB-DATA-PROTOCOL]. 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.

`priority` of type *RTCPriorityType*, readonly

The `priority` attribute returns the priority for this `RTCDataChannel`. The priority is assigned by the user agent at channel creation time. On getting, the attribute *MUST* return the value of the [[DataChannelPriority]] 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.

`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 DOMString

The **`binaryType`** attribute *MUST*, on getting, return 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`".

This attribute controls how binary data is exposed to scripts. See the [WEBSOCKETS-API] for more information.

Methods

`close`

Closes the `RTCDataChannel`. It may be called regardless of whether the `RTCDataChannel` object was created by this peer or the remote peer.

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

1. Let *channel* be the `RTCDataChannel` object which is about to be closed.
2. If *channel*'s [[ReadyState]] slot is `closing` or `closed`, then abort these steps.
3. Set *channel*'s [[ReadyState]] slot 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.

WebIDL

```
dictionary RTCDataChannelInit {
    boolean          ordered = true;
    unsigned short   maxPacketLifeTime;
    unsigned short   maxRetransmits;
    USVString        protocol = "";
    boolean          negotiated = false;
    [EnforceRange]
    unsigned short   id;
    RTCPriorityType priority = "low";
};
```

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

Overrides the default selection of ID for this channel.

`priority` of type *RTCPriorityType*, defaulting to `low`

Priority of this channel.

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. If *channel*'s [[ReadyState]] slot is not `open`, throw an `InvalidStateError`.
3. 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.

- `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`.***

4. If the byte size of *data* exceeds the value of `maxMessageSize` on *channel*'s associated `RTCSctpTransport`, throw a `TypeError`.

5. 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`.***

6. Increase the value of the [[BufferedAmount]] slot by the byte size of *data*.

WebIDL

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

`RTCDataChannelState` Enumeration 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

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

Constructors

`RTCDataChannelEvent`

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, or `close` events.
- [[ReadyState]] slot is `open` and at least one event listener is registered for `message` events, `error` 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 [RTCWEB-AUDIO].

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 {
    void 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. If *transceiver*'s [[Stopped]] slot is `true`, throw an `InvalidStateError`.

4. If *transceiver*'s [[CurrentDirection]] slot is `recvonly` or `inactive`, throw an `InvalidStateError`.
5. Let *dtmf* be the `RTCDTMFSender` associated with *sender*.
6. If determine if DTMF can be sent for *dtmf* returns `false`, throw an `InvalidStateError`.
7. Let *tones* be the method's first argument.
8. If *tones* contains any unrecognized characters, throw an `InvalidCharacterError`.
9. Set the object's [[ToneBuffer]] slot to *tones*.
10. Set *dtmf*'s [[Duration]] slot to the value of the `duration` parameter.
11. Set *dtmf*'s [[InterToneGap]] slot to the value of the `interToneGap` parameter.
12. If the value of the `duration` parameter is less than 40 ms, set *dtmf*'s [[Duration]] slot to 40 ms.
13. If the value of the `duration` parameter is greater than 6000 ms, set *dtmf*'s [[Duration]] slot to 6000 ms.
14. If the value of the `interToneGap` parameter is less than 30 ms, set *dtmf*'s [[InterToneGap]] slot to 30 ms.
15. If the value of the `interToneGap` parameter is greater than 6000 ms, set *dtmf*'s [[InterToneGap]] slot to 6000 ms.
16. If [[ToneBuffer]] slot is an empty string, abort these steps.
17. If a *Playout task* is scheduled to be run, abort these steps; otherwise queue a task that runs the following steps (*Playout task*):
    1. If *transceiver*'s [[Stopped]] slot is `true`, abort these steps.
    2. If *transceiver*'s [[CurrentDirection]] slot is `recvonly` or `inactive`, abort these steps.
    3. 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.
    4. Remove the first character from the [[ToneBuffer]] slot and let that character be *tone*.
    5. 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 steps labelled *Playout task*.

6. 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 steps labelled *Playout task*.
7. 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

To **determine if DTMF can be sent** for an `RTCDTMFSender` instance *dtmfSender*, the user agent *MUST* queue a task that runs the 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. If *sender*'s [[SenderTrack]] is `null` return `false`.
6. If *transceiver*'s [[CurrentDirection]] is neither `"sendrecv"` nor `"sendonly"` return `false`.
7. If *sender*'s [[SendEncodings]]`[0].active` is `false` return `false`.
8. If no codec with mimetype `"audio/telephone-event"` has been negotiated for sending with this *sender*, return `false`.
9. Otherwise, return `true`.

7.4 `RTCDTMFToneChangeEvent`

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

WebIDL

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

Constructors

`RTCDTMFToneChangeEvent`

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 {
    required DOMString tone;
};
```

Dictionary `RTCDTMFToneChangeEventInit` Members

`tone` of type DOMString
: 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. For these objects, their lifetime ends when they are deleted by algorithms. At the time of deletion, a record of their statistics is emitted in a single `statsended` event containing an `RTCStatsReport` object with statistics from all objects deleted at the same time. 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);
    attribute EventHandler onstatsended;
};
```

Attributes

`onstatsended` of type EventHandler

The event type of this event handler is `statsended`.

To **delete stats** for a set of monitored objects associated with an `RTCPeerConnection`, *connection*, the UA *MUST* run the following steps in parallel:

1. Gather stats only for the set of monitored objects to be deleted. These stats *MUST* represent final values at the time of deletion. Stats for these monitored objects *MUST NOT* appear in subsequent calls to getStats().

2. Queue a task that runs the following steps:

    1. Let *report* be a new `RTCStatsReport` object.
    2. For each monitored object, create a new relevant stats object object with the stats gathered above for that monitored object, and add it to *report*.
    3. Fire an event named `statsended` using the `RTCStatsEvent` interface with the `report` attribute set to *report* at *connection*.

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` member 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>;
};
```

This interface has "entries", "forEach", "get", "has", "keys", "values", @@iterator methods and a "size" getter brought by `readonly maplike`.

Use these to retrieve the various dictionaries descended from `RTCStats` that this stats report is composed of. The set of supported property names [WEBIDL-1] 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
: The **`timestamp`**, of type `DOMHighResTimeStamp` [HIGHRES-TIME], associated with this object. The time is relative to the UNIX epoch (Jan 1, 1970, UTC). 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. User agents are free to pick any format for the id as long as it meets the requirements above.

The set of valid values for `RTCStatsType`, and the dictionaries derived from RTCStats that they indicate, are documented in [WEBRTC-STATS].

8.5 `RTCStatsEvent`

The `statsended` event uses the `RTCStatsEvent`.

WebIDL

```
[Constructor(DOMString type, RTCStatsEventInit eventInitDict),
 Exposed=Window]
interface RTCStatsEvent : Event {
    readonly attribute RTCStatsReport report;
};
```

Constructors

`RTCStatsEvent`

Attributes

`report` of type RTCStatsReport
: The **`report`** attribute contains the stats objects of the appropriate subclass of `RTCStats` object giving the value of the statistics for the monitored objects whose lifetime have ended, at the time that it ended.

WebIDL

```
dictionary RTCStatsEventInit : EventInit {
    required RTCStatsReport report;
};
```

Dictionary `RTCStatsEventInit` members

`report` of type *RTCStatsReport*, required
: Contains the `RTCStats` objects giving the stats for the objects whose lifetime have ended.

8.6 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*.

8.7 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 types when the corresponding objects exist on a PeerConnection, with the attributes that are listed when they are valid for that object:

- RTCRTPStreamStats, with attributes ssrc, kind, transportId, codecId, nackCount
- RTCReceivedRTPStreamStats, with all required attributes from its inherited dictionaries, and also attributes packetsReceived, packetsLost, jitter, packetsDiscarded
- RTCInboundRTPStreamStats, with all required attributes from its inherited dictionaries, and also attributes bytesReceived, trackId, receiverId, remoteId, framesDecoded
- RTCRemoteInboundRTPStreamStats, with all required attributes from its inherited dictionaries, and also attributes localId, roundTripTime
- RTCSentRTPStreamStats, with all required attributes from its inherited dictionaries, and also attributes packetsSent, bytesSent
- RTCOutboundRTPStreamStats, with all required attributes from its inherited dictionaries, and also attributes trackId, senderId, remoteId, framesEncoded
- RTCRemoteOutboundRTPStreamStats, with all required attributes from its inherited dictionaries, and also attributes localId, remoteTimestamp

- RTCPeerConnectionStats, with attributes dataChannelsOpened, dataChannelsClosed
- RTCDataChannelStats, with attributes label, protocol, datachannelId, state, messagesSent, bytesSent, messagesReceived, bytesReceived
- RTCMediaStreamStats, with attributes streamIdentifer, trackIds
- RTCMediaStreamTrackStats, with attribute detached
- RTCMediaHandlerStats with attributes trackIdentifier, remoteSource, ended
- RTCAudioHandlerStats with attribute audioLevel
- RTCVideoHandlerStats with attributes frameWidth, frameHeight, framesPerSecond
- RTCVideoSenderStats with attribute framesSent
- RTCVideoReceiverStats with attributes framesReceived, framesDecoded, framesDropped, framesCorrupted
- RTCCodecStats, with attributes payloadType, codec, clockRate, channels, sdpFmtpLine
- RTCTransportStats, with attributes bytesSent, bytesReceived, rtcpTransportStatsId, selectedCandidatePairId, localCertificateId, remoteCertificateId
- RTCIceCandidatePairStats, with attributes transportId, localCandidateId, remoteCandidateId, state, priority, nominated, bytesSent, bytesReceived, totalRoundTripTime, currentRoundTripTime
- RTCIceCandidateStats, with attributes address, port, protocol, candidateType, url
- RTCCertificateStats, with attributes fingerprint, fingerprintAlgorithm, base64Certificate, issuerCertificateId

An implementation *MAY* support generating any other statistic defined in [WEBRTC-STATS], and *MAY* generate statistics that are not documented.

8.8 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 9

```
async function gatherStats() {
  try {
    const sender = pc.getSenders()[0];
    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 (let 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) {
        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 MediaStreamTracks 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 [RTCWEB-RTP], [RTCWEB-AUDIO], and [RTCWEB-TRANSPORT].

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 MediaStreamTracks.

Channels are the smallest unit considered in the MediaStream 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, 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 basics of `MediaTrackSupportedConstraints`, `MediaTrackCapabilites`, `MediaTrackConstraints` and `MediaTrackSettings` is outlined in [GETUSERMEDIA]. However, the `MediaTrackSettings` for a `MediaStreamTrack` sourced by an `RTCPeerConnection` will only be populated with members to the extent that data is supplied by means of the remote `RTCSessionDescription` applied via `setRemoteDescription` and the actual RTP data. This means that certain members, such as `facingMode`, `echoCancellation`, `latency`, `deviceId` and `groupId`, will always be missing.

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 10

```
const signaling = new SignalingChannel(); // handles
JSON.stringify/parse
const constraints = {audio: true, video: true};
const configuration = {iceServers: [{urls:
'stuns: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(await pc.createOffer());
    // send the offer to the other peer
    signaling.send({desc: pc.localDescription});
  } catch (err) {
    console.error(err);
  }
};

// once media for a remote track arrives, show it in the remote
video element
pc.ontrack = (event) => {
  // don't set srcObject again if it is already set.
  if (remoteView.srcObject) return;
  remoteView.srcObject = event.streams[0];
};

// call start() to initiate
async function start() {
  try {
    // get a local stream, show it in a self-view and add it to be
sent
    const stream = await
navigator.mediaDevices.getUserMedia(constraints);
    stream.getTracks().forEach((track) => pc.addTrack(track,
stream));
    selfView.srcObject = stream;
  } catch (err) {
    console.error(err);
  }
}

signaling.onmessage = async ({desc, candidate}) => {
  try {
```

```
      if (desc) {
        // if we get an offer, we need to reply with an answer
        if (desc.type == 'offer') {
          await pc.setRemoteDescription(desc);
          const stream = await
navigator.mediaDevices.getUserMedia(constraints);
          stream.getTracks().forEach((track) => pc.addTrack(track,
stream));
          await pc.setLocalDescription(await pc.createAnswer());
          signaling.send({desc: pc.localDescription});
        } else if (desc.type == 'answer') {
          await pc.setRemoteDescription(desc);
        } else {
          console.log('Unsupported SDP type. Your code may differ
here.');
        }
      } 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 11

```
const signaling = new SignalingChannel();
const configuration = {iceServers: [{urls:
'stuns:stun.example.org'}]};
const audio = null;
const audioSendTrack = null;
const video = null;
const videoSendTrack = null;
const started = false;
let pc;

// Call warmup() to warm-up ICE, DTLS, and media, but not send
```

```
media yet.
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 = (event) => {
    signaling.send(JSON.stringify({candidate: event.candidate}));
  };

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
    try {
      await pc.setLocalDescription(await pc.createOffer());
      // send the offer to the other peer
      signaling.send(JSON.stringify({desc: pc.localDescription}));
    } catch (err) {
      console.error(err);
    }
  };

  // once media for the remote track arrives, show it in the
remote video element
  pc.ontrack = async (event) => {
    try {
      if (event.track.kind == 'audio') {
        if (isAnswerer) {
          audio = event.transceiver;
          audio.direction = 'sendrecv';
          if (started && audioSendTrack) {
            await audio.sender.replaceTrack(audioSendTrack);
          }
        }
      } else if (event.track.kind == 'video') {
        if (isAnswerer) {
          video = event.transceiver;
          video.direction = 'sendrecv';
          if (started && videoSendTrack) {
            await video.sender.replaceTrack(videoSendTrack);
          }
        }
      }
```

```
      // don't set srcObject again if it is already set.
      if (remoteView.srcObject) return;
      remoteView.srcObject = event.streams[0];
    } 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({audio: true, video: true});
    selfView.srcObject = stream;
    audioSendTrack = stream.getAudioTracks()[0];
    if (started) {
      await audio.sender.replaceTrack(audioSendTrack);
    }
    videoSendTrack = stream.getVideoTracks()[0];
    if (started) {
      await video.sender.replaceTrack(videoSendTrack);
    }
  } catch (err) {
    console.erro(err);
  }
}

// Call start() to start sending media.
function start() {
  started = true;
  signaling.send(JSON.stringify({start: true}));
}

signaling.onmessage = async (event) => {
  if (!pc) warmup(true);

  try {
    const message = JSON.parse(event.data);
    if (message.desc) {
      const desc = message.desc;

      // if we get an offer, we need to reply with an answer
      if (desc.type == 'offer') {
        await pc.setRemoteDescription(desc);
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send(JSON.stringify({desc:
```

```
pc.localDescription}));
        } else {
          await pc.setRemoteDescription(desc);
        }
      } else if (message.start) {
        started = true;
        if (audio && audioSendTrack) {
          await audio.sender.replaceTrack(audioSendTrack);
        }
        if (video && videoSendTrack) {
          await video.sender.replaceTrack(videoSendTrack);
        }
      } else {
        await pc.addIceCandidate(message.candidate);
      }
    } catch (err) {
      console.error(err);
    }
  };
```

10.3 Peer-to-peer Example with media before signaling

The answerer may wish to send media in parallel with sending the answer, and the offerer may wish to render the media before the answer arrives.

EXAMPLE 12

```
const signaling = new SignalingChannel();
const configuration = {iceServers: [{urls:
'stuns:stun.example.org'}]};
let pc;

// call start() to initiate
async function start() {
  pc = new RTCPeerConnection(configuration);

  // send any ice candidates to the other peer
  pc.onicecandidate = (event) => {
    signaling.send(JSON.stringify({candidate: event.candidate}));
  };

  // let the "negotiationneeded" event trigger offer generation
  pc.onnegotiationneeded = async () => {
```

```
    try {
      await pc.setLocalDescription(await pc.createOffer());
      // send the offer to the other peer
      signaling.send(JSON.stringify({desc: 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({audio: true, video: true});
    selfView.srcObject = stream;
    // Render the media even before ontrack fires.
    remoteView.srcObject = new
MediaStream(pc.getReceivers().map((r) => r.track));
  } catch (err) {
    console.error(err);
  }
};

signaling.onmessage = async (event) => {
  if (!pc) start();

  try {
    const message = JSON.parse(event.data);
    if (message.desc) {
      const desc = message.desc;

      // if we get an offer, we need to reply with an answer
      if (desc.type == 'offer') {
        await pc.setRemoteDescription(desc);
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send(JSON.stringify({desc:
pc.localDescription}));
      } else {
        await pc.setRemoteDescription(desc);
      }
    } else {
      await pc.addIceCandidate(message.candidate);
    }
  } catch (err) {
    console.error(err);
```

```
  }
};
```

10.4 Simulcast Example

A client wants to send multiple RTP encodings (simulcast) to a server.

EXAMPLE 13

```
const signaling = new SignalingChannel();
const configuration = {'iceServers': [{'urls':
'stuns: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(await pc.createOffer());
      // send the offer to the other peer
      signaling.send(JSON.stringify({desc: 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({audio: true, video: true});
    selfView.srcObject = stream;
    pc.addTransceiver(stream.getAudioTracks()[0], {direction:
'sendonly'});
    pc.addTransceiver(stream.getVideoTracks()[0], {
      direction: 'sendonly',
      sendEncodings: [
        {rid: 'f'},
        {rid: 'h', scaleResolutionDownBy: 2.0},
        {rid: 'q', scaleResolutionDownBy: 4.0}
```

```
      ]
    });
  } catch (err) {
    console.error(err);
  }
}

signaling.onmessage = async (event) => {
  try {
    const message = JSON.parse(event.data);
    if (message.desc) {
      await pc.setRemoteDescription(message.desc);
    } else {
      await pc.addIceCandidate(message.candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

10.5 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 and listeners are attached to monitor when the channel is ready, messages are received and when the channel is closed.

EXAMPLE 14

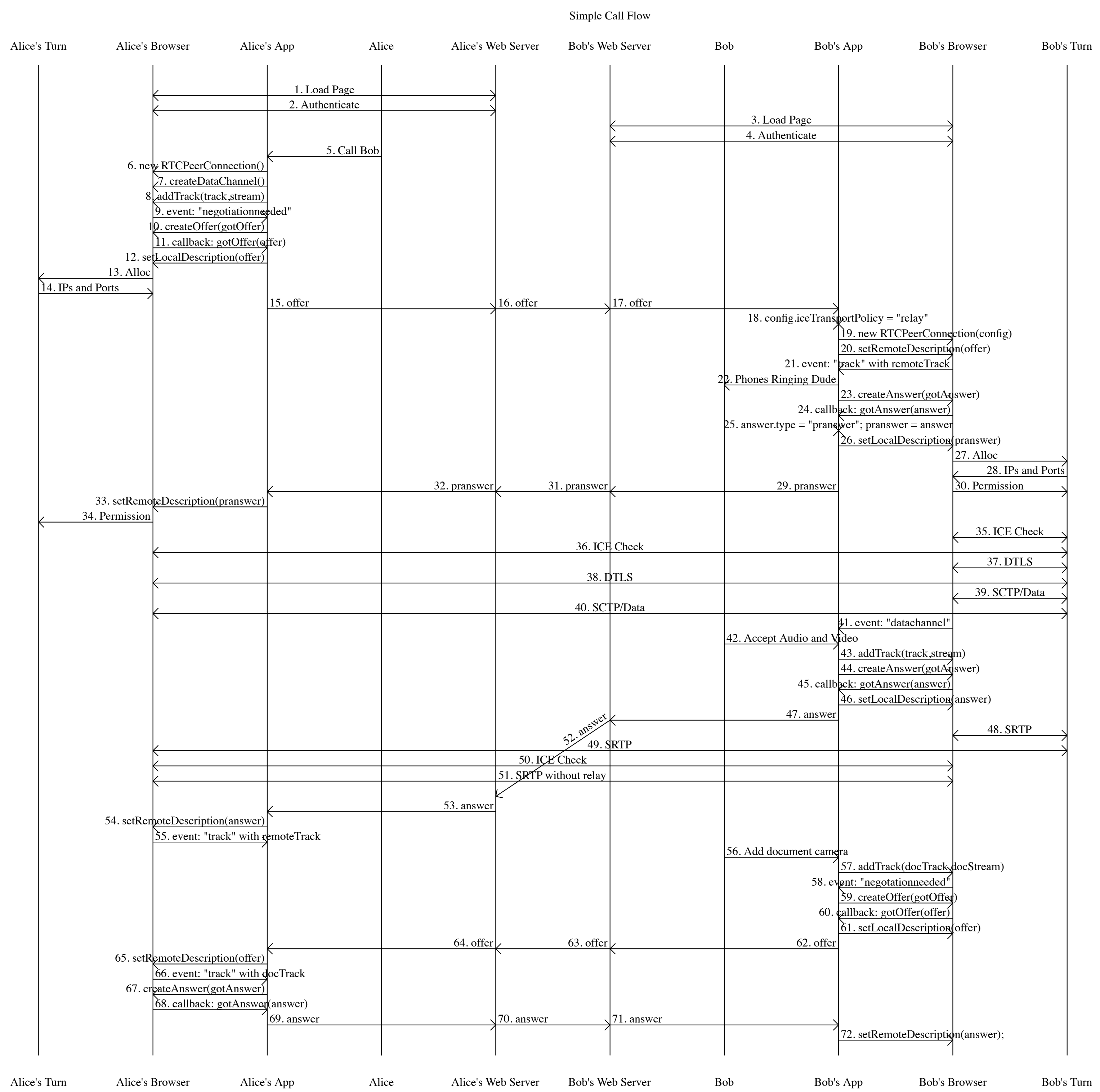
```
const signaling = new SignalingChannel(); // handles
JSON.stringify/parse
const configuration = {iceServers: [{urls:
'stuns:stun.example.org'}]};
let pc;
let channel;

// call start(true) to initiate
function start(isInitiator) {
  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(await pc.createOffer());
        // send the offer to the other peer
        signaling.send({desc: pc.localDescription});
      } catch (err) {
        console.error(err);
      }
    };

    if (isInitiator) {
      // create data channel and setup chat
      channel = pc.createDataChannel('chat');
      setupChat();
    } else {
      // setup chat on incoming data channel
      pc.ondatachannel = (event) => {
        channel = event.channel;
        setupChat();
      };
    }
  }

  signaling.onmessage = async ({desc, candidate}) => {
    if (!pc) start(false);

    try {
      if (desc) {
        // if we get an offer, we need to reply with an answer
        if (desc.type == 'offer') {
          await pc.setRemoteDescription(desc);
          await pc.setLocalDescription(await pc.createAnswer());
          signaling.send({desc: pc.localDescription});
        } else {
          await pc.setRemoteDescription(desc);
        }
      } else {
        await pc.addIceCandidate(candidate);
      }
    } catch (err) {
      console.error(err);
    }
  };
```

```
function setupChat() {
  // e.g. enable send button
  channel.onopen = () => enableChat(channel);
  channel.onmessage = (event) => showChatMessage(event.data);
}
```

10.6 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.



10.7 DTMF Example

Examples assume that *sender* is an `RTCRtpSender`.

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

EXAMPLE 15

```
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 16

```
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 17

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve,
ms));

if (sender.dtmf.canInsertDTMF) {
  const duration = 500;
  sender.dtmf.insertDTMF(sender.dtmf.toneBuffer + '1234',
duration);
  sender.dtmf.ontonechange = async (event) => {
    if (!event.tone) return;
    lightKey(event.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 18

```
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 = (event) => {
    if (event.tone == '1') {
      // append more tones when playout has begun
      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 19

```
if (sender.dtmf.canInsertDTMF) {
  sender.dtmf.ontonechange = (event) => {
    if (event.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');
}
```

11. Error Handling

This section and its subsections extend the list of Error subclasses defined in [ECMASCRIPT-6.0] following the pattern for NativeError in section 19.5.6 of that specification. Assume the following:

- that use of syntax such as [[Something]] and %something% is as used in [ECMASCRIPT-6.0].
- that the rules for ECMAScript standard built-in objects ([ECMASCRIPT-6.0], section 17) are in effect in this section.
- that the new intrinsic objects `%RTCError%` and `%RTCErrorPrototype%` are available as if they had been included in ([ECMASCRIPT-6.0], Table 7) and all referencing sections, e.g. ([ECMASCRIPT-6.0], section 8.2.2), thus behave appropriately.

11.1 ECMAScript 6 Terminology

The following terms used in this section are defined in [ECMASCRIPT-6.0].

Term/Notation	Section in [ECMASCRIPT-6.0]
Type(X)	6
intrinsic object	6.1.7.4
[[ErrorData]]	19.5.1
internal slot	6.1.7.2
NewTarget	various uses, but no definition

active function object	8.3
OrdinaryCreateFromConstructor()	9.1.14
ReturnIfAbrupt()	6.2.2.4
Assert	5.2
String	4.3.17-19, depending on context
PropertyDescriptor	6.2.4
[[Value]]	6.1.7.1
[[Writable]]	6.1.7.1
[[Enumerable]]	6.1.7.1
[[Configurable]]	6.1.7.1
DefinePropertyOrThrow()	7.3.7
abrupt completion	6.2.2
ToString()	7.1.12
[[Prototype]]	9.1
%Error%	19.5.1
Error	19.5
%ErrorPrototype%	19.5.3
Object.prototype.toString	19.1.3.6

11.2 **RTCError** Object

11.2.1 RTCError Constructor

The RTCError Constructor is the `%RTCError%` intrinsic object. When `RTCError` is called as a function rather than as a constructor, it creates and initializes a new `RTCError` object. A call of the object as a function is equivalent to calling it as a constructor with the same arguments. Thus the function call **`RTCError(...)`** is equivalent to the object creation expression **`new RTCError(...)`** with the same arguments.

The `RTCError` constructor is designed to be subclassable. It may be used as the value of an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified `RTCError` behaviour must include a `super` call to the `RTCError` constructor to create and initialize the subclass instance with an [[ErrorData]] internal slot.

11.2.1.1 RTCErrorDetailType Enum

WebIDL

```
enum RTCErrorDetailType {
    "data-channel-failure",
    "dtls-failure",
    "fingerprint-failure",
    "idp-bad-script-failure",
    "idp-execution-failure",
    "idp-load-failure",
    "idp-need-login",
    "idp-timeout",
    "idp-tls-failure",
    "idp-token-expired",
    "idp-token-invalid",
    "sctp-failure",
    "sdp-syntax-error",
    "hardware-encoder-not-available",
    "hardware-encoder-error"
};
```

Enumeration 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".
`idp-bad-script-failure`	The script loaded from the identity provider is not valid JavaScript or did not implement the correct interfaces.
`idp-execution-`	The identity provider has thrown an exception or returned a rejected promise.

failure	
idp-load-failure	Loading of the IdP URI has failed. The httpRequestStatusCode attribute is set to the HTTP status code of the response.
idp-need-login	The identity provider requires the user to login. The idpLoginUrl attribute is set to the URL that can be used to login.
idp-timeout	The IdP timer has expired.
idp-tls-failure	The TLS certificate used for the IdP HTTPS connection is not trusted.
idp-token-expired	The IdP token has expired.
idp-token-invalid	The IdP token is invalid.
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.

11.2.1.2 RTCError (errorDetail, message)

When the RTCError function is called with arguments *errorDetail* and *message* the following steps are taken:

1. If NewTarget is **undefined**, let *newTarget* be the active function object, else let *newTarget* be NewTarget.
2. Let *O* be OrdinaryCreateFromConstructor(*newTarget*, "%RTCErrorPrototype%", «[[ErrorData]]»).
3. ReturnIfAbrupt(*O*).
4. If *errorDetail* is not **undefined**, then

    1. Let *errorDetailDesc* be the PropertyDescriptor{[[Value]]: *errorDetail*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
    2. Let *cStatus* be DefinePropertyOrThrow(O, "`errorDetail`", *errorDetailDesc*).
    3. Assert: *eStatus* is not an abrupt completion.
5. If *message* is not **undefined**, then
    1. Let *msg* be ToString(*message*).
    2. Let *msgDesc* be the PropertyDescriptor{[[Value]]: *msg*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}.
    3. Let *mStatus* be DefinePropertyOrThrow(O, "`message`", *msgDesc*).
    4. Assert: *mStatus* is not an abrupt completion.
6. Return *O*.

11.2.2 Properties of the RTCError Constructor

The value of the [[Prototype]] internal slot of the `RTCError` constructor is the intrinsic object `%Error%`.

Besides the `length` property (whose value is **1**), the `RTCError` constructor has the following properties:

11.2.2.1 RTCError.prototype

The initial value of `RTCError.prototype` is the RTCError prototype object. This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

11.2.3 Properties of the RTCError Prototype Object

The `RTCError` prototype object is an ordinary object. It is not an Error instance and does not have an [[ErrorData]] internal slot.

The value of the [[Prototype]] internal slot of the `RTCError` prototype object is the intrinsic object `%ErrorPrototype%`.

11.2.3.1 RTCError.prototype.constructor

The initial value of the `constructor` property of the prototype for the `RTCError` constructor is the intrinsic object `%RTCError%`.

11.2.3.2 RTCError.prototype.errorDetail

The initial value of the **`errorDetail`** property of the prototype for the `RTCError` constructor is the empty String.

11.2.3.3 RTCError.prototype.sdpLineNumber

The initial value of the **`sdpLineNumber`** property of the prototype for the `RTCError` constructor is 0.

11.2.3.4 RTCError.prototype.httpRequestStatusCode

The initial value of the `httpRequestStatusCode` property of the prototype for the `RTCError` constructor is 0.

11.2.3.5 RTCError.prototype.sctpCauseCode

The initial value of the `sctpCauseCode` property of the prototype for the `RTCError` constructor is 0.

11.2.3.6 RTCError.prototype.receivedAlert

An unsigned integer representing the value of the DTLS alert received. The initial value of the **`receivedAlert`** property of the prototype for the `RTCError` constructor is null.

11.2.3.7 RTCError.prototype.sentAlert

An unsigned integer representing the value of the DTLS alert sent. The initial value of the **`sentAlert`** property of the prototype for the `RTCError` constructor is null.

11.2.3.8 RTCError.prototype.message

The initial value of the `message` property of the prototype for the `RTCError` constructor is the empty String.

11.2.3.9 RTCError.prototype.name

The initial value of the `name` property of the prototype for the `RTCError` constructor is **`"RTCError"`**.

11.2.4 Properties of RTCError Instances

`RTCError` instances are ordinary objects that inherit properties from the `RTCError` prototype object and have an [[ErrorData]] internal slot whose value is **undefined**. The only specified use of [[ErrorData]] is by Object.prototype.toString ([ECMASCRIPT-6.0], section 19.1.3.6) to identify instances of Error or its various subclasses.

The **`RTCErrorEvent`** interface is defined for cases when an RTCError is raised as an event:

WebIDL

```
[Exposed=Window,
 Constructor(DOMString type, RTCErrorEventInit eventInitDict)]
interface RTCErrorEvent : Event {
    readonly attribute RTCError? error;
};
```

Constructors

`RTCErrorEvent`
: Constructs a new `RTCErrorEvent`.

Attributes

`error` of type RTCError, readonly, nullable
: The `RTCError` describing the error that triggered the event (if any).

WebIDL

```
dictionary RTCErrorEventInit : EventInit {
    RTCError? error = null;
};
```

Dictionary `RTCErrorEventInit` Members

`error` of type RTCError, nullable, defaulting to `null`
    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`** [webmessaging]	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.
`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 signaling state has changed. This state change is the result of either setLocalDescription or setRemoteDescription being invoked.
iceconnectionstatechange	Event	The RTCPeerConnection's ICE connection state has changed.
icegatheringstatechange	Event	The RTCPeerConnection's ICE gathering state has changed.
icecandidate	RTCPeerConnectionIceEvent	A new RTCIceCandidate is made available to the script.
connectionstatechange	Event	The RTCPeerConnection connectionState 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.
isolationchange	Event	A new Event is dispatched to the script when the *isolated* attribute on a MediaStreamTrack changes.
statsended	RTCStatsEvent	A new RTCStatsEvent is dispatched to the script in response to one or more monitored objects being deleted at the same time.

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-error" 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 [RTCWEB-SECURITY-ARCH].

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.

The peerIdentity mechanism loads and executes JavaScript code from a third-party server acting as an identity provider. That code is executed in a separate JavaScript realm and does not affect the protections afforded by the same origin policy.

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 [RTCWEB-IP-HANDLING] 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.

A mechanism, `peerIdentity`, is provided that gives Javascript the option of requesting media that the same javascript cannot access, but can only be sent to certain other entities.

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.

If set, the configured default ICE servers exposed by `getDefaultIceServers` on `RTCPeerConnection` instances also provides persistent across time and origins information which increases the fingerprinting surface of a given browser.

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.

14. Change Log

This section will be removed before publication.

Changes since October 02, 2017

1. [#1616] Add Duration, InterToneGap, ToneBuffer and CanInsertDTMF internal slots of RTCDTMFSender
2. [#1614] createDataChannel: Update negotiation needed state on the main thread
3. [#1553] Add API to expose what algorithms the browser supports
4. [#1430] Have createOffer call addTransceiver() on offerToReceive
5. [#1615] setParameters: Use more specific hardware encoder errors
6. [#1621] RTCPeerConnection.close should close Data Channels and SctpTransports
7. [#1538] IdP protocol can only contain characters legal in a URI
8. [#1620] Add RTCSctpTransportState
9. [#1634] Add onstatechange event to SctpTransport Event table
10. [#1631] Explain the scope of DTLS/ICE transport objects in more detail
11. [#1623] Describe how transport objects are assigned to senders/receivers
12. [#1636] Simple text for scaling issue - no letterbox allowed
13. [#1641] Note about video dimension adaptation discussion added

Changes since August 22, 2017

1. [#1559] Reference rtcweb-data-channel for RTCPriorityType enum
2. [#1557] Make fields in RTCStats dictionary required
3. [#1556] Update mandatory to implement stats fields to sync with webrtc-stats
4. [#1555] Fix reference to validating assertion result in requesting assertions
5. [#1551] Clarify the meaning of session description sdp need not match
6. [#1550] Validate binaryType value when setting it in RTCDataChannel
7. [#1549] Allow createAnswer to be called only in valid signaling state
8. [#1547] Wait for certificate to be generated before identity assertion process
9. [#1544] Align stats example with WebRTC stast spec (s/outboundrtp/outbound-rtp)
10. [#1539] Remove unnecessary type checking for selectorArg in getStats
11. [#1536] Define vhen dtmf attribute is set
12. [#1525] Update paragraph that introduces senders/receivers/transceivers
13. [#1541] Specify getCapabilities behavior with an unsupported value of kind
14. [#1487] Check for invalid rollback

15. [#1522] Formalize how createOffer interacts with identity providers
16. [#1558] Throw if a DataChannel, to be negotiated by the script, lacks id
17. [#1552] Harmonize and update our references to other specifications
18. [#1443] SLD/SRD: Check if transceiver is stopped before setting currentDirection
19. [#1548] Add note that IdP must treat contents as opaque
20. [#1561] Clarify which session description to check for if negotiation is needed
21. [#1566] Add a section summarizing different ICE candidate events
22. [#1580] Add ICE candidates only to the applicable descriptions.
23. [#1572] Annotate all interfaces with Exposed extended attribute
24. [#1571] Remove unreferenced [HTML] ref using respec post processing
25. [#1596] Add Promise terms to Terminology section
26. [#1595] Add note that null candidate is for legacy compatibility
27. [#1592] Select sent codec via "codecPayloadType" field rather than reordering
28. [#1591] Add some text about how the AssociatedMediaStreams internal slot is used
29. [#1590] Use internal slots for transceiver's sender/receiver and receiver's track
30. [#1585] Use internal slot for RTCRtpSender's track
31. [#1604] Add 'resolved' (and variants) to Promise terminology
32. [#1582] RTCIceTransport: Use internal slots
33. [#1577] scaleResolutionDownBy: Specify how the User Agent should behave when scaling video
34. [#1607] Update DTMF examples to match specified behavior
35. [#1581] RTCDtlsTransport: Use internal slots
36. [#1560] setParameters: Do argument checks in sync section and specify parellel steps

Changes since June 05, 2017

1. [#1160] Remove getAlgorithm()
2. [#1298] Specify DTMF intertone gap maximum
3. [#1327] Remove fingerprint matching.
4. [#1329] Update maxBitrate definition
5. [#1337] Fix DTMF Examples (Section 11.7)

6. [#1348] Add a note on the absence of privacy impact of configured default ICE
7. [#1349] Show RFC2119 keywords in small-caps (was broken by respec update)
8. [#1350] Remove meaningless case-sensitive qualification of RID characters
9. [#1359] createOffer/Answer: Remove sentence with vague 'reasonably soon'
10. [#1356] createDataChannel: Use TypeError for bad reliability arguments
11. [#1340] Section 4.2.5/4.2.6: Enum Table Inconsistency
12. [#1115] Specify DTLS failures in more detail
13. [#1168] Remove paragraph about removeTrack causing track to be ended remotely
14. [#1209] Throw error if data channel's buffer is filled, rather than closing
15. [#1229] Add detailed steps for constructing RTCIceCandidate
16. [#1321] Start integrating direction into 'create RTCRtpTransceiver' algorithm
17. [#1333] Algorithm for rejecting modification
18. [#1334] createOffer/Answer: Specify built-in certificate behavior
19. [#1338] Clarification: insertDTMF replaces the current tone buffer
20. [#1339] Fill in empty attribute descriptions in ice description
21. [#1358] RTCDataChannel: Use internal slots and specify default values at one location
22. [#1385] Update RTCTrackEvent text
23. [#1388] Make replaceTrack accept null argument
24. [#1373] Specify DTMF playout algorithm for comma
25. [#1392] Add reference to RFC 5245
26. [#1436] RTCRtpTransceiver: Set currentDirection to null when stopping
27. [#1433] Add NotSupportedError for unknown ICE server schema
28. [#1432] addTranscevier: Assume default dictionary argument
29. [#1429] ice-tcp: add note about ice-tcp types a UA will gather
30. [#1428] rtcsessiondescription: attributes are not mutable
31. [#1426] getstats: improve selector definiton
32. [#1421] Clarify enqueue is acted on specific connection's operation queue
33. [#1409] Use true and false instead of "true" and "false"
34. [#1405] Remove "cannot be applied at the media layer"
35. [#1404] In data channel send(), remove unneeded conditions

36. [#1455] Specify data channel label/protocol length restriction
37. [#1453] Removing text talking about key shortening that was incorrect
38. [#1440] Set CurrentDirection slot for provisional answers
39. [#1399] Add reference to JSEP that setLocalDescription triggers ICE gathering
40. [#1449] Replace serializers by toJSON definitions
41. [#1451] Add legacy note about addStream
42. [#1457] Run ice server configuration validation steps for each url
43. [#1480] Update mandatory stats to reflect rtp refactor in webrtc-stats
44. [#1402] Add/remove remote tracks from msid streams based on direction
45. [#1514] Specify that setLocalDescription() with null sdp is not applicable
46. [#1434] Don't proceed w/removeTrack() if sender.track is already null
47. [#1531] Remove SSRCs from RTCRtpEncodingParameters
48. [#1530] Adding defaults for RTCRtpEncodingParameters.active and priority
49. [#1528] Ignore RTCDataChannelInit.id if "negotiated" is false
50. [#1527] Define what an "associated" transceiver is
51. [#1526] Clarify that ICE states should be "new" if there are no transports
52. [#1524] Reference Direction internal slot from addTrack/removeTrack
53. [#1523] Specify how data channel priority enum is initialized from priority integer
54. [#1521] Making getParameters/setParameters matching logic more deterministic
55. [#1534] Change ResourceInUse to OperationError

Changes since May 15, 2017

1. [#1153] Constructor for RTCIceCandidate should accept optional argument
2. [#1203] Invalid RTCRtpTransceiverDirection already throws TypeError.
3. [#1221 Introduction: increase specification scope to general p2p
4. [#1134] Add more detail about how getParameters and setParameters work
5. [#1170] RTCIceCandidate: add component attribute
6. [#1225] Units for maxFramerate
7. [#1226] Removing WebIDL defaults for various RTP parameters
8. [#1239] RTCIceConnectionEventInit: url is nullable

9. [#1220] Reorder createOffer/createAnswer paragraphs
10. [#1252] Remove note about identity is at risk
11. [#1320] Clarify "trusted" origins as whitelisted

Changes since May 08, 2017

1. [#1149] Add paragraph about RtpContributingSources being updated simultaneously
2. [#1172] Adding note about legacy `createAnswer` not supporting options dict
3. [#1175] Expanding RTCPeerConnection introduction
4. [#1176] Adding more detail to RTCIceTransportPolicy enum descriptions
5. [#1180] Change maxFramerate type from unsigned long to double

Changes since March 03, 2017

1. [#1033] "Hybrid" OAuth solution
2. [#1067] Mark getAlgorithm method at risk
3. [#1069] Freeing resources for incoming stream
4. [#1067] Add getStats() to RTCRtpSender/Receiver
5. [#1081] Clarify which "candidate" is referred to in addIceCandidate description
6. [#1071] Specify behavior if browser doesn't implement "negotiate" rtcpMuxPolicy
7. [#1087] Update call flow in Section 11.6
8. [#1088] Make "candidate" non-nullable in addIceCandidate parameter table
9. [#1094] Check RTCPeerConnection isClosed slot before running queued tasks
10. [#1082] Handling RTX in RTCRtpCodecCapabilities
11. [#1100] Clarify when RTCRtpContributingSource.audioLevel can be null
12. [#1097] Mark RTP/RTCP non-mux feature at risk
13. [#1011] Eliminate NetworkError
14. [#1109] Adding configurable "ptime" member of RTCRtpEncodingParameters
15. [#1099] Always update the RTCRtpContributingSource for SSRCs
16. [#1104] Add missing "closed" signaling state
17. [#1107] Section 12.2.1.1: RTCErrorDetailType Enum definition

18. [#1098] Attempt to update RTCRtpContributingSource objects at playout time
19. [#1114] Mark Identity as a "feature at risk"
20. [#1119] Making legacy methods optional to implement
21. [#1122] RTCCertificate.getAlgorithm() to return a compatible AlgorithmIdentifier
22. [#1130] Clarify that configuration.certificates remains undefined in the RTCPeerConnection constructor
23. [#1131] RTCPeerConnection.createDataChannel: Drop [TreatNullAs=EmptyString] for USVString
24. [#1129 Code examples: dont fiddle with srcObject if already set
25. [#1136] Fire the "track" event from a queued task
26. [#1137] Adding more detail about RTCDataChannel.id's default value (null)
27. [#1139] Call RTCDtlsFingerprint a dictionary, not an object
28. [#1140] Add a link to web-platform-tests to the top of the spec
29. [#1133] Split getContributingSources into two methods, for CSRCs and SSRCs
30. [#1145] FrozenArray, sequence and SameObject (Use sequence and getters instead of FrozenArray for getFingerPrints and getDefaultIceServers) (Use SameObject for RTCTrackEvent.streams)
31. [#1147] Add reference to ICE restart

Changes since December 19, 2016

1. [#985] Removed legacy getStats() method
2. [#982] Specify unit for maxPacketLifeTime
3. [#987] Make the ufrag optional in RTCIceCandidateInit, for backwards compat.
4. [#993] Use lowercase values for RTCIceComponent
5. [#994] Changing "non-null" to "missing" to match IDL terminology.
6. [#996] Describe when an RTCSctpTransport is created/set to null.
7. [#999] Make transceiver.stop() send a BYE
8. [#1002] Dispatch event when a transceiver is stopped via remote action
9. [#1003] Change setLocalDescription to require unchanged offer/answer string
10. [#1004] Specify that currentRemoteDescription.sdp need not match remoteDescription.sdp
11. [#1006] Make errorCode required in RTCPeerConnectionIceErrorEventInit

12. [#1005] Add offerToReceive* as legacy extensions
13. [#1001] Specify the effect of a BYE on RtpReceiver.track
14. [#1015] Fix inconsistencies in description of RTCDTMFToneChangeEvent.tone
15. [#1016] Ensure that "track" event is only fired for "send" direction m-sections.
16. [#1018] Mark negotitate in RTCRtcpMuxPolicy at risk
17. [#1029] Add IdP token expired error
18. [#1028] Add IdP invalid token error
19. [#1027] Add string for extra info about idpErrors
20. [#1019] Clarify that it is possible to send the same track in several copies
21. [#1023] Specify how media is centered, cropped, and scaled
22. [#1025] Mention that codecs can be reordered or removed but not modified.
23. [#1038] Make RTCDataChannel.id nullable and describe when it's set.
24. [#1036] Specify how transceivers get their mids in setLocal/RemoteDescription
25. [#1037] Specify when random mid generation happens
26. [#1039] Clarify which timestamp RTCStats.timestamp represents
27. [#1041] Label 'Warm-up example' as 'advanced p2p example'
28. [#1031] Don't fire events on a closed peer connection
29. [#1045] Clarifying exactly what "sdpFmtpLine" represents
30. [#1047] Adding "[EnforceRange]" to RTCDataChannelInit.id
31. [#1054] Throw InvalidModificationError if changing pool size after setLocalDescription
32. [#1055] Changing iceCandidatePoolSize to an octet and adding EnforceRange WebIDL extended attribute
33. [#1057] Add clockrate, channels, sdpFmtpLine to codec capability
34. [#1058] Define 'generation of ICE candidates' and add reference
35. [#1059] Specify how remote tracks get muted
36. [#1060] Specify when to end a remote track
37. [#1061] Remove connecting event from Event summary
38. [#1066] Specify relation between RtpSender and track
39. [#1056] Switch to new, consistent terminology when talking about exceptions
40. [#1030] Add stats selection algorithm based on sender or receiver of selector

Changes since November 23, 2016

1. [#899] Make stats MTI, remove overlap with stats spec
2. [#920] Remove ICE Agent text from RTCPeerConnection due to the new RTCICETransport objects.
3. [#937] Define what happens when transceiver.stop() is called.
4. [#938] Define what happens when setDirection() is called.
5. [#939] Remove "stopped" from removeTrack() and immediately stop sender.
6. [#940] Remove "stopped" from close, insertDTMF, and replaceTrack.
7. [#944] Clarify that sender does not send if sender.track is set to null.
8. [#949] Give legacy callbacks RTCSessionDescriptionInit so they can modify SDP.
9. [#956] Add API for setting QoS priority of data channels.
10. [#960] Allow replaceTrack(null)
11. [#963] Split transceiver direction into "direction" and "currentDirection"
12. [#966] setParameters rejects with InvalidStateError if transceiver.stopped is true.
13. [#968] Add ufrag to IceCandidate and use IceCandidate for end-of-candidates.
14. [#970] Clarify that setParameters cannot add or remove simulcast encodings.
15. [#972, #973] Add generic Error Object that can hold detailed error information.
16. [#936, #953, #967] Editorial: remove old in-spec issue text, update JSEP references, update hold examples, fix section titles

Changes since September 13, 2016

1. [#738] Add ability to get fingerprints of an RTCCertificate
2. [#783] Clarify supported DTMF characters
3. [#785] Reject invalid DTMF characters when inserted
4. [#786] If track is ended or muted, send silence for audio or black frame for video
5. [#790] Add checks that verify that a candidate matches a remote media decription
6. [#791] Add text that fires the 'connectionstatechange' event
7. [#793] Define DTMF tone attribute and make it required
8. [#796] Language cleanup around use of MediaTrackSettings
9. [#797] Clarify when negotiation-needed flag is cleared

10. [#804] Clarify that JSEP is normative in some cases; also numerous small editorial fixes
11. [#805] Reduce insertDTMF max duration from 8000 ms to 6000 ms
12. [#807] Clarify that empty string in DTMFToneChangeEvent indicates that the previous tones (plural) have completed.
13. [#809] Clarify that InvalidStateError is thrown if insertDTMF is called on a stopped sender.
14. [#815] Change IceConnectionState to match PeerConnection state in certain edge cases.
15. [#833, #865, #884, #904] Editorial: update JSEP references
16. [#835] Add definition link for NN and disallow SDP modification
17. [#837] Have insertDTMF validate toneBuffer before returning.
18. [#840] Remove reference to IANA registry for Statistics
19. [#844] Throw InvalidAccessError if removeTrack is called with invalid sender
20. [#847] Specify how to handle invalid data channel IDs, or lack of IDs.
21. [#851] Editorial: Fix wording for insertDTMF
22. [#852] Remove insertDTMF's duration and interToneGap attributes
23. [#853] Make insertDTMF tone string normalization mandatory
24. [#855] Clarify that insertDTMF's DTMFToneChangeEvent also requires toneBuffer to be empty in order to fire
25. [#860] More clarification around when removeTrack should throw an exception, now considering rollback as well
26. [#861] Clarify what setConfiguration changes
27. [#864] Define channel member of RTCDataChannelEventInit and make it required
28. [#871] Remove ability to modify RID via addTrack()
29. [#872] Pass peer identity to IdP via new RTCIdentityProviderOptions
30. [#875, #893] Major restructuring of createOffer and createAnswer to eliminate race conditions
31. [#877] Store RTCConfiguration so getConfiguration can return it
32. [#880, #896] Clean up definition of expires in certificates
33. [#882] Split gathering state variables into two types, gathering and gatherer, and clean up descriptions of values for each
34. [#883] Clarify that insertDTMF interToneGap is in milliseconds
35. [#895] Add steps in "setting a description" for rolling back transceivers.
36. [#900] Reject incoming tracks using transceiver.stop()

37. [#913] Overhaul NN text while adjusting it to key off transceivers
38. [#919] Remove incorrect statement related to IP leaking issue
39. [#929] Have RTCSessionDescription's sdp member default to ""
40. [#863, #870, #890, #892, #911, #912, #915, #926, #928, #935] Editorial: typos, links, dead text, WebIDL, Travis, etc.

Changes since July 22, 2016

1. [#713] Missing destruction sequence for ICE Agent
2. [#730] Revised WebRTC 1.0 RTCIceTransportState transition diagram
3. [#722] How setDirection interacts with active/inactive sender/receivers
4. [#716] Improve error handling for IdP proxy interactions
5. [#719] The IdP environment can be spoofed
6. [#733] Clarification on RTX in Codec Capabilities/Parameters
7. [#734] RTCRtpEncodingParameters attribute to turn on/off sending CN/DTX
8. [#737] Fix mistakes in examples
9. [#739] Replace set of senders/receivers/transceivers with algorithms
10. [#721] Specify the synchronous and queued steps for addIceCandidate
11. [#759] Clarification on receipt of multiple RTP encodings
12. [#758] Support replaceTrack with the previous track ended
13. [#745] Add steps to createOffer and explicitly specify what is queued
14. [#762] Remove closed check from addIceCandidate steps (covered by enqueue steps)
15. [#750] RTCIdentityProviderGlobalScope needs Exposed attributes
16. [#752] Add steps to createAnswer and explicitly specify what is queued
17. [#756] Integrate queueing into the setLocal/RemoteDescription steps
18. [#765] Adding more detail to the definition of the ICE `disconnected` state.
19. [#778] Remove void conformance requirement on interToneGap
20. [#779] Make duration and interToneGap attributes unsigned long

Changes since May 13, 2016

1. [#640, #641, #659, #679, #680, #681, #682, #686, #694, #696, #697, #707, #708, #711] General editorial fixes
2. [#642] Editorial: make last arg of addTransceiver optional
3. [#643] Document defaultIceServers as source of fingerprinting
4. [#646] Create table of RTCRtpEncodingParameters for RtpSender/RtpReceiver
5. [#648] Clarify MIME (media/sub-) type
6. [#649] Example of how to do hold
7. [#662] Clarify effect of RTCRtpReceiver.track.stop()
8. [#663] Define a 7XX STUN error code
9. [#665] Clarify when setDirection() acts
10. [#666] Clarify that transports can be null
11. [#676] Transceiver.stop() causes negotiationneeded to be set
12. [#677] Clean up rtcpTransport description
13. [#701] In addTrack, mention that MSID of new track is added
14. [#702, #704] Define algs for creating sender/receiver/transceiver, then use them in addTrack() and addTransceiver()
15. [#725] Change 'process to apply candidate' to 'add the ICE candidate'

Changes since February 15, 2016

1. [#475] Definition of Active for an RTCRtpReceiver
2. [#500] Reserve and use RangeError for scaleResolutionDownBy < 1.0
3. [#504] Add getParameters() method to RTCRtpReceiver
4. [#509] RID unmodifiable in setParameters()
5. [#510] Gather spec text about the ICE Agent at one place
6. [#512] Use 'connection' as configuration target instead of User Agent
7. [#505] Add activateReceiver method to RTCRtpTransceiver
8. [#516] Support for DTMF tones A-D
9. [#499] Certificate API: add getAlgorithm method
10. [#507] Make the definition of addIceCandidate() more explicit
11. [#525] Add STUN Error Code reference

12. [#524] Add error codes reference (RTCPeerConnectionIceErrorEvent)
13. [#522] Let setting ice candidate pool size trigger start of gathering
14. [#519] Relation between local track and outgoing encoding
15. [#520] Add text about 'remote sources' and how they are stopped
16. [#527] Enable trickling of end-of-candidates through addIceCandidate
17. [#544] Remove "public" from ice transport policy
18. [#547] Datachannel label and protocol are USVString
19. [#552] Never close the RTCPeerConnection if setting a local/remote description fails
20. [#535] Update MID to be random values when not received in offer
21. [#553] Move 'closed' state from RTCSignalingState to RTCPeerConnectionState
22. [#557] Splitting apart RTCIceConnectionState and RTCIceTransportState
23. [#560] Changing from callback interface to dictionary for RTCIdentityProvider
24. [#574] Make RTCSessionDescription readonly, and createOffer return dictionary
25. [#577] Make RtpSender.track nullable
26. [#587] Defining how track settings are set for remote tracks
27. [#603] Add closed state and same state checks to update ice connection/gathering state steps
28. [#604] ReplaceTrack: Use sender's transceiver to determine if a 'simple track swap' is enough
29. [#606] RTCIceCandidate: Use nullable members in init dictionary to describe constructor behavior
30. [#466] Use an enum to describe directionality of RTP Stream
31. [#602] addTransceiver(): Throw a TypeError on a bogus track kind
32. [#610] Server cannot be reached - Issues with IPv6
33. [#611] Clarify ICE consent freshness feedback
34. [#618] Fix RTCPeerConnection legacy overloads
35. [#620] RTCRtpTransceiver: add setDirection and readonly direction attribute
36. [#625] Unifiy DTMF time with rtcweb WG
37. [#630] Add ICE candidate type references
38. [#635] pc.addTrack: Add kind check when reusing a sender and skip early returns
39. [#636] replaceTrack: Use 'transceiver kind' instead of track.kind (track may be null)

Changes since January 26, 2016

1. [#485] Update SOTD as the document is now quite stable and the group is looking for wide review
2. [#468, #335] Replace DOMError with DOMException
3. [#472, #319] Update error reports to align with existing DOM Errors
4. [#491, #479] Specify error when rejecting invalid SDP changes
5. [#462] Add PeerConnection.activateSender() and update early media example
6. [#434] Change setParameters call to be Async

Changes since December 22, 2015

1. [#179, #439] Document IP address leakage in RTCIceCandidate
2. [#439] Complete security considerations based on security questionnaire and IP address discussions #439
3. [#446] Non-nullable RTCTrackEvent args means Init dict members are required
4. [#449] Clarify flow of SDP exchanges (Update simple p2p example)
5. [#451] Clean up event handler attribute descriptions
6. [#452, #438] Make replaceTrack() handle "not sending yet" case
7. [#454] Add contributing source voice activity flag
8. [#455, #439] Add references to parsing stun/turn URLs section
9. [#456, #338] SDP changes between the createOffer and setLocalDescription (add JSEP reference)
10. [#459] Add non-normative ICE state transitions
11. [#460, #461] getRemoteCertificates() behavior in "new" and "connecting" states
12. [#465, #140] Use ErrorEvent as interface for events emitted by RTCDataChannel.onerror
13. [#469, #382, #373] Reject changes to peerIdentity and certificates in setConfiguration
14. [#474, #406] Define RTCIceTransport.component when RTP/RTCP mux is in use

Changes since November 23, 2015

1. [#353] Plan X: Add an API for using RID to do simulcast
2. [#365] Adding an accessor for the browser-configured ICE servers

3. [#398] Make RtpTransceiver.mid nullable and remove RtpSender.mid and RtpReceiver.mid
4. [#402, #391] Remove requirement about DTMF tones A-D
5. [#403, #377] Use positive values for AudioLevel
6. [#401, #267] Add bitrate definition
7. [#404] Remove 'Events on MediaStream' section (duplicates new text in Media Capture spec)
8. [#410, #328] Make RTCBundlePolicy Enum section normative
9. [#411, #408] Clarify component for IceTransport when RTP/RTCP mux is used
10. [#414] Define ReSpec processor for cross-reference to JSEP
11. [#418] Make degradationPreference per-sender instead of per-encoding
12. [#416] RTCRtpSender.replaceTrack() fixes (e.g. handle closed RTCPeerConnection)
13. [#421] Require sdp in RTCSessionDescription{,Init}
14. [#422] Remove confusing paragraph on fourth party interception
15. [#423] Add specific references to JSEP where possible
16. [#428] Don't create a default stream in 'dispatch a receiver' steps
17. [#429] Adding expires attribute to generateCertificate
18. [#430] Add maxFramerate knob for simulcast
19. [#432] Update RTCIceTransportPolicy
20. [#433] Use unsigned long ssrc in stats
21. [#424] Editorial: Distinguish states from their attribute representation

Changes since October 6, 2015

1. [#325] Adding additional members to RTCIceCandidate dictionary
2. [#327] Adding sha-256 to the certificate management options for RSA
3. [#342] Using DOMTimestamp for RTCCertificate::expires
4. [#293] Add RTCRtpTransceiver and PeerConnection.addMedia
5. [#366, #343] Use RTCDegradationPreference
6. [#374] Throw on too long label/protocol in createDataChannel()
7. [#266] Tidy up setLocal/RemoteDescription processing model
8. [#361] Adding setCodecPreferences to RTCRtpTransceiver

9. [#371] Add RtcpMuxPolicy
10. [#385, #312] Don't invoke public API in legacy function section
11. [#394, #393] don't throw on empty iceServers list

Changes since September 22, 2015

1. [#289, #153] Add way to set size of ICE candidate pool
2. [#256] Fix prose on getStats() wo/selector + move type check to sync section
3. [#242] Remove SyntaxError on malformed ICE candidate
4. [#284] Add icecandidateerror event for indicating ICE gathering errors
5. [#298] Add support for codec reordering and removal in RtpParameters
6. [#311] Fixing syntax for required RTCCertificate arguments
7. [#280] Add extra IceTransport read-only attributes and methods
8. [#291] Add PeerConnection.connectionState
9. [#300, #4, #6, #276] Add API to get SSRC and audio levels
10. [#301] Fix RTCStatsReport with object and maplike instead of getter
11. [#302] (Partly) removing interface use for RTCSessionDescription and RTCIceCandidate
12. [#314, #299] Update the operations queue to handle promises and closed signalling
13. [#273] Add a bunch of fields to RtpParameters and RtpEncodingParameters

Changes since June 11, 2015

1. [#234] Add RTCRtpParameters, RTCRtpSender.getParameters, and RTCRtpSender.setParameters
2. [#225] Support for pending and current SDP
3. [#229] Removing the weird optionality from RTCSessionDescription and its constructor.
4. [#235] Modernize getStats() with promises
5. [#243] Mark candidate property of RTCIceCandidateInit required
6. [#248] Fix error handling for certificate management
7. [#259] Change type of RtpEncodingParameters.priority to an enum
8. [#21, #262] Sort out 2119 MUSTs and SHOULDs
9. [#268] Add RtpEncodingParameters.maxBitrate

10. [#241] Add RtpSender.transport, RtpReceiver.transport, RTCDtlsTransport, RTCIceTransport, etc
11. [#224, #261] Sort out when responding PeerConnection reaches iceConnetionState completed
12. [#303] Replace track without renegotiation
13. [#269] Add RTCRtpSender.getCapabilities and RTCRtpReceiver.getCapabilities

Changes since March 6, 2015

1. [PR #167] Removed RTCPeerConnection.createDTMFSender and added RTCRtpSender.dtmf, along with corresponding examples.
2. [PR #184] RTCPeerConnection will NOT connect unless identity is verified.
3. [PR #27] Documenting practice with candidate events
4. [PR #203] Rewrote mitigations text for security considerations section
5. [PR #192] Added support for auth tokens. Fixes #190
6. [PR #207] Update ice config examples to use multiple urls and *s schemes
7. [PR #210] Optional RTCConfiguration in RTCPC constructor
8. [PR #171] Add RTCAnswerOptions (with common RTCOfferAnswerOptions dictionary)
9. [PR #178] Identity provider interface redesign
10. [PR #193] Add .mid property to sender/receiver. Fixes #191
11. [PR #218] Enqueue addIceCandidate
12. [PR #213 (1)] Rename updateIce() to setConfiguration()
13. [PR #213 (2)] Make RTCPeerConnection.setConfiguration() replace the existing configuration
14. [PR #214] Certificate management API (Bug 21880)
15. [PR #220] Clarify muted state (proposed fix for issue #139)
16. [PR #221] Define when RTCRtpReceivers are created and dispatced (issue #198)
17. [PR #215] Adding expires attribute to certificate management
18. [PR #233] Add a "bufferedamountlow" event

Changes since December 5, 2014

1. Properly define the negotiationneeded event, and its interactions with other API calls.
2. Add support for RTCRtpSender and RTCRtpReceiver.
3. Update misleading local/RemoteDescription attribute text.
4. Add RTCBundlePolicy.
5. All callback-based methods have been moved to a legacy section, and replaced by same-named overloads using Promises instead.
6. [PR #194] Added first version of Security Considerations (more work needed)
7. Updated identity provider structure.

Changes since June 4, 2014

1. Bug 25724: Allow garbage collection of closed PeerConnections
2. Bug 27214: Add onicegatheringstatechange event
3. Bug 26644: Fixing end of candidates event

Changes since April 10, 2014

1. Bug 25774: Mixed isolation

Changes since April 10, 2014

1. Bug 25855: Clarification about conformance requirements phrased as algorithms
2. Bug 25892: SignalingStateChange event should be fired only if there is a change in signaling state.
3. Bug 25152: createObjectURL used in examples is no longer supported by Media Capture and Streams.
4. Bug 25976: DTMFSender.insertDTMF steps should validate the values of duration and interToneGap.
5. Bug 25189: Mandatory errorCallback is missing in examples for getStats.
6. Bug 25840: Creating DataChannel with same label.
7. Updated comment above example ice state transitions (discussed in Bug 25257).
8. Updated insertDTMF() algorithm to ignore unrecognized characters (as discussed in bug 25977).

9. Made formatting of references to ice connection state consistent.
10. Made insertDTMF() throw on unrecognized characters (used to ignore).
11. Removed requestIdentity from RTCConfiguration and RTCOfferAnswerOptions. Removed RTCOfferAnswerOptions as a result.
12. Adding isolated property and associated event to MediaStreamTrack.

Changes since March 21, 2014

1. Changes to identity-related text:
    - Removed noaccess constraint
    - Add the ability to peerIdentity constrain RTCPeerConnection, which limits communication to a single peer
    - Change the way that the browser communicates with IdP to a message channel (http://www.w3.org/TR/webmessaging/#message-channels)
    - Improved error feedback from IdP interactions (added new events with more detailed context)
    - Changed the way that an IdP is able to request user login (LOGINNEEDED message)
2. Bug 25155: maxRetransmitTime is not the name of the SCTP concept it points to.

Changes since January 27, 2014

1. Refined identity assertion generation and validation.
2. Default DTMF gap changed from 50 to 70 ms.
3. Bug 24875: Examples in the WebRTC spec are not updated As per the modified API.

Changes since August 30, 2013

1. Make RTCPeerConnection close method be idempotent.
2. Clarified ICE server configuration could contain URI types other than STUN and TURN.
3. Changed the DTMF timing values.
4. Allow offerToReceiveAudio/video indicate number of streams to offer.
5. ACTION-98: Added text about clamping of maxRetransmitTime and maxRetransmits.

6. ACTION-88: Removed nullable types from dictionaries (added attribute default values for attributes that would be left uninitialized without the init dictionary present.
7. InvalidMediaStreamTrackError changed to InvalidParameter.
8. Fire NetworkError when the data transport is closed with an error.
9. Add an exception for data channel with trying to use existing code.
10. Change maxRetransmits to be an unsigned type.
11. Clarify state changes when ICE restarts.
12. Added InvalidStateError exception for operations on an RTCPeerConnection that is closed.
13. Major changes to Identity Proxy section.
14. (ACTION: 95) Moved IceTransports (constraint) to RTCConfiguration dictionary.
15. (ACTION: 95) Introduced RTCOfferAnswerOptions and RTCOfferOptions dictionaries.
16. (ACTION: 95) Removed constraints argument from addStream() (and removed IANA Constraints section).
17. Added validation of the RTCConfiguration dictionary argument(s).
18. Added getConfiguration() on RTCPeerConnection.

Changes since June 3, 2013

1. Removed synchronous section left-overs.
2. RTCIceServer now accepts multiple URLs.
3. Redefined the meaning of negotiated for DataChannel.
4. Made iceServers a sequence (instead of an Array).
5. Updated error reporting (to use DOMError and camel cased names).
6. Added success and failure callbacks to addIceCandidate().
7. Made local/remoteDescription attributes nullable.
8. Added username member to RTCIceServer dictionary.

Changes since March 22, 2013

1. Added IceRestart constraint.
2. Big updates on DataChannel API to use new channel setup procedures.

Changes since Feb 22, 2013

1. Example review: Updated DTMF and Stats examples. Added text about when to fire "negotiationneeded" event to align with examples.
2. Updated RTCPeerConnection state machine. Added a shared processing model for setLocalDescription()/setRemoteDescription().
3. Updated simple callflow to match the current API.

Changes since Jan 16, 2013

1. Initial import of Statistics API to version 2.
2. Integration of Statistics API version 2.5 started.
3. Updated Statistics API to match Boston/list discussions.
4. Extracted API extensions introduced by features, such as the P2P Data API, from the RTCPeerConnection API.
5. Updated DTMF algorithm to dispatch an event when insertDTMF() is called with an empty string to cancel future tones.
6. Updated DTMF algorithm to not cancel and reschedule if a playout task is running (only update toneBuffer and other values).

Changes since Dec 12, 2012

1. Changed AudioMediaStreamTrack to RTCDTMFSender and gave it its own section. Updated text to reflect most recent agreements. Also added examples section.
2. Replaced the localStreams and remoteStreams attributes with functions returning sequences of MediaStream objects.
3. Added spec text for attributes and methods adopted from the WebSocket interface.
4. Changed the state ENUMs and transition diagrams.
5. Aligned the data channel processing model a bit more with WebSockets (mainly closing the underlying transport).

Changes since Nov 13, 2012

1. Made some clarifications as to how operation queuing works, and fixed a few errors with the error handling description.

2. Introduced new representation of tracks in a stream (removed MediaStreamTrackList). Added algorithm for creating a track to represent an incoming network media component.
3. Renamed MediaStream.label to MediaStream.id (the definition needs some more work).

Changes since Nov 03, 2012

1. Added text describing the queuing mechanism for RTCPeerConnection.
2. Updated simple P2P example to include all mandatory (error) callbacks.
3. Updated P2P data example to include all mandatory (error) callbacks. Also added some missing RTC prefixes.

Changes since Oct 19, 2012

1. Clarified how createOffer() and createAnswer() use their callbacks.
2. Made all failure callbacks mandatory.
3. Added error object types, general error handling principles, and rules for when errors should be thrown.

Changes since Sept 23, 2012

1. Restructured the document layout and created separate sections for features like Peer-to-peer Data API, Statistics and Identity.

Changes since Aug 16, 2012

1. Replaced stringifier with serializer on RTCSessionDescription and RTCIceCandidate (used when JSON.stringify() is called).
2. Removed offer and createProvisionalAnswer arguments from the createAnswer() method.
3. Removed restart argument from the updateIce() method.
4. Made RTCDataChannel an EventTarget
5. Updated simple RTCPeerConnection example to match spec changes.
6. Added section about RTCDataChannel garbage collection.
7. Added stuff for identity proxy.

8. Added stuff for stats.
9. Added stuff peer and ice state reporting.
10. Minor changes to sequence diagrams.
11. Added a more complete RTCDataChannel example
12. Various fixes from Dan's Idp API review.
13. Patched the Stats API.

Changes since Aug 13, 2012

1. Made the RTCSessionDescription and RTCIceCandidate constructors take dictionaries instead of a strings. Also added detailed stringifier algorithm.
2. Went through the list of issues (issue numbers are only valid with HEAD at fcda53c460). Closed (fixed/wontfix): 1, 8, 10, 13, 14, 16, 18, 19, 22, 23, 24. Converted to notes: 4, 12. Updated: 9.
3. Incorporate changes proposed by Li Li.
4. Use an enum for DataChannelState and fix IDLs where using an optional argument also requires all previous optional arguments to have a default value.

Changes since Jul 20, 2012

1. Added RTC Prefix to names (including the notes below).
2. Moved to new definition of configuration and ice servers object.
3. Added correlating lines to candidate structure.
4. Converted setLocalDescription and setRemoteDescription to be asynchronous.
5. Added call flows.

Changes since Jul 13, 2012

1. Removed peer attribute from RTCPeerConnectionIceEvent (duplicates functionality of Event.target attribute).
2. Removed RTCIceCandidateCallback (no longer used).
3. Removed RTCPeerConnectionEvent (we use a simple event instead).

4. Removed RTCSdpType argument from setLocalDescription() and setRemoteDescription(). Updated simple example to match.

Changes since May 28, 2012

1. Changed names to use RTC Prefix.
2. Changed the data structure used to pass in STUN and TURN servers in configuration.
3. Updated simple RTCPeerConnection example (RTCPeerConnection constructor arguments; use icecandidate event).
4. Initial import of new Data API.
5. Removed some left-overs from the old Data Stream API.
6. Renamed "underlying data channel" to "underlying data transport". Fixed closing procedures. Fixed some typos.

Changes since April 27, 2012

1. Major rewrite of RTCPeerConnection section to line up with IETF JSEP draft.
2. Added simple RTCPeerConnection example. Initial update of RTCSessionDescription and RTCIceCandidate to support serialization and construction.

Changes since 21 April 2012

1. Moved MediaStream and related definitions to getUserMedia.
2. Removed section "Obtaining local multimedia content".
3. Updated getUserMedia() calls in examples (changes in Media Capture TF spec).
4. Introduced MediaStreamTrackList interface with support for adding and removing tracks.
5. Updated the algorithm that is run when RTCPeerConnection receives a stream (create new stream when negotiated instead of when data arrives).

Changes since 12 January 2012

1. Clarified the relation of Stream, Track, and Channel.

Changes since 17 October 2011

1. Tweak the introduction text and add a reference to the IETF RTCWEB group.
2. Changed the first argument to getUserMedia to be an object.
3. Added a MediaStreamHints object as a second argument to RTCPeerConnection.addStream.
4. Added AudioMediaStreamTrack class and DTMF interface.

Changes since 23 August 2011

1. Separated the SDP and ICE Agent into separate agents and added explicit state attributes for each.
2. Removed the send method from PeerConenction and associated callback function.
3. Modified MediaStream() constructor to take a list of MediaStreamTrack objects instead of a MediaStream. Removed text about MediaStream parent and child relationship.
4. Added abstract.
5. Moved a few paragraphs from the MediaStreamTrack.label section to the MediaStream.label section (where they belong).
6. Split MediaStream.tracks into MediaStream.audioTracks and MediaStream.videoTracks.
7. Removed a sentence that implied that track access is limited to LocalMediaStream.
8. Updated a few getUserMedia()-examples to use MediaStreamOptions.
9. Replaced calls to URL.getObjectURL() with URL.createObjectURL() in example code.
10. Fixed some broken getUserMedia() links.
11. Introduced state handling on MediaStreamTrack (removed state handling from MediaStream).
12. Reintroduced onended on MediaStream to simplify checking if all tracks are ended.
13. Aligned the MediaStreamTrack ended event dispatching behavior with that of MediaStream.
14. Updated the LocalMediaStream.stop() algorithm to implicitly use the end track algorithm.
15. Replaced an occurrence the term finished track with ended track (to align with rest of spec).
16. Moved (and extended) the explanation about track references and media sources from LocalMediaStream to MediaStreamTrack.

A. 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.

B. References

B.1 Normative references

[BUNDLE]
: *Negotiating Media Multiplexing Using the Session Description Protocol (SDP)*. C. Holmberg; H. Alvestrand; C. Jennings. IETF. 31 August 2017. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-mmusic-sdp-bundle-negotiation

[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. W3C. 26 October 2017. W3C Working Draft. URL: https://www.w3.org/TR/FileAPI/

[FIPS-180-4]
: *FIPS PUB 180-4 Secure Hash Standard*. U.S. Department of Commerce/National Institute of Standards and Technology. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[GETUSERMEDIA]
: *Media Capture and Streams*. Daniel Burnett; Adam Bergkvist; Cullen Jennings; Anant Narayanan; Bernard Aboba. W3C. 3 October 2017. W3C Candidate Recommendation.

URL: https://www.w3.org/TR/mediacapture-streams/

[HIGHRES-TIME]

High Resolution Time Level 2. Ilya Grigorik; James Simonsen; Jatinder Mann. W3C. 1 March 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/hr-time-2/

[HTML]

HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HTML51]

HTML 5.1 2nd Edition. Steve Faulkner; Arron Eicholz; Travis Leithead; Alex Danilo. W3C. 3 October 2017. W3C Recommendation. URL: https://www.w3.org/TR/html51/

[IANA-HASH-FUNCTION]

Hash Function Textual Names. IANA. URL: https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xml

[ICE]

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

[INFRA]

Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[JSEP]

Javascript Session Establishment Protocol. Justin Uberti; Cullen Jennings; Eric Rescorla. IETF. 10 October 2017. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-jsep/

[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

[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

[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

[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

[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

[RFC5761]

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

[RFC5888]

The Session Description Protocol (SDP) Grouping Framework. G. Camarillo; H. Schulzrinne. IETF. June 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5888

[RFC6236]

Negotiation of Generic Image Attributes in the Session Description Protocol (SDP). I. Johansson; K. Jung. IETF. May 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6236

[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

[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

[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

[RFC6749]

The OAuth 2.0 Authorization Framework. D. Hardt, Ed.. IETF. October 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6749

[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

[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

[RFC7515]

JSON Web Signature (JWS). M. Jones; J. Bradley; N. Sakimura. IETF. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7515

[RFC7635]

Session Traversal Utilities for NAT (STUN) Extension for Third-Party Authorization. T. Reddy; P. Patil; R. Ravindranath; J. Uberti. IETF. August 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7635

[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

[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

[RTCWEB-AUDIO]

WebRTC Audio Codec and Processing Requirements. JM. Valin; C. Bran. IETF. May 2016. Proposed Standard. URL: https://tools.ietf.org/html/rfc7874

[RTCWEB-DATA]

RTCWeb Data Channels. R. Jesup; S. Loreto; M. Tuexen. IETF. 14 October 2015. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel

[RTCWEB-DATA-PROTOCOL]

RTCWeb Data Channel Protocol. R. Jesup; S. Loreto; M. Tuexen. IETF. 14 October 2015. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-protocol

[RTCWEB-RTP]

Web Real-Time Communication (WebRTC): Media Transport and Use of RTP. C. Perkins; M. Westerlund; J. Ott. IETF. 17 March 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage

[RTCWEB-TRANSPORT]

Transports for RTCWEB. H. Alvestrand. IETF. 31 October 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-transports

[SCTP-SDP]

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. 20 March 2017. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-mmusic-sctp-sdp

[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

[STUN-BIS]

Session Traversal Utilities for NAT (STUN). M. Petit-Huguenin; G. Salgueiro; J. Rosenberg; D. Wing; R. Mahy; P. Matthews. IETF. 16 February 2017. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-tram-stunbis

[TRICKLE-ICE]

Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol. E. Ivov; E. Rescorla; J. Uberti. IETF. 20 July 2015. Internet Draft (work in progress). URL: http://datatracker.ietf.org/doc/draft-ietf-mmusic-trickle-ice

[TSVWG-RTCWEB-QOS]

DSCP Packet Markings for WebRTC QoS. S. Dhesikan; C. Jennings; D. Druta; P. Jones; J. Polk. IETF. 22 August 2016. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-tsvwg-rtcweb-qos

[WebCryptoAPI]

Web Cryptography API. Mark Watson. W3C. 26 January 2017. W3C Recommendation. URL: https://www.w3.org/TR/WebCryptoAPI/

[WEBIDL]

Web IDL. Cameron McCormack; Boris Zbarsky; Tobie Langel. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

[WEBIDL-1]

WebIDL Level 1. Cameron McCormack. W3C. 15 December 2016. W3C Recommendation. URL: https://www.w3.org/TR/2016/REC-WebIDL-1-20161215/

[webmessaging]

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

[WEBRTC-IDENTITY]

Identity for WebRTC 1.0. Adam Bergkvist; Daniel Burnett; Cullen Jennings; Anant Narayanan; Bernard Aboba; Taylor Brandstetter. W3C. W3C Candidate Recommendation. URL: https://w3c.github.io/webrtc-identity/identity.html

[WEBRTC-STATS]

Identifiers for WebRTC's Statistics API. Harald Alvestrand; Varun Singh. W3C. 3 July 2018. 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

B.2 Informative references

[FLEXFEC]

RTP Payload Format for Flexible Forward Error Correction (FEC). V. Singh; A. Begen; M. Zanaty; G. Mandyam. IETF. 3 July 2017. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-ietf-payload-flexible-fec-scheme

[IANA-RTP-2]

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

[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/

[OAUTH-POP-KEY-DISTRIBUTION]

OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution. J. Bradley; P. Hunt; M. Jones; H. Tschofenig. IETF. 5 March 2015. Internet Draft (work in progress). URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-pop-key-distribution/

[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

[RFC5109]

RTP Payload Format for Generic Forward Error Correction. A. Li, Ed.. IETF. December 2007. Proposed Standard. URL: https://tools.ietf.org/html/rfc5109

[RFC5285]

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

[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

[RTCWEB-IP-HANDLING]

WebRTC IP Address Handling Recommendations. Guo-wei Shieh; Justin Uberti. IETF. 20 March 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-ip-handling

[RTCWEB-OVERVIEW]

Overview: Real Time Protocols for Brower-based Applications. H. Alvestrand. IETF. 14 February 2014. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-overview

[RTCWEB-SECURITY]

Security Considerations for WebRTC. Eric Rescorla. IETF. 22 January 2014. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security

[RTCWEB-SECURITY-ARCH]
WebRTC Security Architecture. Eric Rescorla. IETF. 10 December 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch

[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

[WEBSOCKETS-API]
The WebSocket API. Ian Hickson. W3C. 20 September 2012. W3C Candidate Recommendation. URL: https://www.w3.org/TR/websockets/

[XMLHttpRequest]
XMLHttpRequest Level 1. Anne van Kesteren; Julian Aubourg; Jungkee Song; Hallvord Steen et al. W3C. 6 October 2016. W3C Note. URL: https://www.w3.org/TR/XMLHttpRequest/

↑