W3C Working Draft

WebRTC 1.0: Real-time Communication Between Browsers

W3C Working Draft 27 October 2011

This version:
http://www.w3.org/TR/2011/WD-webrtc-20111027/
Latest published version:
http://www.w3.org/TR/webrtc/
Latest editor's draft:
http://dev.w3.org/2011/webrtc/editor/webrtc.html
Previous version:
none
Editors:
Adam Bergkvist, Ericsson
Daniel C. Burnett, Voxeo
Cullen Jennings, Cisco
Anant Narayanan, Mozilla

Initial Author of this Specification was Ian Hickson, Google Inc., with the following copyright statement:
© Copyright 2004-2011 Apple Computer, Inc., Mozilla Foundation, and Opera Software ASA. You are granted a license to use, reproduce and create derivative works of this document.

All subsequent changes since 26 July 2011 done by the W3C WebRTC Working Group are under the following Copyright:
© 2011 W3C® (MIT, ERCIM, Keio), All Rights Reserved. Document use rules apply.

For the entire publication on the W3C site the liability and trademark rules apply.

Abstract

This document defines a set of APIs that allow local media, including audio and video, to be requested from a platform, media to be sent over the network to another browser or device implementing the appropriate set of real-time protocols, and media received from another browser or device to be processed and displayed locally. This specification is being developed in conjunction with a protocol specification developed by the IETF RTCWEB group.

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 http://www.w3.org/TR/.

This document is a First Public Working Draft and is not complete. It is subject to major changes and, while early experimentations are encouraged, it is therefore not intended for implementation. The API is based on preliminary work done in the WHATWG. The Web Real-Time Communications Working Group expects this specification to evolve significantly based on:

- The outcomes of ongoing exchanges in the companion RTCWEB group at IETF to define the set of protocols that, together with this document, will enable real-time communications in Web browsers.
- Privacy issues that arise when exposing local capabilities and local streams.

This version is outdated!
For the latest version, please look at https://www.w3.org/TR/webrtc/.
expand

IPR2025-00265
SportsCastr Inc. EX-2026

- Feedback received from other groups and individuals.

As the specification matures, the group hopes to strike the right balance between a low-level API that would enable interested parties to tweak potentially complex system parameters, and a more high-level API that Web developers can use without *a priori* technical knowledge about real-time communications.

This document was published by the Web Real-Time Communications Working Group as a Working Draft. This document is intended to become a W3C Recommendation. If you wish to make comments regarding this document, please send them to public-webrtc@w3.org (subscribe, archives). All feedback is welcome.

Publication as a Working Draft 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 5 February 2004 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.

Table of Contents

1. Introduction
2. Obtaining local multimedia content
    - 2.1 Definition
        - 2.1.1 NavigatorUserMedia
            - 2.1.1.1 Methods
        - 2.1.2 MediaStreamOptions
            - 2.1.2.1 Attributes
        - 2.1.3 NavigatorUserMediaSuccessCallback
            - 2.1.3.1 Methods
        - 2.1.4 NavigatorUserMediaError and NavigatorUserMediaErrorCallback
            - 2.1.4.1 Attributes
            - 2.1.4.2 Constants
            - 2.1.4.3 Methods
    - 2.2 Examples
3. Stream API
    - 3.1 Introduction
    - 3.2 Interface definitions
        - 3.2.1 MediaStream
            - 3.2.1.1 Attributes
            - 3.2.1.2 Methods
            - 3.2.1.3 Constants
        - 3.2.2 LocalMediaStream
            - 3.2.2.1 Methods
        - 3.2.3 MediaStreamTrack
            - 3.2.3.1 Attributes
        - 3.2.4 MediaStreamRecorder
            - 3.2.4.1 Methods
        - 3.2.5 BlobCallback
            - 3.2.5.1 Methods
        - 3.2.6 URL
            - 3.2.6.1 Methods
    - 3.3 Examples
4. Peer-to-peer connections
    - 4.1 PeerConnection
        - 4.1.1 Attributes
        - 4.1.2 Methods
        - 4.1.3 Constants

4.2 SignalingCallback
4.2.1 Methods
4.3 Examples
5. The data stream
6. Garbage collection
7. Event definitions
7.1 MediaStreamEvent
7.1.1 Attributes
7.1.2 Methods
8. Event summary
9. application/html-peer-connection-data
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 video-conferencing in HTML:

- Getting a multimedia stream (video, audio, or both) from local devices (video cameras, microphones, Web cams) or from prerecorded files provided by the user.
- Recording such streams locally.
- Connecting to remote peers using NAT-traversal technologies such as ICE, STUN, and TURN.
- Sending the locally-produced streams to remote peers and receiving streams from remote peers.
- Displaying such streams (both the locally-produced ones and the remotely-obtained ones) locally using the `video` or `audio` elements.
- Sending arbitrary data 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.

2. Obtaining local multimedia content

2.1 Definition

2.1.1 NavigatorUserMedia

WebIDL

```
[NoInterfaceObject]
interface NavigatorUserMedia {
    void getUserMedia (MediaStreamOptions? options, NavigatorUserMediaSuccessCallback?
successCallback, optional NavigatorUserMediaErrorCallback? errorCallback);
};
```

2.1.1.1 Methods

`getUserMedia`

Prompts the user for permission to use their Web cam or other video or audio input.

The *options* argument is an object of type `MediaStreamOptions`. The object can contain the following boolean properties:

"**`audio`**"

The provided media needs to include audio data.

"`video`"
: The provided media needs to include video data.

If the user accepts, the *successCallback* is invoked, with a suitable `LocalMediaStream` object as its argument.

If the user declines, the *errorCallback* (if any) is invoked.

When the `getUserMedia()` method is called, the user agent must run the following steps:

1. Let *options* be the method's first argument.
2. Let *successCallback* be the callback indicated by the method's second argument.
3. Let *errorCallback* be the callback indicated by the method's third argument, if any, or null otherwise.
4. If *successCallback* is null, abort these steps.
5. Let *audio* be true.
6. Let *video* be true.
7. If the *audio* property in *options* is present and set to false, let *audio* be false.
8. If the *video* property in *options* is present and set to false, let *video* be false.
9. If both *audio* and *video* are still false, then throw a `NOT_SUPPORTED_ERR` exception and abort these steps.
10. Return, and run the remaining steps asynchronously.
11. Optionally, e.g. based on a previously-established user preference, for security reasons, or due to platform limitations, jump to the step labeled *failure* below.
12. Prompt the user in a user-agent-specific manner for permission to provide the entry script's origin with a `LocalMediaStream` object representing a media stream.

    If *audio* is true, then the provided media should include an audio track. If *audio* is false, then the provided media must not include an audio track.

    If *video* is true, then the provided media should include a video track. If *video* is false, then the provided media must not include a video track.

    User agents are encouraged to default to using the user's primary or system default camera and/or microphone (as appropriate) to generate the media stream. User agents may allow users to use any media source, including pre-recorded media files.

    User agents may wish to offer the user more control over the provided media. For example, a user agent could offer to enable a camera light or flash, or to change settings such as the frame rate or shutter speed.

    If the user grants permission to use local recording devices, user agents are encouraged to include a prominent indicator that the devices are "hot" (i.e. an "on-air" or "recording" indicator).

    If the user denies permission, jump to the step labeled *failure* below. If the user never responds, this algorithm stalls on this step.
13. Let *stream* be the `LocalMediaStream` object for which the user granted permission.
14. Queue a task to invoke *successCallback* with *stream* as its argument.

15. Abort these steps.
16. *Failure*: If *errorCallback* is null, abort these steps.
17. Let *error* be a new `NavigatorUserMediaError` object whose `code` attribute has the numeric value 1 (`PERMISSION_DENIED`).
18. Queue a task to invoke *errorCallback* with *error* as its argument.

The task source for these tasks is the user interaction task source.

Parameter	Type	Nullable	Optional	Description
options	`MediaStreamOptions`	✓	✗	
successCallback	`NavigatorUserMediaSuccessCallback`	✓	✗	
errorCallback	`NavigatorUserMediaErrorCallback`	✓	✓	

No exceptions.
Return type: `void`

WebIDL

```
Navigator implements NavigatorUserMedia;
```

All instances of the `Navigator` type are defined to also implement the `NavigatorUserMedia` interface.

2.1.2 MediaStreamOptions

WebIDL

```
[NoInterfaceObject]
interface MediaStreamOptions {
    attribute boolean audio;
    attribute boolean video;
};
```

2.1.2.1 Attributes

`audio` of type boolean
Set to false if an audio track is not required, default is true
No exceptions.

`video` of type boolean
Set to false if a video track is not required, default is true
No exceptions.

2.1.3 NavigatorUserMediaSuccessCallback

WebIDL

```
[Callback=FunctionOnly, NoInterfaceObject]
interface NavigatorUserMediaSuccessCallback {
    void handleEvent (LocalMediaStream stream);
};
```

2.1.3.1 Methods

`handleEvent`

(Explanation of handleEvent TBD)

Parameter	Type	Nullable	Optional	Description
stream	`LocalMediaStream`	✗	✗	

No exceptions.
Return type: `void`

2.1.4 NavigatorUserMediaError and NavigatorUserMediaErrorCallback

WebIDL

```
[NoInterfaceObject]
interface NavigatorUserMediaError {
    const unsigned short PERMISSION_DENIED = 1;
    readonly attribute unsigned short code;
};
```

2.1.4.1 Attributes

`code` of type unsigned short, readonly
Returns the current error's error code. At this time, this will always be 1, for which the constant `PERMISSION_DENIED` is defined.
No exceptions.

2.1.4.2 Constants

`PERMISSION_DENIED` of type unsigned short
The user denied the page permission to use the user's media devices.

WebIDL

```
[Callback=FunctionOnly, NoInterfaceObject]
interface NavigatorUserMediaErrorCallback {
    void handleEvent (NavigatorUserMediaError error);
};
```

2.1.4.3 Methods

`handleEvent`

(Explanation of handleEvent TBD)

Parameter	Type	Nullable	Optional	Description
error	`NavigatorUserMediaError`	✗	✗	

No exceptions.
Return type: `void`

2.2 Examples

> A voice chat feature in a game could attempt to get access to the user's microphone by calling the API as follows:
>
> ```
> <script>
> navigator.getUserMedia('audio', gotAudio);
> function gotAudio(stream) {
>  // ... use 'stream' ...
> ```

```
 }
</script>
```

A video-conferencing system would ask for both audio and video:

```
<script>
 function beginCall() {
   navigator.getUserMedia('audio,video user', gotStream);
 }
 function gotStream(stream) {
   // ... use 'stream' ...
 }
</script>
```

3. Stream API

3.1 Introduction

The `MediaStream` interface is used to represent streams of media data, typically (but not necessarily) of audio and/or video content, e.g. from a local camera or a remote site. The data from a `MediaStream` object does not necessarily have a canonical binary form; for example, it could just be "the video currently coming from the user's video camera". This allows user agents to manipulate media streams in whatever fashion is most suitable on the user's platform.

Each `MediaStream` object can represent zero or more tracks, in particular audio and video tracks. Tracks can contain multiple channels of parallel data; for example a single audio track could have nine channels of audio data to represent a 7.2 surround sound audio track.

Each track represented by a `MediaStream` object has a corresponding `MediaStreamTrack` object.

A `MediaStream` object has an input and an output. The input depends on how the object was created: a `LocalMediaStream` object generated by a `getUserMedia()` call, for instance, might take its input from the user's local camera, while a `MediaStream` created by a `PeerConnection` object will take as input the data received from a remote peer. The output of the object controls how the object is used, e.g. what is saved if the object is written to a file, what is displayed if the object is used in a `video` element, or indeed what is transmitted to a remote peer if the object is used with a `PeerConnection` object.

Each track in a `MediaStream` object can be disabled, meaning that it is muted in the object's output. All tracks are initially enabled.

A `MediaStream` can be finished, indicating that its inputs have forever stopped providing data. When a `MediaStream` object is finished, all its tracks are muted regardless of whether they are enabled or disabled.

The output of a `MediaStream` object must correspond to the tracks in its input. Muted audio tracks must be replaced with silence. Muted video tracks must be replaced with blackness.

A new `MediaStream` object can be created from a list of `MediaStreamTrack` objects using the `MediaStream()` constructor. The list of `MediaStreamTrack` objects can be the track list of another stream, a subset of the track list of a stream or a composition of `MediaStreamTrack` objects from different `MediaStream` objects.

The ability to duplicate a MediaStream, i.e. create a new MediaStream object from the track list of an existing stream, allows for greater control since separate MediaStream instances can be manipulated and consumed individually. This 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 tracks from different MediaStream objects into a new MediaStream makes it possible to, e.g., record selected tracks from a conversation involving several MediaStream objects with a single MediaStreamRecorder.

The LocalMediaStream interface is used when the user agent is generating the stream's data (e.g. from a camera or streaming it from a local video file). It allows authors to control individual tracks during the generation of the content, e.g. to allow the user to temporarily disable a local camera during a video-conference chat.

When a LocalMediaStream object is being generated from a local file (as opposed to a live audio/video source), the user agent should stream the data from the file in real time, not all at once. This reduces the ease with which pages can distinguish live video from pre-recorded video, which can help protect the user's privacy.

3.2 Interface definitions

3.2.1 MediaStream

The *MediaStream(trackList)* constructor must return a new MediaStream object with a newly generated label. A new MediaStreamTrack object is created for every unique underlying media source in *trackList* and appended to the new MediaStream object's track list according to the track ordering constraints.

A MediaStream object is said to *end* when the user agent learns that no more data will ever be forthcoming for this stream.

When a MediaStream object ends for any reason (e.g. because the user rescinds the permission for the page to use the local camera, or because the data comes from a finite file and the file's end has been reached and the user has not requested that it be looped, or because the stream comes from a remote peer and the remote peer has permanently stopped sending data, it is said to be ***finished*** . When this happens for any reason other than the stop() method being invoked, the user agent must queue a task that runs the following steps:

1. If the object's readyState attribute has the value ENDED (2) already, then abort these steps. (The stop() method was probably called just before the stream stopped for other reasons, e.g. the user clicked an in-page stop button and then the user-agent-provided stop button.)

2. Set the object's readyState attribute to ENDED (2).

3. Fire a simple event named ended at the object.

As soon as a MediaStream object is finished, the stream's tracks start outputting only silence and/or blackness, as appropriate, as defined earlier.

If the end of the stream was reached due to a user request, the task source for this task is the user interaction task source. Otherwise the task source for this task is the networking task source.

WebIDL

```
[Constructor (in MediaStreamTrackList trackList)]
interface MediaStream {
    readonly attribute DOMString            label;
    readonly attribute MediaStreamTrackList tracks;
    MediaStreamRecorder record ();
    const unsigned short LIVE = 1;
    const unsigned short ENDED = 2;
    readonly attribute unsigned short       readyState;
```

```
                attribute Function?            onended;
};
```

3.2.1.1 Attributes

`label` of type DOMString, readonly
Returns a label that is unique to this stream, so that streams can be recognized after they are sent through the `PeerConnection` API.
No exceptions.

`onended` of type Function, nullable
This event handler, of type `ended`, must be supported by all objects implementing the `MediaStream` interface.
No exceptions.

`readyState` of type unsigned short, readonly
The ***`readyState`*** attribute represents the state of the stream. It must return the value to which the user agent last set it (as defined below). It can have the following values: ***LIVE*** or ***ENDED***.

When a `MediaStream` object is created, its `readyState` attribute must be set to `LIVE` (1), unless it is being created using the `MediaStream()` constructor whose argument is a list of `MediaStreamTrack` objects whose underlying media sources will never produce any more data, in which case the `MediaStream` object must be created with its `readyState` attribute set to `ENDED` (2).

No exceptions.

`tracks` of type MediaStreamTrackList, readonly
Returns a `MediaStreamTrackList` object representing the tracks that can be enabled and disabled.

A `MediaStream` can have multiple audio and video sources (e.g. because the user has multiple microphones, or because the real source of the stream is a media resource with many media tracks). The stream represented by a `MediaStream` thus has zero or more tracks.

The ***`tracks`*** attribute must return an array host object for objects of type `MediaStreamTrack` that is *fixed length* and *read only*. The same object must be returned each time the attribute is accessed. [*WEBIDL*]

The array must contain the `MediaStreamTrack` objects that correspond to the the tracks of the stream. The relative order of all tracks in a user agent must be stable. All audio tracks must precede all video tracks. Tracks that come from a media resource whose format defines an order must be in the order defined by the format; tracks that come from a media resource whose format does not define an order must be in the relative order in which the tracks are declared in that media resource. Within these constraints, the order is user-agent defined.

No exceptions.

3.2.1.2 Methods

`record`
Begins recording the stream. The returned `MediaStreamRecorder` object provides access to the recorded data.

When the ***`record()`*** method is invoked, the user agent must return a new `MediaStreamRecorder` object associated with the stream.

No parameters.
No exceptions.
Return type: **`MediaStreamRecorder`**

3.2.1.3 Constants

`ENDED` of type unsigned short
The stream has finished (the user agent is no longer receiving or generating data, and will never receive or generate more data for this stream).

`LIVE` of type unsigned short
The stream is active (the user agent is making a best-effort attempt to receive or generate data in real time).

WebIDL

```
MediaStream implements EventTarget;
```

All instances of the `MediaStream` type are defined to also implement the EventTarget interface.

3.2.2 LocalMediaStream

WebIDL

```
interface LocalMediaStream : MediaStream {
    void stop ();
};
```

3.2.2.1 Methods

`stop`

When a `LocalMediaStream` object's *`stop()`* method is invoked, the user agent must queue a task that runs the following steps:

1. If the object's `readyState` attribute is in the `ENDED` (2) state, then abort these steps.
2. Permanently stop the generation of data for the stream. If the data is being generated from a live source (e.g. a microphone or camera), and no other stream is being generated from a live source, then the user agent should remove any active "on-air" indicator. If the data is being generated from a prerecorded source (e.g. a video file), any remaining content in the file is ignored. The stream is finished. The stream's tracks start outputting only silence and/or blackness, as appropriate, as defined earlier.
3. Set the object's `readyState` attribute to `ENDED` (2).
4. Fire a simple event named `ended` at the object.

The task source for the tasks queued for the `stop()` method is the DOM manipulation task source.

No parameters.
No exceptions.
Return type: `void`

3.2.3 MediaStreamTrack

WebIDL

```
typedef MediaStreamTrack[] MediaStreamTrackList;
```

Throughout this specification, the identifier MediaStreamTrackList is used to refer to the array of `MediaStreamTrack` type.

WebIDL

```
interface MediaStreamTrack {
    readonly attribute DOMString kind;
    readonly attribute DOMString label;
             attribute boolean   enabled;
};
```

3.2.3.1 Attributes

enabled of type boolean

The *MediaStreamTrack.enabled* attribute, on getting, must return the last value to which it was set. On setting, it must be set to the new value, and then, if the MediaStreamTrack object is still associated with a track, must enable the track if the new value is true, and disable it otherwise.

Note

Thus, after a MediaStreamTrack is disassociated from its track, its enabled attribute still changes value when set, it just doesn't do anything with that new value.

No exceptions.

kind of type DOMString, readonly

The *MediaStreamTrack.kind* attribute must return the string "audio" if the object's corresponding track is or was an audio track, "video" if the corresponding track is or was a video track, and a user-agent defined string otherwise.

No exceptions.

label of type DOMString, readonly

When a LocalMediaStream object is created, the user agent must generate a globally unique identifier string, and must initialize the object's label attribute to that string. Such strings must only use characters in the ranges U+0021, U+0023 to U+0027, U+002A to U+002B, U+002D to U+002E, U+0030 to U+0039, U+0041 to U+005A, U+005E to U+007E, and must be 36 characters long.

When a MediaStream is created to represent a stream obtained from a remote peer, the label attribute is initialized from information provided by the remote source.

When a MediaStream is created from another using the MediaStream() constructor, the label attribute is initialized to a newly generated value.

The *label* attribute must return the value to which it was initialized when the object was created.

Note

The label 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, a locally generated stream could be sent from one user to a remote peer using PeerConnection, and then sent back to the original user in the same manner, in which case the original user will have multiple streams with the same label (the locally-generated one and the one received from the remote peer).

User agents may label audio and video sources (e.g. "Internal microphone" or "External USB Webcam"). The *MediaStreamTrack.label* attribute must return the label of the object's corresponding track, if any. If the corresponding track has or had no label, the attribute must instead return the empty string.

Note

Thus the `kind` and `label` attributes do not change value, even if the `MediaStreamTrack` object is disassociated from its corresponding track.

No exceptions.

3.2.4 MediaStreamRecorder

WebIDL

```
interface MediaStreamRecorder {
    voice getRecordedData (BlobCallback? callback);
};
```

3.2.4.1 Methods

`getRecordedData`

Creates a `Blob` of the recorded data, and invokes the provided callback with that `Blob`.

When the ***`getRecordedData()`*** method is called, the user agent must run the following steps:

1. Let *callback* be the callback indicated by the method's first argument.
2. If *callback* is null, abort these steps.
3. Let *data* be the data that was streamed by the `MediaStream` object from which the `MediaStreamRecorder` was created since the creation of the `MediaStreamRecorder` object.
4. Return, and run the remaining steps asynchronously.
5. Generate a file that containing *data* in a format supported by the user agent for use in `audio` and `video` elements.
6. Let *blob* be a `Blob` object representing the contents of the file generated in the previous step. [*FILE-API*]
7. Queue a task to invoke *callback* with *blob* as its argument.

Note

The `getRecordedData()` method can be called multiple times on one `MediaStreamRecorder` object; each time, it will create a new file as if this was the first time the method was being called. In particular, the method does not stop or reset the recording when the method is called.

Parameter	Type	Nullable	Optional	Description
callback	**`BlobCallback`**	✓	✗	

No exceptions.
Return type: `voice`

3.2.5 BlobCallback

WebIDL

```
[Callback=FunctionOnly, NoInterfaceObject]
interface BlobCallback {
    void handleEvent (Blob blob);
};
```

3.2.5.1 Methods

`handleEvent`

Def TBD

Parameter	Type	Nullable	Optional	Description
blob	`Blob`	✗	✗	

No exceptions.
Return type: `void`

3.2.6 URL

Note that the following is actually only a partial interface, but ReSpec does not yet support that.

WebIDL

```
interface URL {
    static DOMString createObjectURL (MediaStream stream);
};
```

3.2.6.1 Methods

`createObjectURL`

Mints a Blob URL to refer to the given `MediaStream`.

When the ***`createObjectURL()`*** method is called with a `MediaStream` argument, the user agent must return a unique Blob URL for the given `MediaStream`. [*FILE-API*]

For audio and video streams, the data exposed on that stream must be in a format supported by the user agent for use in `audio` and `video` elements.

A ***Blob URL*** is the same as what the File API specification calls a Blob URI, except that anything in the definition of that feature that refers to `File` and `Blob` objects is hereby extended to also apply to `MediaStream` and `LocalMediaStream` objects.

Parameter	Type	Nullable	Optional	Description
stream	`MediaStream`	✗	✗	

No exceptions.
Return type: `static DOMString`

3.3 Examples

> This sample code exposes a button. When clicked, the button is disabled and the user is prompted to offer a stream. The user can cause the button to be re-enabled by providing a stream (e.g. giving the page access to the local camera) and then disabling the stream (e.g. revoking that access).

```
<input type="button" value="Start" onclick="start()" id="startBtn">
<script>
 var startBtn = document.getElementById('startBtn');
 function start() {
   navigator.getUserMedia('audio,video', gotStream);
   startBtn.disabled = true;
 }
 function gotStream(stream) {
   stream.onended = function () {
     startBtn.disabled = false;
   }
```

```
 }
</script>
```

This example allows people to record a short audio message and upload it to the server. This example even shows rudimentary error handling.

```
<input type="button" value="⬤" onclick="msgRecord()" id="recBtn">
<input type="button" value="■" onclick="msgStop()" id="stopBtn" disabled>
<p id="status">To start recording, press the ⬤ button.</p>
<script>
 var recBtn = document.getElementById('recBtn');
 var stopBtn = document.getElementById('stopBtn');
 function report(s) {
   document.getElementById('status').textContent = s;
 }
 function msgRecord() {
   report('Attempting to access microphone...');
   navigator.getUserMedia('audio', gotStream, noStream);
   recBtn.disabled = true;
 }
 var msgStream, msgStreamRecorder;
 function gotStream(stream) {
   report('Recording... To stop, press to ■ button.');
   msgStream = stream;
   msgStreamRecorder = stream.record();
   stopBtn.disabled = false;
   stream.onended = function () {
     msgStop();
   }
 }
 function msgStop() {
   report('Creating file...');
   stopBtn.disabled = true;
   msgStream.onended = null;
   msgStream.stop();
   msgStreamRecorder.getRecordedData(msgSave);
 }
 function msgSave(blob) {
   report('Uploading file...');
   var x = new XMLHttpRequest();
   x.open('POST', 'uploadMessage');
   x.send(blob);
   x.onload = function () {
     report('Done! To record a new message, press the ⬤ button.');
     recBtn.disabled = false;
   };
   x.onerror = function () {
     report('Failed to upload message. To try recording a message again, press the ⬤
button.');
     recBtn.disabled = false;
   };
 }
 function noStream() {
   report('Could not obtain access to your microphone. To try again, press the ⬤ button.');
   recBtn.disabled = false;
 }
</script>
```

This example allows people to take photos of themselves from the local video camera.

```
<article>
 <style scoped>
  video { transform: scaleX(-1); }
  p { text-align: center; }
 </style>
 <h1>Snapshot Kiosk</h1>
 <section id="splash">
  <p id="errorMessage">Loading...</p>
 </section>
 <section id="app" hidden>
  <p><video id="monitor" autoplay></video> <canvas id="photo"></canvas>
```

```
      <p><input type=button value="&#x1F4F7;" onclick="snapshot()">
     </section>
     <script>
      navigator.getUserMedia('video user', gotStream, noStream);
      var video = document.getElementById('monitor');
      var canvas = document.getElementById('photo');
      function gotStream(stream) {
        video.src = URL.getObjectURL(stream);
        video.onerror = function () {
          stream.stop();
        };
        stream.onended = noStream;
        video.onloadedmetadata = function () {
          canvas.width = video.videoWidth;
          canvas.height = video.videoHeight;
          document.getElementById('splash').hidden = true;
          document.getElementById('app').hidden = false;
        };
      }
      function noStream() {
        document.getElementById('errorMessage').textContent = 'No camera available.';
      }
      function snapshot() {
        canvas.getContext('2d').drawImage(video, 0, 0);
      }
     </script>
    </article>
```

4. Peer-to-peer connections

A `PeerConnection` allows two users to communicate directly, browser-to-browser. Communications are coordinated via a signaling channel provided by script in the page via the server, e.g. using `XMLHttpRequest`.

Calling "new `PeerConnection`(*configuration*, *signalingCallback*)" creates a `PeerConnection` object.

The *configuration* string gives the address of a STUN or TURN server to use to establish the connection. [*STUN*] [*TURN*]

The allowed formats for this string are:

"***TYPE* 203.0.113.2:3478**"
: Indicates a specific IP address and port for the server.

"***TYPE* relay.example.net:3478**""
: Indicates a specific host and port for the server; the user agent will look up the IP address in DNS.

"***TYPE* example.net**""
: Indicates a specific domain for the server; the user agent will look up the IP address and port in DNS.

The "*TYPE*" is one of:

STUN
: Indicates a STUN server

STUNS
: Indicates a STUN server that is to be contacted using a TLS session.

TURN
: Indicates a TURN server

TURNS
: Indicates a TURN server that is to be contacted using a TLS session.

The *signalingCallback* argument is a method that will be invoked when the user agent needs to send a message to the other host over the signaling channel. When the callback is invoked, convey its first argument (a string) to the other peer using whatever method is being used by the Web application to relay signaling messages. (Messages returned from the other peer are provided back to the user agent using the `processSignalingMessage()` method.)

A `PeerConnection` object has an associated ***`PeerConnection` signaling callback***, a ***`PeerConnection` ICE Agent***, a ***`PeerConnection` readiness state*** and an ***SDP Agent***. These are initialized when the object is created.

When the ***`PeerConnection()`*** constructor is invoked, the user agent must run the following steps. This algorithm has a synchronous section (which is triggered as part of the event loop algorithm). Steps in the synchronous section are marked with ⏳.

1. Let *serverConfiguration* be the constructor's first argument.
2. Let *signalingCallback* be the constructor's second argument.
3. Let *connection* be a newly created `PeerConnection` object.
4. Create an ICE Agent and let *connection*'s `PeerConnection` ICE Agent be that ICE Agent. [*ICE*]
5. If *serverConfiguration* contains a U+000A LINE FEED (LF) character or a U+000D CARRIAGE RETURN (CR) character (or both), remove all characters from *serverConfiguration* after the first such character.
6. Split *serverConfiguration* on spaces to obtain *configuration components*.
7. If *configuration components* has two or more components, and the first component is a case-sensitive match for one of the following strings:
   - "`STUN`"
   - "`STUNS`"
   - "`TURN`"
   - "`TURNS`"

   ...then run the following substeps:

   1. Let *server type* be STUN if the first component of *configuration components* is '`STUN`' or '`STUNS`', and TURN otherwise (the first component of *configuration components* is "`TURN`" or "`TURNS`").
   2. Let *secure* be true if the first component of *configuration components* is "`STUNS`" or "`TURNS`", and false otherwise.
   3. Let *host* be the contents of the second component of *configuration components* up to the character before the first U+003A COLON character (:), if any, or the entire string otherwise.
   4. Let *port* be the contents of the second component of *configuration components* from the character after the first U+003A COLON character (:) up to the end, if any, or the empty string otherwise.
   5. Configure the `PeerConnection` ICE Agent's STUN or TURN server as follows:
      - If *server type* is STUN, the server is a STUN server. Otherwise, *server type* is TURN and the server is a TURN server.
      - If *secure* is true, the server is to be contacted using TLS-over-TCP, otherwise, it is to be contacted using UDP.
      - The IP address, host name, or domain name of the server is *host*.
      - The port to use is *port*. If this is the empty string, then only a domain name is configured (and the ICE Agent will use DNS SRV requests to determine the IP address and port).
      - The long-term username for the STUN or TURN server is the ASCII serialization of the entry script's origin; the long-term password is the empty string.

If the given IP address, host name, domain name, or port are invalid, then the user agent must act as if no STUN or TURN server is configured.

8. Let the *connection*'s `PeerConnection` signaling callback be *signalingCallback*.
9. Set *connection*'s `PeerConnection` readiness state to `NEW` (0).
10. Set *connection*'s `PeerConnection` ice state to `NEW` (0).
11. Set *connection*'s `PeerConnection` sdp state to `NEW` (0).
12. Let *connection*'s `localStreams` attribute be an empty read-only `MediaStream` array. [*WEBIDL*]
13. Let *connection*'s `remoteStreams` attribute be an empty read-only `MediaStream` array. [*WEBIDL*]
14. Return *connection*, but continue these steps asynchronously.
15. Await a stable state. The synchronous section consists of the remaining steps of this algorithm. (Steps in synchronous sections are marked with ⏳.)
16. ⏳ If the ice state is set to NEW, it must queue a task to start gathering ICE address and set the ice state to ICEGATHERING.
17. ⏳ Once the ICE address gathering is complete, if there are any streams in localStreams, the SDP Agent will send the initial the SDP offer. The initial SDP offer `MUST` contain both the ICE candidate information as well as the SDP to represent the media descriptions for all the streams in localStreams.

During the lifetime of the peerConnection object, the following procedures are followed:

1. If a local media stream has been added and an SDP offer needs to be sent, and the ICE state is not NEW or ICEGATHERING, and the SDP Agent state is NEW or SDPIDLE, then send and queue a task to send an SDP offer and change the SPD state to SDP Waiting.
2. If an SDP offer has been received, and the SDP state is NEW or SDPIDLE, pass the ICE candidates from the SDP offer to the ICE Agent and change it state to ICECHECKING. Construct an appropriate SDP answer, update the remote streams, queue a task to send the SDP offer, and set the SDPAgent state to SDPIDLE.
3. At the point the sdpState changes from NEW to some other state, the readyState changes to NEGOTIATING.
4. If the ICE Agent finds a candidates that froms a valid connection, the ICE state is changed to ICECONNECTED
5. If the ICE Agent finishes checking all candidates, if a connection has been found, the ice state is changed to ICECOMPLETED and if not connection has been found it is changed to ICEFAILED.
6. If the iceState is ICECONNECTED or ICECOMPLETED and the SDP stat is SDPIDLE, the readyState is set to ACTIVE.
7. If the iceState is ICEFAILED, a task is queued to calls the close method.
8. The close method will cause the system to wait until the sdpStat is SDPIDLE then it will send an SDP offer terminating all media and change the readyState to CLOSING as well as stop all ICE process and change the iceState to ICE_CLOSED. Once an SDP anser to this offer is received, the readyState will be changed to CLOSED.

User agents may negotiate any codec and any resolution, bitrate, or other quality metric. User agents are encouraged to initially negotiate for the native resolution of the stream. For streams that are then rendered (using a `video` element), user agents are encouraged to renegotiate for a resolution that matches the rendered display size.

Note

Starting with the native resolution means that if the Web application notifies its peer of the native resolution as it starts sending data, and the peer prepares its `video` element accordingly, there will be no need for a renegotiation once the stream is flowing.

All SDP media descriptions for streams represented by `MediaStream` objects must include a label attribute ("`a=label:`") whose value is the value of the `MediaStream` object's `label` attribute. [*SDP*] [*SDPLABEL*]

`PeerConnection`s must not generate any candidates for media streams whose media descriptions do not have a label attribute ("`a=label:`"). [*ICE*] [*SDP*] [*SDPLABEL*]

When a user agent starts receiving media for a component and a candidate was provided for that component by a `PeerConnection`, the user agent must follow these steps:

1. Let *connection* be the `PeerConnection` expecting this media.
2. If there is already a `MediaStream` object for the media stream to which this component belongs, then associate the component with that media stream and abort these steps. (Some media streams have multiple components; this API does not expose the role of these individual components in ICE.)
3. Create a `MediaStream` object to represent the media stream. Set its `label` attribute to the value of the SDP Label attribute for that component's media stream.
4. Queue a task to run the following substeps:
    1. If the *connection*'s `PeerConnection` readiness state is `CLOSED` (3), abort these steps.
    2. Add the newly created `MediaStream` object to the end of *connection*'s `remoteStreams` array.
    3. Fire a stream event named `addstream` with the newly created `MediaStream` object at the *connection* object.

When a `PeerConnection` finds that a stream from the remote peer has been removed (its port has been set to zero in a media description sent on the signaling channel), the user agent must follow these steps:

1. Let *connection* be the `PeerConnection` associated with the stream being removed.
2. Let *stream* be the `MediaStream` object that represents the media stream being removed, if any. If there isn't one, then abort these steps.
3. By definition, *stream* is now finished.

    Note

    A task is thus queued to update *stream* and fire an event.

4. Queue a task to run the following substeps:
    1. If the *connection*'s `PeerConnection` readiness state is `CLOSED` (3), abort these steps.
    2. Remove *stream* from *connection*'s `remoteStreams` array.
    3. Fire a stream event named `removestream` with *stream* at the *connection* object.

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

To prevent network sniffing from allowing a fourth party to establish a connection to a peer using the information sent out-of-band to the other peer and thus spoofing the client, the configuration information should always be transmitted using an encrypted connection.

4.1 PeerConnection

WebIDL

```
[Constructor (in DOMString configuration, in SignalingCallback signalingCallback)]
interface PeerConnection {
    void processSignalingMessage (DOMString message);
    const unsigned short NEW = 0;
    const unsigned short NEGOTIATING = 1;
    const unsigned short ACTIVE = 2;
    const unsigned short CLOSING = 4;
    const unsigned short CLOSED = 3;
    readonly attribute unsigned short readyState;
    const unsigned short ICE_GATHERING = 0x100;
    const unsigned short ICE_WAITING = 0x200;
    const unsigned short ICE_CHECKING = 0x300;
    const unsigned short ICE_CONNECTED = 0x400;
    const unsigned short ICE_COMPLETED = 0x500;
    const unsigned short ICE_FAILED = 0x600;
    const unsigned short ICE_CLOSED = 00x700;
    readonly attribute unsigned short iceState;
    const unsigned short SDP_IDLE = 0x1000;
    const unsigned short SDP_WAITING = 0x2000;
    const unsigned short SDP_GLARE = 0x3000;
    readonly attribute unsigned short sdpState;
    void addStream (MediaStream stream, MediaStreamHints hints);
    void removeStream (MediaStream stream);
    readonly attribute MediaStream[]  localStreams;
    readonly attribute MediaStream[]  remoteStreams;
    void close ();
             attribute Function?      onconnecting;
             attribute Function?      onopen;
             attribute Function?      onstatechange;
             attribute Function?      onaddstream;
             attribute Function?      onremovestream;
};
```

4.1.1 Attributes

`iceState` of type unsigned short, readonly

The *`iceState`* attribute must return the state of the `PeerConnection` ICE Agent `PeerConnection` ICE state, represented by a number from the following list:

`PeerConnection` . **`NEW`** (0)
The object was just created, and no networking has yet occurred.

`PeerConnection` . **`ICE_GATHERING`** (0x100)
The ICE Agent is attempting to establish a gather addresses.

`PeerConnection` . **`ICE_WAITING`** (0x200)
The ICE Agent is waiting for candidates from the other side before it can start checking.

`PeerConnection` . **`ICE_CHECKING`** (0x300)
The ICE Agent is checking candidates but has not yet found a connection.

`PeerConnection` . **`ICE_CONNECTED`** (0x400)
The ICE Agent has found a connection but is still checking other candidates to see if there is a better connection.

`PeerConnection` . **`ICE_COMPLETED`** (0x500)
The ICE Agent has finished checking and found a connection.

`PeerConnection` . **`ICE_FAILED`** (0x600)
The ICE Agent is finished checking all candidates and failed to find a connection.

`PeerConnection` . `ICE_CLOSED` (0x700)
The ICE Agent has shut down and is no longer responding to STUN requests.

No exceptions.

`localStreams` of type array of *MediaStream*, readonly
Returns a live array containing the streams that the user agent is currently attempting to transmit to the remote peer (those that were added with `addStream()`).

Specifically, it must return the read-only `MediaStream` array that the attribute was set to when the `PeerConnection`'s constructor ran.

No exceptions.

`onaddstream` of type Function, nullable
This event handler, of event handler event type `addstream`, must be supported by all objects implementing the `PeerConnection` interface.
No exceptions.

`onconnecting` of type Function, nullable
This event handler, of event handler event type `connecting`, must be supported by all objects implementing the `PeerConnection` interface.
No exceptions.

`onopen` of type Function, nullable
This event handler, of event handler event type `open`, must be supported by all objects implementing the `PeerConnection` interface.
No exceptions.

`onremovestream` of type Function, nullable
This event handler, of event handler event type `removestream`, must be supported by all objects implementing the `PeerConnection` interface.
No exceptions.

`onstatechange` of type Function, nullable
This event handler, of event handler event type `open`, must be supported by all objects implementing the `PeerConnection` interface. It is called any time the readyState, iceState, or sdpState changes.
No exceptions.

`readyState` of type unsigned short, readonly
The *`readyState`* attribute must return the `PeerConnection` object's `PeerConnection` readiness state, represented by a number from the following list:

`PeerConnection` . `NEW` (0)
The object was just created, and no networking has yet occurred.

`PeerConnection` . `NEGOTIATING` (1)
The user agent is attempting to establish an connection with the ICE Agent and to negotiate codecs with the SDP Agent.

`PeerConnection` . `ACTIVE` (2)
The ICE Agent has found a connection the SDP Agent has performed a round of codec negotiation. It is possible for whatever media was negotiated to flow.

`PeerConnection` . `CLOSING` (4)
The `PeerConnection` object is terminating all media and is in the process of closing the Ice Agent and SDP Agent.

`PeerConnection` . `CLOSED` (3)
The connection is closed.

No exceptions.

remoteStreams of type array of *MediaStream*, readonly

Returns a live array containing the streams that the user agent is currently receiving from the remote peer.

Specifically, it must return the read-only MediaStream array that the attribute was set to when the PeerConnection's constructor ran.

This array is updated when addstream and removestream events are fired.

No exceptions.

sdpState of type unsigned short, readonly

The *sdpState* attribute must return the state of the PeerConnection SDP Agent , represented by a number from the following list:

PeerConnection . **NEW** (0)

The object was just created, and no networking has yet occurred.

PeerConnection . **SDP_IDLE** (0x1000)

At least one SDP offer or answer has been exchange and the SDP Agent is ready to send an SDP offer or receive an SDP answer.

PeerConnection . **SDP_WAITING** (0x2000)

The SDP Agent has sent and offer and is waiting for a answer.

PeerConnection . **SDP_GLARE** (0x3000)

The SDP Agent received an offer while waiting for an answer and now much wait a rondom amount of time before retrying to send the offer.

No exceptions.

4.1.2 Methods

addStream

Attempts to starting sending the given stream to the remote peer. The format for the MediaStreamHints objects is currently undefined by the specification.

When the other peer starts sending a stream in this manner, an addstream event is fired at the PeerConnection object.

When the *addStream()* method is invoked, the user agent must run the following steps:

1. Let *stream* be the method's first argument.
2. Let *hints* be the method's second argument.
3. If the PeerConnection object's PeerConnection readiness state is CLOSED (3), throw an INVALID_STATE_ERR exception.
4. If *stream* is already in the PeerConnection object's localStreams object, then abort these steps.
5. Add *stream* to the end of the PeerConnection object's localStreams object.
6. Return from the method.
7. Parse the *hints* provided by the application and apply them to the MediaStream, if possible.

8. Have the `PeerConnection` add a media stream for *stream* the next time the user agent provides a stable state. Any other pending stream additions and removals must be processed at the same time.

Parameter	Type	Nullable	Optional	Description
stream	`MediaStream`	✗	✗	
hints	`MediaStreamHints`	✗	✗	

No exceptions.
Return type: `void`

`close`

When the ***`close()`*** method is invoked, the user agent must run the following steps:

1. If the `PeerConnection` object's `PeerConnection` readiness state is `CLOSED` (3), throw an `INVALID_STATE_ERR` exception.

2. Destroy the `PeerConnection` ICE Agent, abruptly ending any active ICE processing and any active streaming, and releasing any relevant resources (e.g. TURN permissions).

3. Set the object's `PeerConnection` readiness state to `CLOSED` (3).

Note

The `localStreams` and `remoteStreams` objects remain in the state they were in when the object was closed.

No parameters.
No exceptions.
Return type: `void`

`processSignalingMessage`

When a message is relayed from the remote peer over the signaling channel is received by the Web application, pass it to the user agent by calling the `processSignalingMessage()` method.

The order of messages is important. Passing messages to the user agent in a different order than they were generated by the remote peer's user agent can prevent a successful connection from being established or degrade the connection's quality if one is established.

When the ***`processSignalingMessage()`*** method is invoked, the user agent must run the following steps:

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

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

3. If *connection*'s `PeerConnection` readiness state is `CLOSED` (3), throw an `INVALID_STATE_ERR` exception.

4. If the first four characters of *message* are not "`SDP`" followed by a U+000A LINE FEED (LF) character, then abort these steps. (This indicates an error in the signaling channel implementation. User agents may report such errors to their developer consoles to aid debugging.)

   Note

   Future extensions to the `PeerConnection` interface might use other prefix values to implement additional features.

5. Let *sdp* be the string consisting of all but the first four characters of *message*.

6. Pass the *sdp* to the `PeerConnection` SDP Agent as a subsequent offer or answer, to be interpreted as appropriate given the current state of the SDP Agent. [*ICE*]

When a `PeerConnection` ICE Agent forms a connection to the the far side and enters the state ICECONNECTED, the user agent must queue a task that sets the `PeerConnection` object's `PeerConnection` readiness state to `ACTIVE` (2) and then fires a simple event named `open` at the `PeerConnection` object.

Parameter	Type	Nullable	Optional	Description
message	`DOMString`	✗	✗	

No exceptions.
Return type: `void`

`removeStream`
Stops sending the given stream to the remote peer.

When the other peer stops sending a stream in this manner, a `removestream` event is fired at the `PeerConnection` object.

When the ***`removeStream()`*** method is invoked, the user agent must run the following steps:

1. Let *stream* be the method's argument.
2. If the `PeerConnection` object's `PeerConnection` readiness state is `CLOSED` (3), throw an `INVALID_STATE_ERR` exception.
3. If *stream* is not in the `PeerConnection` object's `localStreams` object, then abort these steps.
4. Remove *stream* from the `PeerConnection` object's `localStreams` object.
5. Return from the method.
6. Have the `PeerConnection`remove the media stream for *stream* the next time the user agent provides a stable state. Any other pending stream additions and removals must be processed at the same time.

Parameter	Type	Nullable	Optional	Description
stream	**`MediaStream`**	✗	✗	

No exceptions.
Return type: `void`

4.1.3 Constants

`ACTIVE` of type unsigned short
A connection has been formed and if any media streams were successfully negotiated, any relevant media can be streaming.

`CLOSED` of type unsigned short
The `close()` method has been invoked.

`CLOSING` of type unsigned short
The object is starting to shut down after the `close()` method has been invoked.

`ICE_CHECKING` of type unsigned short
The ICE Agent is checking candidates but has not yet found a connection that works.

`ICE_CLOSED` of type unsigned short
The ICE Agent is terminating and will no longer repined to STUN connectivity checks.

`ICE_COMPLETED` of type unsigned short
The ICE Agent has finished checking all candidates and a connection has been formed.

`ICE_CONNECTED` of type unsigned short
The ICE Agent has found at least one candidate that works but is still checking.

`ICE_FAILED` of type unsigned short
The ICE Agent has finished checking all candidates and no connection was worked.

`ICE_GATHERING` of type unsigned short
The ICE Agent is gather addresses that can be used.

`ICE_WAITING` of type unsigned short
THE ICE Agent has complete gathering addresses and is waiting for candidates to start checking.

`NEGOTIATING` of type unsigned short
The peerConenction object is attempting to get to the point wehre media can flow.

`NEW` of type unsigned short
The object was just created and its ICE and SDP Agent have not yet been started.

`SDP_GLARE` of type unsigned short
Both side sent SDP offers at the same time and the SDP Agent is waiting to be able to retransmit the SDP offer.

`SDP_IDLE` of type unsigned short
A valid offer anser pair has been exchanged and the SDP Agent is waiting for the next SDP transaction.

`SDP_WAITING` of type unsigned short
The SDP Agent has sent an SDP offer and is waiting for a response.

WebIDL

```
PeerConnection implements EventTarget;
```

All instances of the `PeerConnection` type are defined to also implement the EventTarget interface.

4.2 SignalingCallback

WebIDL

```
[Callback=FunctionOnly, NoInterfaceObject]
interface SignalingCallback {
    void handleEvent (DOMString message, PeerConnection source);
};
```

4.2.1 Methods

`handleEvent`
Def TBD

Parameter	Type	Nullable	Optional	Description
message	`DOMString`	✗	✗	
source	`PeerConnection`	✗	✗	

No exceptions.
Return type: `void`

4.3 Examples

> 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.
>
> ```
> // the first argument describes the STUN/TURN server configuration
> var local = new PeerConnection('TURNS example.net', sendSignalingChannel);
> local.signalingChannel(...); // if we have a message from the other side, pass it along here
>
> // (aLocalStream is some LocalMediaStream object)
> local.addStream(aLocalStream); // start sending video
>
> function sendSignalingChannel(message) {
>  ... // send message to the other side via the signaling channel
> }
>
> function receiveSignalingChannel (message) {
>  // call this whenever we get a message on the signaling channel
>  local.signalingChannel(message);
> }
>
> local.onaddstream = function (event) {
>  // (videoElement is some <video> element)
>  videoElement.src = URL.getObjectURL(event.stream);
> };
> ```

5. The data stream

Note

Although progress is being made, there is currently not enough agreement on the data channel to write it up. This section will be filled in as rough consensus is reached.

6. Garbage collection

A `Window` object ***has a strong reference*** to any `PeerConnection` objects created from the constructor whose global object is that `Window` object.

7. Event definitions

The `addstream` and `removestream` events use the `MediaStreamEvent` interface:

7.1 MediaStreamEvent

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

WebIDL

```
interface MediaStreamEvent : Event {
    readonly attribute MediaStream? stream;
    void initMediaStreamEvent (DOMString typeArg, boolean canBubbleArg, boolean cancelableArg,
MediaStream? streamArg);
};
```

7.1.1 Attributes

`stream` of type *MediaStream*, readonly, nullable

The *`stream`* attribute represents the `MediaStream` object associated with the event.

No exceptions.

7.1.2 Methods

`initMediaStreamEvent`

The *`initMediaStreamEvent()`* method must initialize the event in a manner analogous to the similarly-named method in the DOM Events interfaces. [*DOM-LEVEL-3-EVENTS*]

Parameter	Type	Nullable	Optional	Description
typeArg	`DOMString`	✗	✗	
canBubbleArg	`boolean`	✗	✗	
cancelableArg	`boolean`	✗	✗	
streamArg	`MediaStream`	✓	✗	

No exceptions.
Return type: `void`

8. Event summary

This section is non-normative.

The following event fires on `MediaStream` objects:

Event name	Interface	Fired when...
`ended`	`Event`	The `MediaStream` object will no longer stream any data, either because the user revoked the permissions, or because the source device has been ejected, or because the remote peer stopped sending data, or because the `stop()` method was invoked.

The following events fire on `PeerConnection` objects:

Event name	Interface	Fired when...
`connecting`	`Event`	The ICE Agent has begun negotiating with the peer. This can happen multiple times during the lifetime of the `PeerConnection` object.
`open`	`Event`	The ICE Agent has finished negotiating with the peer.
`message`	`MessageEvent`	A data UDP media stream message (to be defined) was received.
`addstream`	`MediaStreamEvent`	A new stream has been added to the `remoteStreams` array.
`removestream`	`MediaStreamEvent`	A stream has been removed from the `remoteStreams` array.

9. application/html-peer-connection-data

This registration is for community review and will be submitted to the IESG for review, approval, and registration with IANA.

Type name:
application

Subtype name:
html-peer-connection-data

Required parameters:
No required parameters

Optional parameters:
No optional parameters

Encoding considerations:
This MIME type defines a binary protocol format which uses UTF-8 for text encoding.

Security considerations:
This format is used for encoding UDP packets transmitted by potentially hostile Web page content via a trusted user agent to a destination selected by a potentially hostile remote server. To prevent this mechanism from being abused for cross-protocol attacks, all the data in these packets is masked so as to appear to be random noise. The intent of this masking is to reduce the potential attack scenarios to those already possible previously.

However, this feature still allows random data to be sent to destinations that might not normally have been able to receive them, such as to hosts within the victim's intranet. If a service within such an intranet cannot handle receiving UDP packets containing random noise, it might be vulnerable to attack from this feature.

Interoperability considerations:
Rules for processing both conforming and non-conforming content are defined in this specification.

Published specification:
This document is the relevant specification.

Applications that use this media type:
This type is only intended for use with SDP. [*SDP*]

Additional information:

Magic number(s):
No sequence of bytes can uniquely identify data in this format, as all data in this format is intentionally masked to avoid cross-protocol attacks.

File extension(s):
This format is not for use with files.

Macintosh file type code(s):
This format is not for use with files.

Person & email address to contact for further information:
Daniel C. Burnett <dburnett@voxeo.com>

Intended usage:
Common

Restrictions on usage:
No restrictions apply.

Author:
Daniel C. Burnett <dburnett@voxeo.com>

Change controller:
W3C

Fragment identifiers cannot be used with `application/html-peer-connection-data` as URLs cannot be used to identify streams that use this format.

A. Acknowledgements

The editors wish to thank the Working Group chairs, Harald Alvestrand and Stefan Håkansson, for their support.

B. References

B.1 Normative references

[DOM-LEVEL-3-EVENTS]
Björn Höhrmann; Tom Pixley; Philippe Le Hégaret. *Document Object Model (DOM) Level 3 Events Specification.* 31 May 2011. W3C Working Draft. (Work in progress.) URL: http://www.w3.org/TR/2011/WD-DOM-Level-3-Events-20110531/

[FILE-API]
Arun Ranganathan. *File API.* 17 November 2009. W3C Working Draft. (Work in progress.) URL: http://www.w3.org/TR/2009/WD-FileAPI-20091117/

[ICE]
J. Rosenberg *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols.* April 2010. Internet RFC 5245. URL: http://tools.ietf.org/html/rfc5245

[SDP]
J. Rosenberg, H. Schulzrinne. *An Offer/Answer Model with the Session Description Protocol (SDP).* June 2002. Internet RFC 3264. URL: http://tools.ietf.org/html/rfc3264

[SDPLABEL]
O. Levin, G. Camarillo. *The Session Description Protocol (SDP) Label Attribute.* August 2006. Internet RFC 4574. URL: http://tools.ietf.org/html/rfc4574

[STUN]
J. Rosenberg, R. Mahy, P. Matthews, D. Wing. *Session Traversal Utilities for NAT (STUN).* October 2008. Internet RFC 5389. URL: http://tools.ietf.org/html/rfc5389

[TURN]
P. Mahy, P. Matthews, J. Rosenberg. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN).* April 2010. Internet RFC 5766. URL: http://tools.ietf.org/html/rfc5766

[WEBIDL]
Cameron McCormack. *Web IDL.* 19 December 2008. W3C Working Draft. (Work in progress.) URL: http://www.w3.org/TR/2008/WD-WebIDL-20081219

B.2 Informative references

No informative references.