W3C WebRTC WG Meeting

June 11, 2019
8:30 AM Pacific Time

Chairs: Bernard Aboba
Harald Alvestrand
Jan-Ivar Bruaroey

W3C WG IPR Policy

- This group abides by the W3C Patent Policy https://www.w3.org/Consortium/Patent-Policy/
- Only people and companies listed at https://www.w3.org/2004/01/pp-impl/47318/status are allowed to make substantive contributions to the WebRTC specs

Welcome!

- Welcome to the interim meeting of the W3C WebRTC WG!
  - During this meeting, we hope to make progress on open issues in webrtc-ice, media capture and webrtc-nv use cases as well as to decide on proposals for substantive changes to webrtc-pc.
- Introducing our new WebRTC WG co-chair, Jan-Ivar!

About this Virtual Meeting

Information on the meeting:

- Meeting info:
  - https://www.w3.org/2011/04/webrtc/wiki/June_11_2019
- Link to latest drafts:
  - https://w3c.github.io/mediacapture-main/
  - https://w3c.github.io/mediacapture-output/
  - https://w3c.github.io/mediacapture-screen-share/
  - https://w3c.github.io/mediacapture-record/
  - https://w3c.github.io/webrtc-pc/
  - https://w3c.github.io/webrtc-stats/
  - https://www.w3.org/TR/mst-content-hint/
  - https://w3c.github.io/webrtc-nv-use-cases/
  - https://w3c.github.io/webrtc-dscp-exp/
  - https://github.com/w3c/webrtc-svc
  - https://github.com/w3c/webrtc-ice
- Link to Slides has been published on WG wiki
- Scribe? IRC http://irc.w3.org/ Channel: #webrtc
- The meeting is being recorded.

For Discussion Today

- **WebRTC-ICE**
  - **PR 22: Add initial FlexICE methods and attributes (pthatcher)**
- **Media Capture**
  - **Issue 551: Should we allow empty string device IDs? (youenn)**
- **WebRTC-NV Use Cases**
  - PRs
    - **PR 21: Requirements for Secure Web Conferencing (bernard)**
    - **Issue 5/PR 32: N03: “Bandwidth Limit Across Mesh Endpoints” requires clarification or modification (Harald)**
  - Issues
    - **Issue 15: Distributed hash tables (feross/lgrahl)**

For Discussion Today (cont'd)

- **WebRTC-PC**
  - PRs
    - **PR 2175: Permission API for receive-only media and data use cases (lgrahl)**
    - **(WebRTC-NV) PR 14: N29: A way to obtain user consent for one-way media and data use cases (lgrahl)**
    - **Issue 2167/PR 2169: {iceRestart: true} works poorly with negotiationneeded (jan-ivar)**
  - Issues
    - **Issue 2165: A simpler glare-proof SLD() (jan-ivar)**
    - **Issue 2166: A simpler non-racy rollback (jan-ivar)**
    - **Issue 2176: Spec steps on stop() underestimates BUNDLE problem (jan-ivar)**

WebRTC-ICE Issues

- PR 22: Add initial FlexICE methods and attributes (pthatcher)

PR 22: Add initial FlexICE methods and attributes (pthatcher)

- I (Peter) still haven't done the work to get feedback from developers who might need this.
- Given that we don't have such feedback yet, is anyone interested in implementing any part of this?

Media Capture Issues

- Issue 551: Should we allow empty string device IDs? (youenn)
  - PR 587: Clarify that some device ids may not be origin-unique (youenn)
  - PR 595: Interpret the empty string as if the constraint is not specified (youenn)

PR 587: Clarify that some device ids may not be origin-unique (youenn)

- Chrome exposes device ids ‘default’ values
- Safari exposes device ids ‘’ values
  - As long as a page does not have ‘device-info’ permission
- Specification requires device ids to be origin-unique identifiers
  - To prevent user tracking
- ‘default’ and ‘’ do not create any cross-origin information leakage
- Proposal
  - Identifiers SHOULD be origin-unique, per device type
  - An identifier can be reused across origins as long as it is not tied to the user

PR 595: Interpret the empty string as if the constraint is not specified (youenn)

- Safari treats ‘’ for deviceId constraint as if constraint is not specified
  - For compatibility with existing web pages that call first enumerateDevices, then getUserMedia with a deviceId value given by enumerateDevices
- Proposal
  - Interpret the empty string as if the constraint is not specified
    - Similarly to the special case of an empty array
  - Would apply to any DOMString constraint
    - facingMode, resizeMode, deviceId, groupId
  - PR 595 scopes it down to ‘’ specifically
    - No specific meaning to [‘’], or [‘’, ‘deviceId2’]

WebRTC-NV Use Cases

- PRs
  - PR 21: Requirements for Secure Web Conferencing (bernard)
  - Issue 5/PR 32: N03: “Bandwidth Limit Across Mesh Endpoints” requires clarification or modification (Harald)
- Issues
  - Issue 15: Distributed hash tables (Igrahl)

PR 21: Requirements for Secure Web Conferencing (bernard)

- Submitted by Cullen Jennings.
- Distinguishes two use-cases: Untrusted JS and Trusted JS.

```
+ <section id="no-trust-webex">
+ <h3>Don't Pown My Video Conferencing </h3>
+
+ <p>Cloud video conferencing system such as WebEx, Hangouts, and Zoom
+   have no need to be able to see all the content of the media flowing
+   over their media servers. Some of these conferencing services desire
+   to be able to promote trust by explicitly showing they do not have
+   access to the audio and video contents of their users' calls. They are
+   trusted to connect the right people to the conference and route the
+   packets but they are not trusted to see the contents of the audio and
+   video media in the call.</p>
+
+ <p>Solutions to this problem fall into two major categories: ones where
+   the JavaScript comes from a source trusted to see the media contents,
+   and ones where it does not:</p>
```

PR 21: Requirements for Secure Web Conferencing (cont'd)

```
289 + <h4>Untrusted JS Cloud Conferencing</h4>
290 +
291 + <p>There are many cases where a system such as WebEx is trusted to
292 +   connect the members of a conference but has no need to access the
293 +   audio or video contents of the conference. This is true of the
294 +   majority of audio-video conferencing systems on the web today. Just to
295 +   highlight the scope of this requirement, there are more minutes of
296 +   WebRTC that are used in conferences where the media server has no need
297 +   of access to the media than any other use of WebRTC audio by orders of
298 +   magnitude. This is the primary use case for WebRTC audio at this point
299 +   in time and accounts for billions of minute per month of potential use
300 +   of WebRTC.</p>
301 +
302 + <p>In this use case, the JS comes from the operator of the conference
303 +   bridge. The WebRTC use the isolated media features of WebRTC to keep
304 +   the JS from accessing the media and the identity features to provide a
305 +   user interface that allows the user to know it connected to the
306 +   correct conferences. The goal is for the end users to be able to see
307 +   the audio and video media contents, but the web service that provides
308 +   the JS and the media switching bridges and SFUs cannot see the
309 +   media. Some metadata about the audio, such as the power levels of the
310 +   audio media, are encrypted to the media server.</p>
311 +
312 + <p>A possible solution this problem is the browser to negotiate end to
313 +   end encryption keys using approach such as the IETF PERC work. The end
314 +   to end keys are not revealed to the JavaScript.</p>
```

PR 21: Requirements for Secure Web Conferencing (cont'd)

```
316 + <h4>Trusted JS</h4>
317 +
318 + <p>In some highly controlled situation such as the department of
319 +   defence, the JavaScript can come from a trusted source while the media
320 +   server the distributes the packets is not trusted to see the contents
321 +   of the conference. The media server is trusted to forward packets
322 +   appropriate and not DOS the conferences.</p>
323 +
324 + <p>In this case, the JavaScript running the WebRTC applications comes
325 +   from a trusted source and is allowed to access to the contents of the
326 +   media. These use cases tend to be relatively small deployments as they
327 +   require the JavaScript come from a trusted source but still trying to
328 +   use a media server from an untrusted source. The media needs to be
329 +   encrypted with keys that are not known to the operator of the media
330 +   server, then sent to the media server which deals with
331 +   conferencing. It is acceptable for the power levels of the audio
332 +   content to be revealed to the media server even though this reveals
333 +   some information about the content.</p>
334 +
335 + <pre class="highlight">
336 +    ----------------------------------------------------------------
337 +    REQ-ID          DESCRIPTION
338 +    ----------------------------------------------------------------
339 +    N??             TBD
340 + </pre>
```

PR 21: Requirements for Secure Web Conferencing (cont'd)

- Discuss.

Issue 5/PR 32: N03: “Bandwidth Limit Across Mesh Endpoints” requires clarification or modification (Harald)

- N03 is currently phrased as “The user agent must be able to impose a bandwidth limit across mesh endpoints.”
- It’s completely unclear to this reader what this is supposed to mean.
- The justification is “Also, the ability to impose a bandwidth limit across all mesh endpoints limits the build up of queues that can affect audio quality or perceived latency in the game.”
- Experience with congestion control indicates that app-imposed bandwidth limits are not the proper way to achieve this - the app doesn’t know enough about network state.
- Reformulation suggested in PR 32: “Congestion control must be able to manage audio quality and latency in a fair manner between multiple connections.”
- Principle: Requirements should say what you want to achieve, not how to do it.

Issue 15: Distributed hash tables (feross/lgrahl)

- DHTs provide a way for millions of computers to self-organize into a network, communicate with other computers in the network, and share resources (e.g. files, blobs, objects) between computers, without the use of a central registry.
- Used in nearly every decentralized protocol: BitTorrent, IPFS, Dat, Ethereum, etc.
- One of the most beautiful ideas in Distributed Systems (and maybe all of CS)
- Despite no central coordination, DHTs offer deterministic lookup time which is **O(log(n))** in the number of peers in the network.
- See academic paper: *Maymounkov P., Mazières D. (2002) Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*

Kademlia Properties

- Self-organizing
- For a network with 10,000,000 peers, only ~20 hops are necessary for communication with any subset of peers.

RPC Messages

- **STORE**: Instructs a peer to store a key-value pair
- **FIND_NODE**: Returns information about the ***k*** peers closest to the target id (ip:port)
- **FIND_VALUE**: Similar to FIND_NODE, but if the recipient has received a STORE for the given key, it returns the corresponding value.

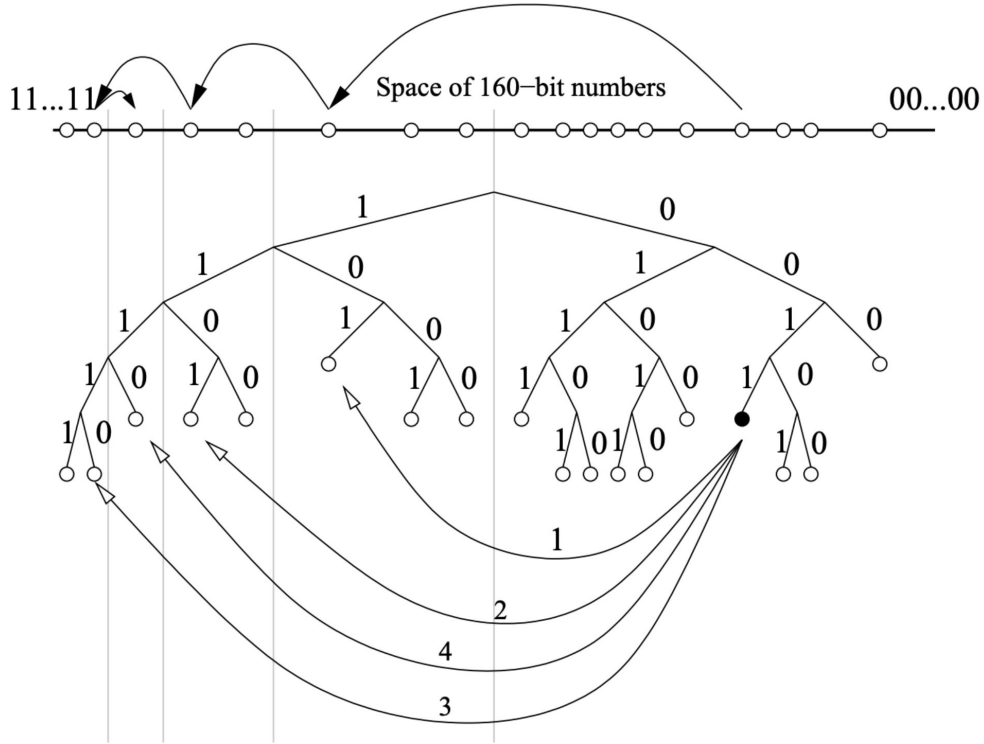


Fig. 2: Locating a node by its ID. Here the node with prefix 0011 finds the node with prefix 1110 by successively learning of and querying closer and closer nodes. The line segment on top represents the space of 160-bit IDs, and shows how the lookups converge to the target node. Below we illustrate RPC messages made by 1110. The first RPC is to node 101, already known to 1110. Subsequent RPCs are to nodes returned by the previous RPC.

Limitations of WebRTC for DHT Use Case

- **WebRTC connection model makes DHT lookup impractical**
  - Instead of replying to FIND_NODE queries with connection information ("ip:port" string), peers must act as signaling server to setup the connection.
  - Peers need to remain actively connected to any peer they wish to let other peers know about, because there is no way to get an offer, disconnect from a peer, and later reconnect (since there is no way to send answer).
- **WebRTC connections are slow**
  - Many roundtrips required to setup a connection, which makes the DHT's iterative lookup very slow.
  - High CPU usage, memory, and hitting browser limits that make opening more connections fail or even cause a browser crash.

Issue 15: Distributed hash tables (feross/lgrahl)

- **Requirement: Listen/Multiplex**
  - The application must be able to post "long-lasting" signalling data once and multiplex incoming peer connections.
  - The application must be able to accept incoming peer connections.
  - The above requirements must be usable in worker threads.
- **Requirement: Efficiency**
  - The UA must be able to handle many peer-to-peer connections at the same time efficiently.
- **Requirement: Connection Setup Time**
  - The amount of RTTs required to set up a peer-to-peer connection should be minimised.

Issue 15: Distributed hash tables (feross/lgrahl)

- **"Long-shot" Controversial Extra Requirement:** Reusable Connections
  - The application must be able to move the peer-to-peer context into persistent storage and reuse it at a later stage without having to do the signalling process again.

WebRTC-PC Issues

- PRs
  - PR 2175: Permission API for receive-only media and data use cases (lgrahl)
  - (WebRTC-NV) PR 14: N29: A way to obtain user consent for one-way media and data use cases (lgrahl)
  - Issue 2167/PR 2169: {iceRestart: true} works poorly with negotiationneeded (jan-ivar)
- Issues
  - Issue 2165: A simpler glare-proof SLD() (jan-ivar)
  - Issue 2166: A simpler non-racy rollback (jan-ivar)
  - Issue 2176: Spec steps on stop() underestimates BUNDLE problem (jan-ivar)

PR 2175: Permission API for receive-only media and data use cases (lgrahl)

- **PR 2175 submitted to WebRTC-PC repo, PR 14 to WebRTC-NV repo**
- **Status Quo**
  - Without getUserMedia: Usually IP handling mode ≥ 2
    - Terminology: *Default mode*
  - With getUserMedia: Usually IP handling mode 1
    - Terminology: *Best available mode*
- **What's wrong?**
  - Not all use cases can use getUserMedia
    - Receive only media (meeting or class viewers), data channels
    - These are often innovative *local network* use cases
  - *Default mode* and *best available mode* are diverging further
    - libwebrtc: mDNS host candidate obfuscation

PR 2175: Permission API for receive-only media and data use cases (lgrahl)

- **Goals**
  - Allow to resolve privilege escalation of getUserMedia
  - Clarify when/how IP handling modes are being applied/effective/signalled.
  - Ensure use case neutrality
- **Permission API**
  - Add `"direct-connection"` permission.
  - *Media Devices* `"camera"` or `"microphone"` MAY implicitly grant `"direct-connection"`
- **WebRTC API**

WebIDL

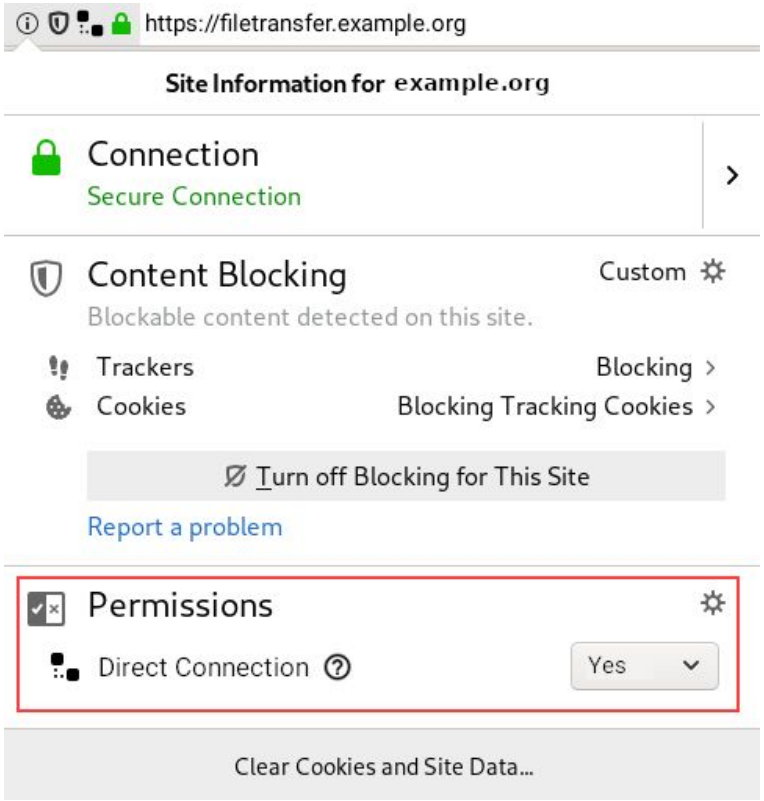
```
partial interface RTCPeerConnection {
    static void registerDirectConnectionInterest ();
    attribute EventHandler onconnectionupgradable;
};
```

PR 2175: Permission API for receive-only media and data use cases (lgrahl)

- **How does it work?**
  1. Call `RTCPeerConnection.registerDirectConnectionInterest()` eventually and ideally with context (see examples of the PR).
  2. Register `connectionupgradable` event.
  3. <Insert UX chosen by the UA here to notify the `"direct-connection"` permission interest>
  4. `connectionupgradable` fires iff the mode has been upgraded to *best available mode.* This fires on each `RTCPeerConnection` of the current realm. This MAY also fire as a result of `getUserMedia`.
  5. Do an ICE restart.

PR 2175: Permission API for receive-only media and data use cases (lgrahl)

- **UX?**
  - API is tailored to encourage the `direct-connection` permission being granted without having to rely on a prompt.
  - Gives UI designers room for experimentation.



EXAMPLE 10

```
pc.onconnectionupgradable = async () => {
  // Initiate an ICE restart
  const offer = await pc.createOffer({iceRestart: true});

  // Exchange offer/answer via the signalling channel...
};

pc.onconnectionstatechange = async () => {
  const state = pc.connectionState;
  if (state === 'failed' || state === 'connected') {
    // For audio/video...
    const iceTransport = rtpReceiver.transport.iceTransport

    // For data channels...
    const iceTransport = pc.sctp.transport.iceTransport;

    // Check if requesting a direct connection would be useful
    const pair = iceTransport.getSelectedCandidatePair();
    if (pair.local.type !== 'host') {
      // Once granted, this will fire the 'connectionupgradable' event
      RTCPeerConnection.registerDirectConnectionInterest();
    }
  }
};
```

PR 14: N29: A way to obtain user consent for one-way media and data use cases (lgrahl)

- Same as what was presented earlier (**PR 2175: Permission API for receive-only media and data use cases)**, just proposed for NV as a backup plan.

Issue 2167: {iceRestart: true} works poorly with ONN (Jan-Ivar)

How does one restart ICE today when using `negotiationneeded`? Here's a common trick (but spot the bugs!):

```
pc.onnegotiationneeded = async options => {
  await pc.setLocalDescription(await pc.createOffer(options));
  io.send({desc: pc.localDescription});
};
pc.oniceconnectionstatechange = () => {
  if (pc.iceConnectionState == "failed") {
    pc.onnegotiationneeded({iceRestart: true});
  }
};
```

Clever reuse... except this will fail if `iceconnectionstatechange` fires outside of `"stable"` state! An intermittent!

Furthermore, what if your ONN uses rollback (e.g. to implement "the polite peer")? Your ICE restart just got rolled back! What do you do? You'd need to write application logic to persist until the offer is applied and not rolled back by the other peer. Users will most likely never do this, or get it right, leaving them open to intermittents.

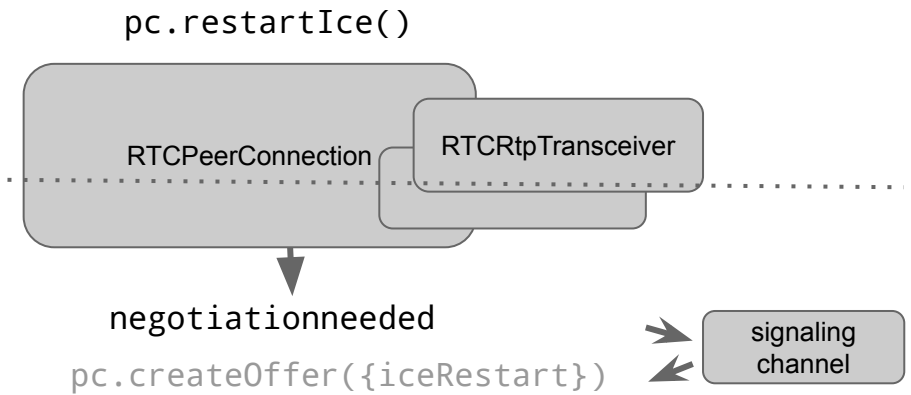
This is hard to polyfill in a way that catches all known corner cases that lead to races in common apps. PR next:

PR 2169: Add pc.restartIce() method (Jan-Ivar)

Proposal: `pc.restartIce();` `// sets [[RestartIce]], fires ONN. Cleared in SRD(answer)`

Implemented in Firefox behind pref (but not landed yet).

First-class high-level application method.
Sets [[RestartIce]] internal slot, and fires ONN.
Only cleared in SRD(answer) or by full remote ICE restart.
Flips createOffer's {iceRestart} to default to true.



Has POLA behaviors. WPT tests are written here:

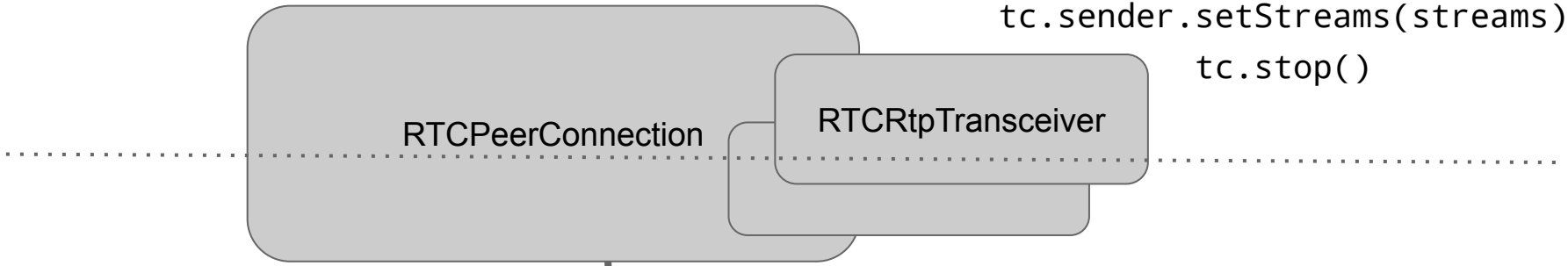
- }, "restartIce() survives rollback");
- }, "restartIce() causes fresh ufrags");
- }, "restartIce() survives remote offer");
- }, "restartIce() fires negotiationneeded");
- }, "restartIce() returns whether state changed");
- }, "restartIce() is satisfied by remote ICE restart");
- }, "{iceRestart: false} overrides and cancels local restartIce()");
- }, "restartIce() survives remote offer containing partial restart");

Issue 2165/6/7: Premise: perfect negotiation (jan-ivar)

High-level application methods

```
pc.addTrack(track, stream)      pc.addTransceiver(kind)      pc.restartIce()
pc.removeTrack(sender)          pc.createDataChannel(name)
```

```
tc.direction = “sendrecv”
tc.sender.setStreams(streams)
tc.stop()
```




```
negotiationneeded   icecandidate
```

```
pc.createOffer()
pc.createAnswer()          tc.reject()
pc.setLocalDescription()
pc.setRemoteDescription()
```

signaling channel

```
pc.addIceCandidate(candidate)
```

Low-level signaling methods

Issue 2165/6/7: Premise: perfect negotiation (jan-ivar)

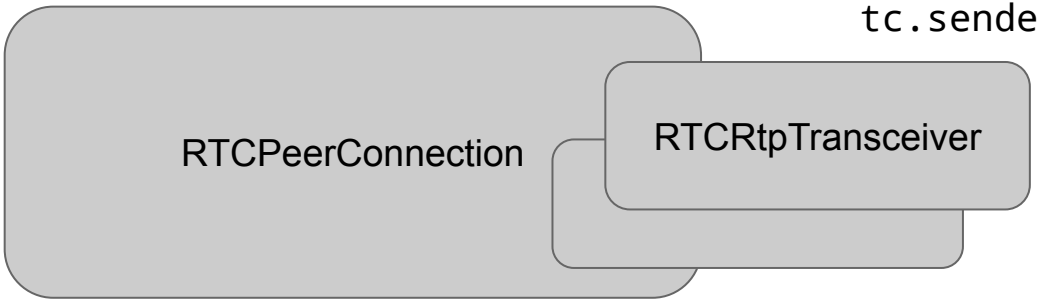
High-level application methods

```
pc.addTrack(track, stream)
pc.removeTrack(sender)
```

```
pc.addTransceiver(kind)
pc.createDataChannel(name)
```

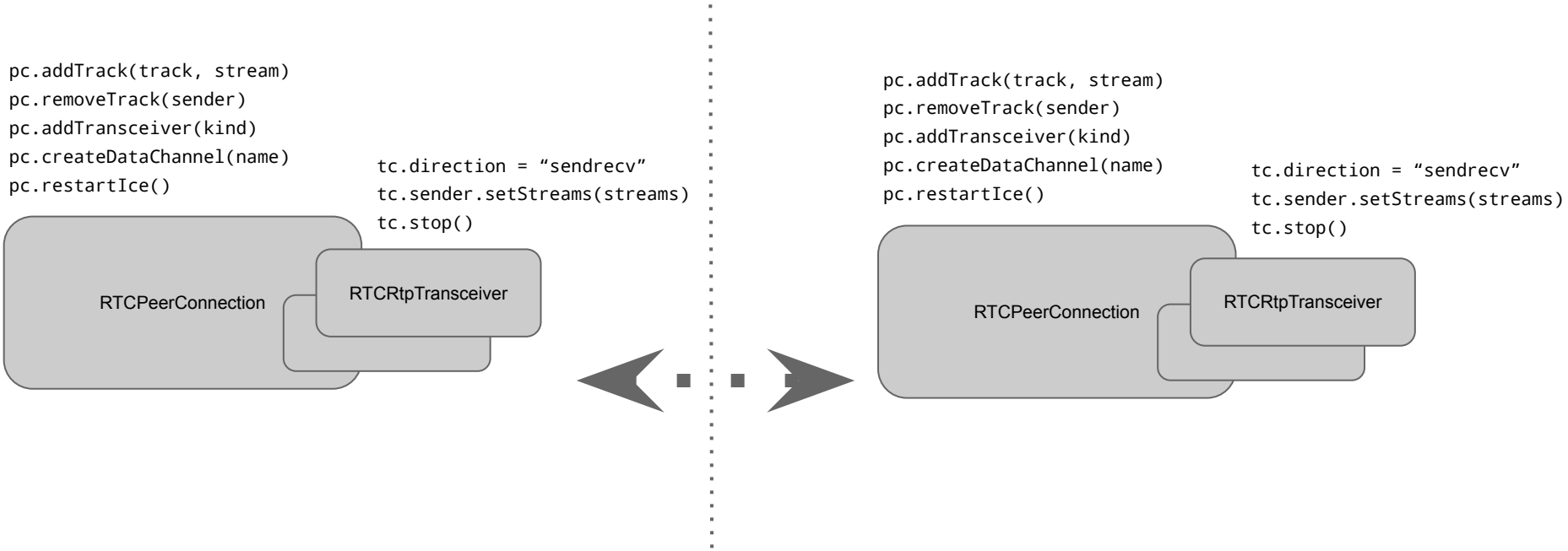
```
pc.restartIce()
```

```
tc.direction = “sendrecv”
tc.sender.setStreams(streams)
tc.stop()
```



Issue 2165/6/7: Premise: perfect negotiation (jan-ivar)

Application with perfect negotiation



(Glare is solved in negotiationneeded using rollback)

Issue 2165/6/7: Premise: perfect negotiation (jan-ivar)

Imagine: What if we could add/remove media to/from a live RTCPeerConnection, without worrying about state, glare, role (which side you're on), or what condition the connection is in? 🤔 💡

We'd simply call:

```
pc.addTrack(track, stream); // ...and that's it! Track appears remotely

// Negotiation, written once, isolated from the rest of the application logic.
pc.onicecandidate = e => { … };
pc.onnegotiationneeded = e => { … };
io.onmessage = e => { /* handles glare using rollback, on next slide */ };
```

Crazy? Chrome 75 finally fixed `negotiationneeded`, but only Firefox implements `"rollback"`.

I did this in Perfect negotiation in WebRTC blog. **TL;DR: Our APIs are racy and glare-prone!**

Issue 2165/6/7: Not a pipe-dream. Works in Firefox today (jan-ivar)

The “polite peer” signaling strategy: One side is polite, the other is not.

The polite peer uses `"rollback"` to always yield to incoming offers from (impolite) peer on **glare**:

```
// Negotiation, written once, isolated from the rest of the application logic.
io.onmessage = async ({data: {description, candidate}}) => {
  if (description) {
    if (description.type == "offer" && pc.signalingState != "stable") {
      if (!polite) return;
      await Promise.all([pc.setLocalDescription({type: "rollback"}), // Q: Why Promise.all???
                         pc.setRemoteDescription(description)]);     // A: Racy rollback! #2166
    } else {
      await pc.setRemoteDescription(description);
    }
    if (description.type == "offer") {
      await pc.setLocalDescription(await pc.createAnswer());
      io.send({description: pc.localDescription});
    }
  } else if (candidate) await pc.addIceCandidate(candidate); // Mustn't race ahead of SRD call!
};
```

Issue 2166: A simpler non-racy rollback (jan-ivar)

If remote candidates come in between rollback & SRD, don't miss them! Promise.all + PeerConnection queue avoid this, enqueueing methods ahead of addIceCandidate. But intermittents == Bad API. A simpler/safe API:

```
// Negotiation, written once, isolated from the rest of the application logic.
io.onmessage = async ({data: {description, candidate}}) => {
  if (description) {
    if (!polite && description.type == "offer" && pc.signalingState != "stable") return;
    await pc.setRemoteDescription(description, {rollback: true});  // Simpler, race-proof!
    if (description.type == "offer") {
      await pc.setLocalDescription(await pc.createAnswer());
      io.send({description: pc.localDescription});
    }
  } else if (candidate) await pc.addIceCandidate(candidate); // Never races ahead of SRD now.
};
```

PROPOSAL A: SRD takes a `{rollback: true}` options argument that will roll back an "offer" ahead of applying, *if needed*, instead of rejecting with `InvalidStateError`. Once SRD() is enqueued, addIceCandidate cannot beat it (JS ahead of first `await foo()` runs synchronously and to completion, including the synchronous part of `foo()`).
PROPOSAL B: Always do this implicitly on SRD. Then no API change is needed.

PR 2212: Add setRemoteDescription(desc, {rollback: true}) (jib)

`setRemoteDescription`

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

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

1. Let *description* be the method's first argument.
2. Let *options* be the method's second argument.
3. Let *connection* be the `RTCPeerConnection` object on which the method was invoked.
4. Return the result of enqueuing the following steps to *connection*'s operation queue:
    1. If the *description*'s `type` is `"offer"` and is invalid for the current signaling state of *connection* as described in [JSEP] (section 5.5. and section 5.6.), and *options.rollback* is `true`, then run the following sub-steps:
        1. Let *p* be the result of setting the RTCSessionDescription to a new description with a `type` of `"rollback"`.
        2. Return the result of transforming *p* with a fulfillment handler that returns the result of setting the RTCSessionDescription to *description*, and abort these steps.
    2. Return the result of setting the RTCSessionDescription to *description*.

PR: A simpler non-racy rollback (jan-ivar)

§ **4.2.9 setRemoteDescription Options**

This dictionary describes the options that can be used to control the setting of a remote description.

WebIDL

```
dictionary RTCSetRemoteDescriptionOptions {
             boolean rollback = false;
};
```

§ *Dictionary* `RTCSetRemoteDescriptionOptions` *Members*

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

When the value of this dictionary member is true, if the description being set with `setRemoteDescription` is of type `"offer"` and is invalid for the current signaling state of the `RTCPeerConnection`, the connection will be rolled back to `"stable` state ahead of attempting to set the description. The rollback, if successful, is final and not reverted should the setting of the provided description fail.

```
Promise<void> setRemoteDescription (RTCSessionDescriptionInit description,
                                    optional RTCSetRemoteDescriptionOptions options);
```

Issue 2165: A simpler glare-proof setLocalDescription() (jan-ivar)

The “polite peer” exercise revealed a similar race in `negotiationneeded`, which is in fact glare-prone:

```
pc.onnegotiationneeded = async () => {       // Always called from “stable” state only. Good!
  const offer = await pc.createOffer();     // ...except await means createOffer takes time.
  if (pc.signalingState != "stable") return; // Q: Why?! A: Avoid race w/incoming offers #2165
  await pc.setLocalDescription(offer);      // Otherwise this may fail w/InvalidStateError!
  io.send({description: pc.localDescription});
}
```

A remote offer may come in between *createOffer* & *SLD(offer)*, causing the latter to fail with `InvalidStateError`
But who’s going to know/remember that, over some rare intermittent? Instead, I propose a simpler and safe API:

```
pc.onnegotiationneeded = async () => {
  await pc.setLocalDescription();           // Simpler, glare-proof!
  io.send({description: pc.localDescription});
}
```

Proposal 1: SLD without `{sdp}` implicitly calls createOffer/Answer, if needed, instead of `InvalidStateError`.
Proposal 2: SLD without `{type}` infers type from signaling state (`state.includes("offer")`? ”answer” : ”offer”)

Issue 2165: A simpler glare-proof setLocalDescription() (jan-ivar)

Not that different from today. **Fun fact:** the sdp argument to *setLocalDescription()* is already *unused*, a ritual:

```
await pc.setLocalDescription(await pc.createOffer());
```

...is *identical* to:

```
await pc.createOffer(); await pc.setLocalDescription({type: "offer"});
```

...because the spec already says to fish out **[[LastCreatedOffer]]** and use that here. Ditto for the answer.

Proposal A: The next natural step here is...

> If **[[LastCreatedOffer]]** is null, instead of rejecting with `InvalidModificationError`, invoke the *createdOffer* algorithm implicitly to set it, before proceeding. Ditto answer.

Proposal B: Proposal A +

> Default `{type}` to (effectively) `signalingState.includes("offer")? "answer" : "offer"`

100% backwards compatible. *setRemoteDescription* would remain unchanged.

Extra slide: Perfect negotiation with a pushy SFU

Not covered in *"Perfect negotiation in WebRTC"*, is dealing with an SFU. Fippo explained SFUs can be “pushy”: They’ll send an offer, followed immediately by a second “better” offer. One strategy is “FIFO peer”:

```
io.onmessage = async ({data: {description, candidate}}) => {
  if (description) {                                  // FIFO peer:
    await Promise.all([                               // ←- Avoids race!
      pc.setRemoteDescription(description),           // ←- Always an “offer”. Fixed roles
      pc.createAnswer(),                              // ←-
      pc.setLocalDescription({type: "answer"})        // ←- Unusual, but works today
    ]);                                               // ←-
    io.send({description: pc.localDescription});
```

A second offer may come in while we’re busy responding to the first offer. Use `Promise.all` to front-load the peer connection’s queue with all our methods to get back to `"stable"` before any other methods get a go! Even with our API fixes proposed so far, we’re not able to fully get rid of `Promise.all` here. We’ll still need:

```
io.onmessage = async ({data: {description, candidate}}) => {
  if (description) {
    await Promise.all([                             // ←- Still needed
      pc.setRemoteDescription(description);
      pc.setLocalDescription();
    ]);
    io.send({description: pc.localDescription});
```

Extra slide: Negotiation methods are racy and vestigial

The SFU FIFO case shows SLD() & SRD() are racy and a bit outdated: from an earlier time when SDP mangling between createOffer/Answer to SLD was allowed (it is now forbidden), and when rejecting m-lines in the answer was common. I propose we give people a simpler and safer alternative that's race-free and glare-proof:

```
io.onmessage = async ({data: {description, candidate}}) => {
  if (description) {                                        // FIFO peer:
    await pc.setRemoteAndLocalDescriptions(description);    // ←- Always an "offer". Fixed roles
    io.send({description: pc.localDescription});
  } else if (candidate) {
    await pc.addIceCandidate(candidate);
  }
};
```

Proposal: `pc.setRemoteAndLocalDescriptions(description)` works like regular SRD, plus, if description is an offer, generates and sets a local answer before resolving. Behavior-neutral with the Promise.all case. I.e. if SRD succeeds, but SLD fails, we won't roll back on failure.

Issues 2176/2150: stop() underestimates BUNDLE problem (jib)

Current stop() language assumes BUNDLE problem is limited to "have-remove-offer" state, but it's not.

The problem is equally present if stop() is called any time *before* SRD(offer), as long as what ultimately ends up being called next is SRD(offer) rather than SLD(offer). Only then is the bundle transport toast (not everyone uses negotiationneeded)

> ***NOTE***
>
> OR BEFORE!
>
> ***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:

NO GOOD! DOESN'T COVER "STABLE" + LATER SRD(OFFER)!

6. If *connection*'s signaling state is `have-remote-offer` and *transceiver* is associated with the "offerer tagged" media description as defined in [BUNDLE], then for each `RTCRtpTransceiver` *associatedTransceiver* that is associated with a media description in the same BUNDLE group as *transceiver*, stop the RTCRtpTransceiver specified by *associatedTransceiver*.

PROBLEM: No way to stop the other transceivers synchronously in this case, since we don't know outcome yet.

POSSIBLE SOLUTIONS:

1. Stop them later in SLD(answer) only in this corner case (only solves #2176).
2. A safer stop() that only affects createOffer, & doesn't reject m-lines. A separate reject() method would work like old stop() but throw InvalidStateError outside "have-local-offer" => Solved. (solving #2176 and #2150). But this defies JSEP, which has no way to create and set a stopped offer without entering stopped state.

A JSEP problem: Once stopped in "stable" state, we don't know what comes next, SLD(offer) or SRD(offer).

Issues 2176/2150: transceiver.stop() needs more work (jan-ivar)

PROBLEM: The BUNDLE spec has painted us in a corner where calling `stop()` on the first transceiver in ***or before*** "`have-remote-offer`" signalingState, is **lethal**: stops all transceivers. This is racy behavior.

Impossible to fix in BUNDLE. Yet this flies in the face of the purpose of `negotiationneeded`, which was to abstract away negotiation, and free us from signaling state management, by separating it from high-level actions.

The two use-cases for `transceiver.stop()` are:

1. High-level (everyone): Relinquish resources after an app is done with a transceiver:

```
button.onclick = () => {
  if (button.checked) {
    this.transceiver = pc.addTransceiver(track, {streams: [stream]});
  } else {
    this.transceiver.stop();
  }
}
```

⚠️ The above code will work 99% of the time, but once in a blue moon it will stop all transceivers, just because we happened to hit the time-window where signalingState == "have-remote-offer". A footgun!

2. Low-level (expert): Reject an offered m-line in time for the answer (in "`have-remote-offer`").

Issues 2176/2150: transceiver.stop() needs more work (jan-ivar)

Why is stop() important? Web sites have been doing the following every time a participant enters & later drops:

```
const sender = pc.addTrack(track, stream}); // participant enters
pc.removeTrack(sender);                     // participant drops
```

(or something similar but convoluted for receive-only participants, but let's keep things simple to make this point)

In **PLAN-B**, this worked ok, but in **UNIFIED-PLAN**, this accumulates resources, because transceiver remains!

To prevent resources and m-lines from accumulating, sites need to call stop():

```
const tc = pc.addTransceiver(track, {streams: [stream]}); // participant enters

pc.removeTrack(tc.sender);                                // participant drops
tc.stop();                                                // resources dropped
```

⚠️ The above code will work 99% of the time, but once in a blue moon it will stop all transceivers, just because we happened to hit the time-window where signalingState == "have-remote-offer". A footgun!

Browsers cannot infer users want auto-stop, because stop() stops both directions. So this is why we need stop() to work, and not be a footgun. It's answer-rejecting properties are undesirable here. **How do we solve this?**

Issues 2176/2150: transceiver.stop() needs more work (jan-ivar)

There's a seemingly simple workaround:

```
const pc = new RTCPeerConnection(config);
pc.addTransceiver("audio");                    // Add a dummy for BUNDLE!
/* Your regular WebRTC code here */
```

...except, this may upset your app logic if you plan on using pc.addTrack:

```
pc.addTrack(track, stream);                    // Attaches itself to dummy!
```

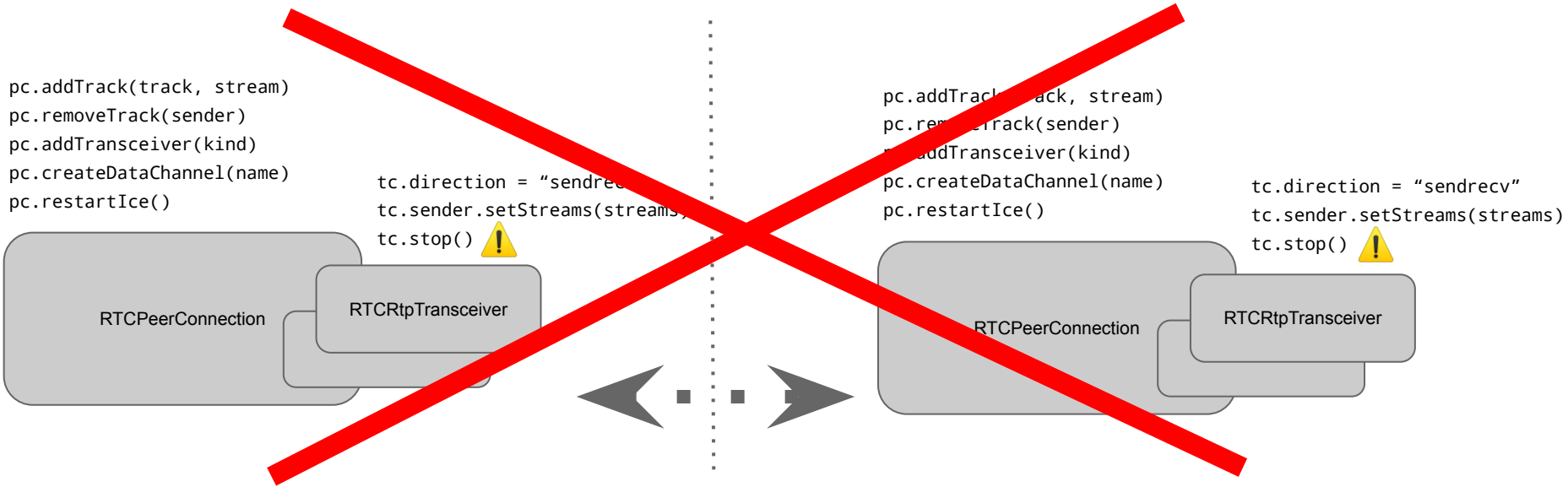
You can avoid addTrack, but addTransceiver has different semantics, e.g. won't pair up with remote transceivers. Another problem is if both ends do this, you end up with two dummies (wasted m-lines, confusion over order of transceivers and which one is used).

PROPOSAL: `pc.addTransceiver("audio", {offererTagged: true});` With special properties:

1. If another transceiver exists, throw `InvalidStateError`.
2. This transceiver is ineligible for reuse by `pc.addTrack`.
3. This transceiver is paired with remote's one if remote also uses `{offererTagged: true}`.
4. This transceiver is otherwise a regular transceiver (e.g. can be stop()ed).

Issue 2165/6/7: If we can't solve races & stop() then the API fails.

~~Application with perfect negotiation~~



Issue 2165/6/7: Hopefully we can fix this, and have a race-free API:

```
const pc = new RTCPeerConnection(config);

mediaButton.onclick = () => {
  if (button.checked) {
    this.transceiver = pc.addTransceiver(track, {streams: [stream]});
  } else {
    this.transceiver.stop();
  }
}
pc.addTransceiver("audio", {offererTagged: true});

io.onmessage = async ({data: {description, candidate}}) => {
  if (description) {
    if (!polite && description.type == "offer" && pc.signalingState != "stable") return;
    await pc.setRemoteAndLocalDescriptions(description, {rollback: true});
    if (description.type == "offer") {
      io.send({desc: pc.localDescription});
    }
  } else if (candidate) await pc.addIceCandidate(candidate);
};
pc.onnegotiationneeded = async () => {
  await pc.setLocalDescription();
  io.send({desc: pc.localDescription});
}
pc.onicecandidate = ({candidate}) => io.send({candidate});
pc.oniceconnectionstatechange = () => (pc.iceConnectionState == "failed") && pc.restartIce();
```

For extra credit

Name that lizard!

Thank you

Special thanks to:

WG Participants, Editors & Chairs

The lizard