W3C WebRTC WG Meeting

July 2, 2019
8 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 decide on proposals for substantive changes to webrtc-pc.

About this Virtual Meeting

Information on the meeting:

- Meeting info:
    - https://www.w3.org/2011/04/webrtc/wiki/July_2_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.

WebRTC-PC Issues

- Issues & PRs
  - Issue 2150/PR 2220: stop() needs more work (jan-ivar)
  - Issue 2176/PR 2220: Spec steps on stop() underestimates BUNDLE problem (jan-ivar)
  - Issue 2165: A simpler glare-proof SLD() (jan-ivar)
  - Issue 2166: A simpler non-racy rollback (jan-ivar)
  - Issue 2167/PR 2169: {iceRestart: true} works poorly with negotiationneeded (jan-ivar)
  - Issue 2221: Negotiation methods are racy with a pushy SFU (jan-ivar)
  - Tweaking Jan-Ivar's stop() - if time allows (henbos)

Introduction: 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, is isolated from the rest of the application logic
pc.onicecandidate = e => { … };        // Written perfectly
pc.onnegotiationneeded = e => { … }; // Written perfectly
io.onmessage = e => { /* Written perfectly to handle glare using rollback */ };
```

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

See the [Perfect negotiation in WebRTC]() blog, where I did this, for how to write it perfectly.

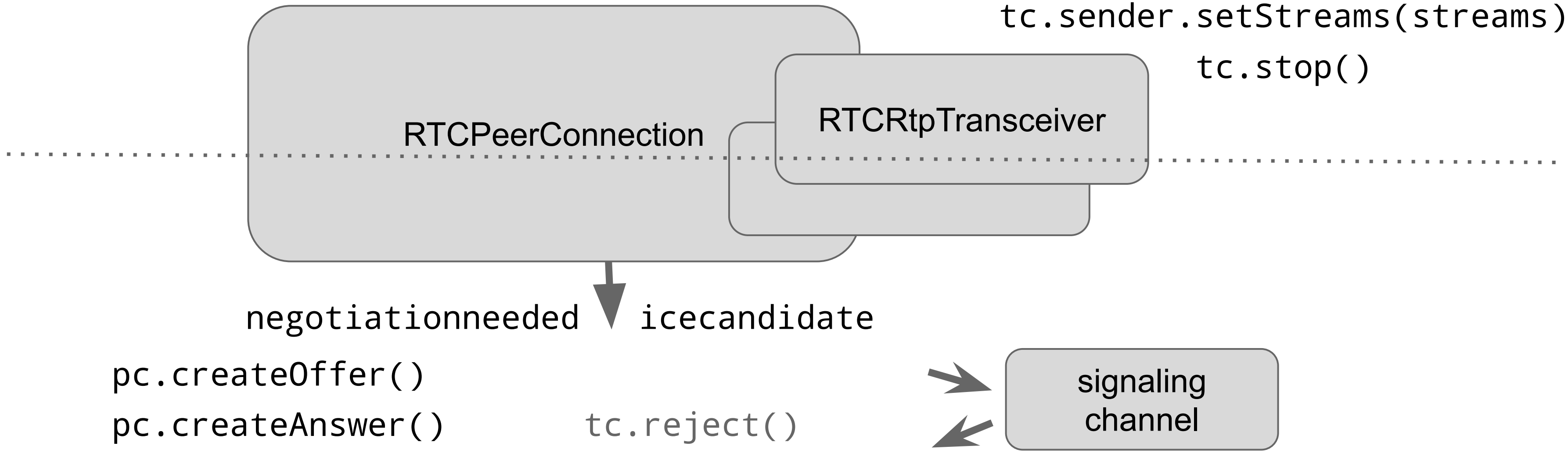
TL;DR: Works in Firefox! eveals our spec APIs to be racy and glare-prone!

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()
```



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

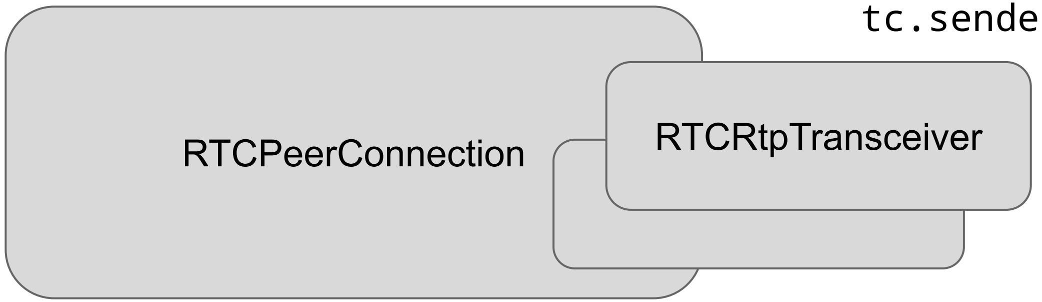
Low-level signaling methods

Premise: Perfect negotiation (jan-ivar)

High-level application methods

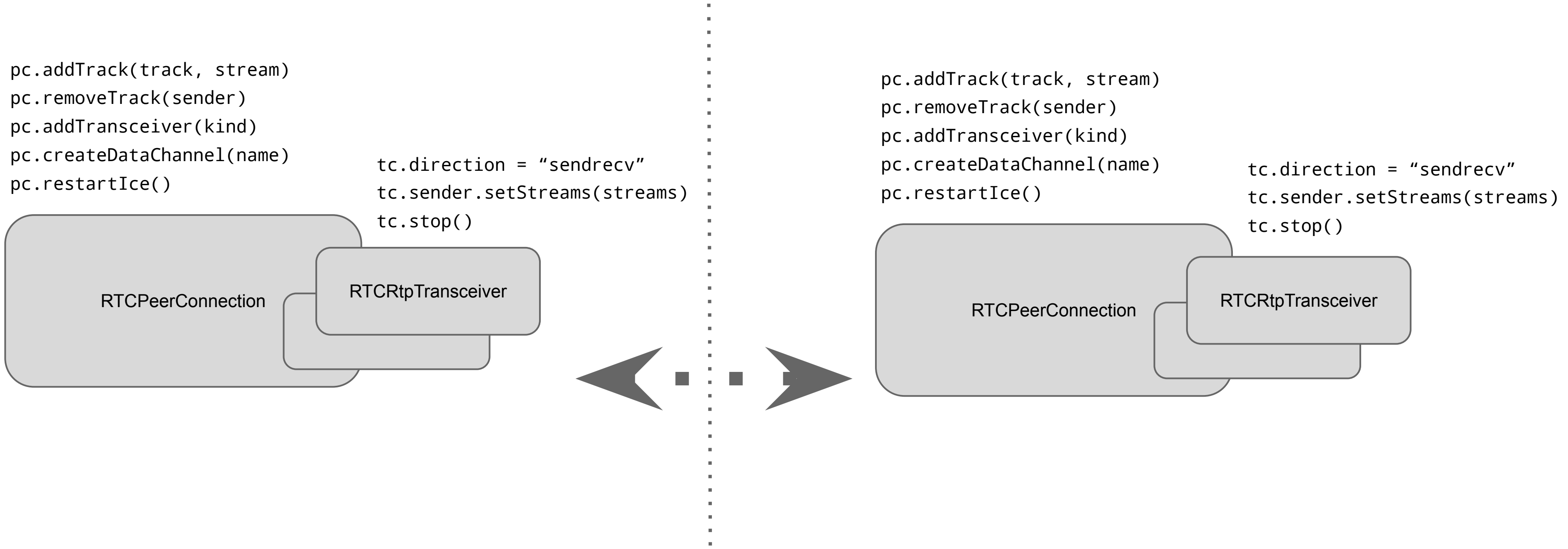
```
pc.addTrack(track, stream)    pc.addTransceiver(kind)      pc.restartIce()
pc.removeTrack(sender)        pc.createDataChannel(name)
```

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



Premise: Perfect negotiation (jan-ivar)

Application logic with perfect negotiation



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

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

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

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

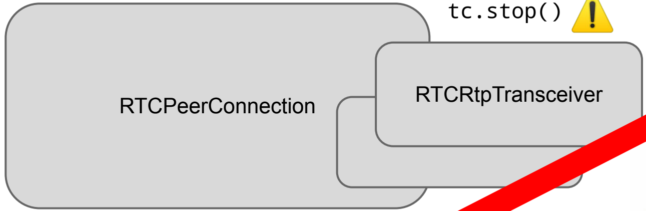
(Glare is solved in negotiationneeded using rollback)

Issue 2165/6/7: If we don’t solve races & stop() then it doesn’t work

~~Application logic with perfect negotiation~~

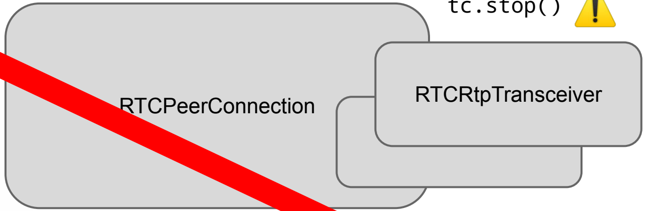
```
pc.addTrack(track, stream)
pc.removeTrack(sender)
pc.addTransceiver(kind)
pc.createDataChannel(name)
pc.restartIce()
```

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



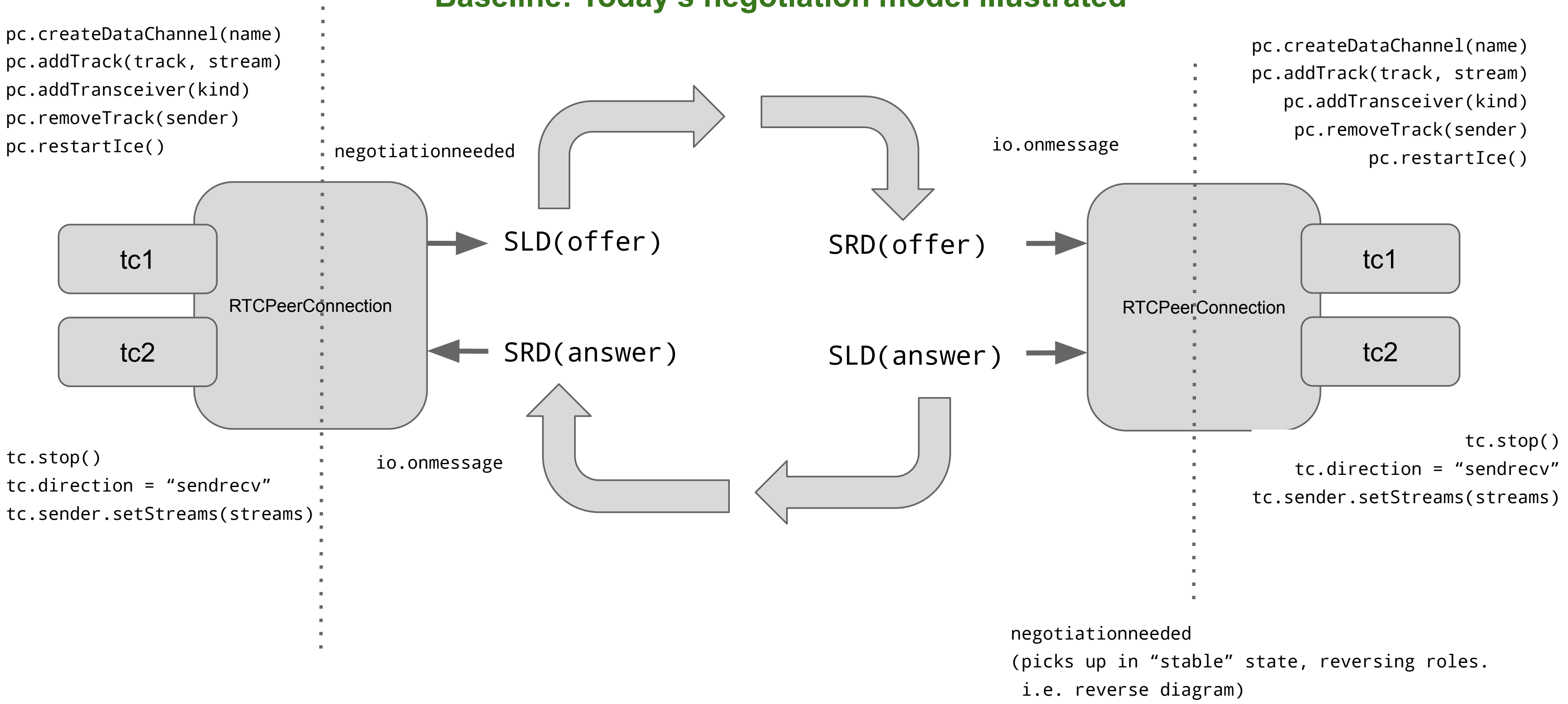
```
pc.addTrack(track, stream)
pc.removeTrack(sender)
pc.addTransceiver(kind)
pc.createDataChannel(name)
pc.restartIce()
```

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

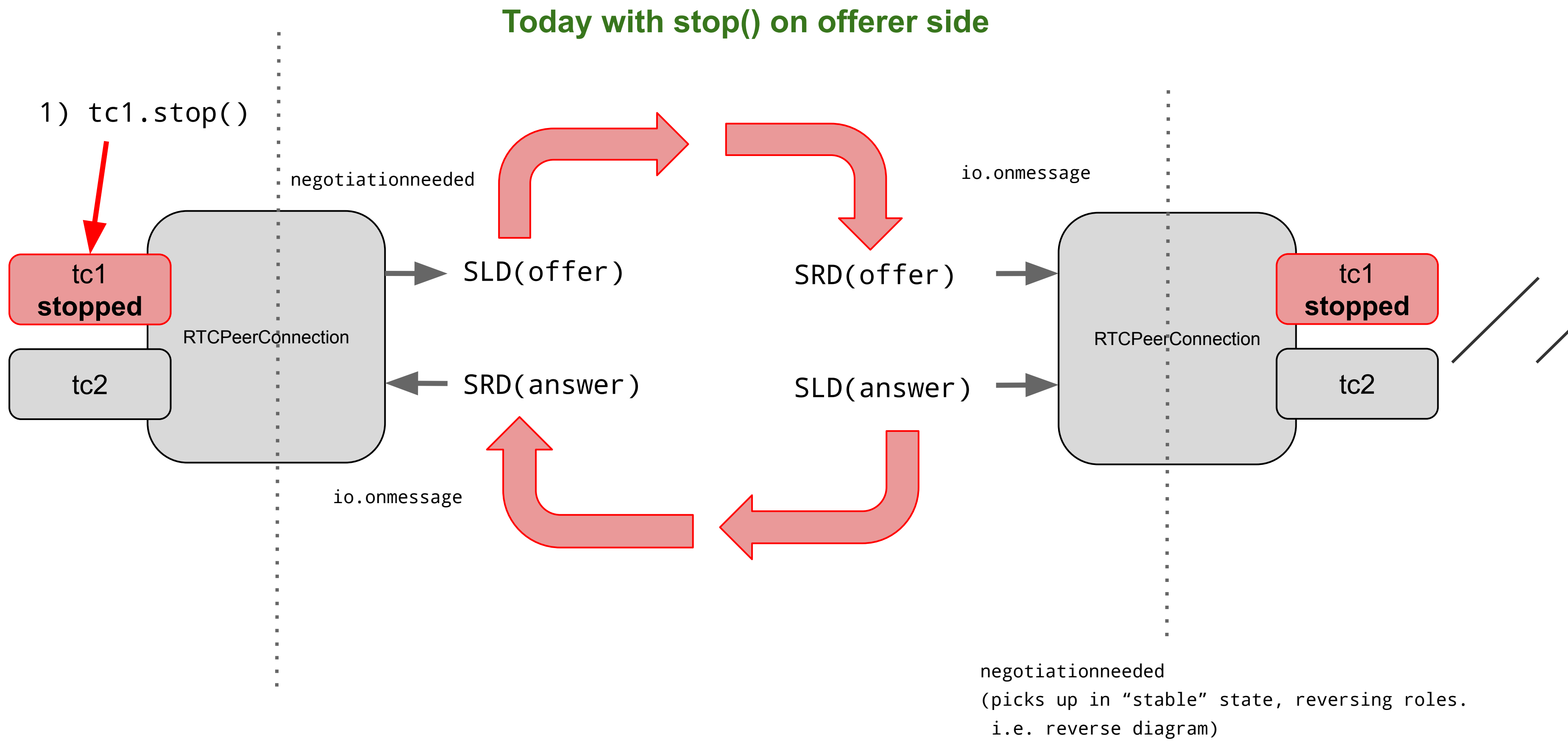


Issues 2176 & 2150: stop()'s BUNDLE problem

Baseline: Today's negotiation model illustrated

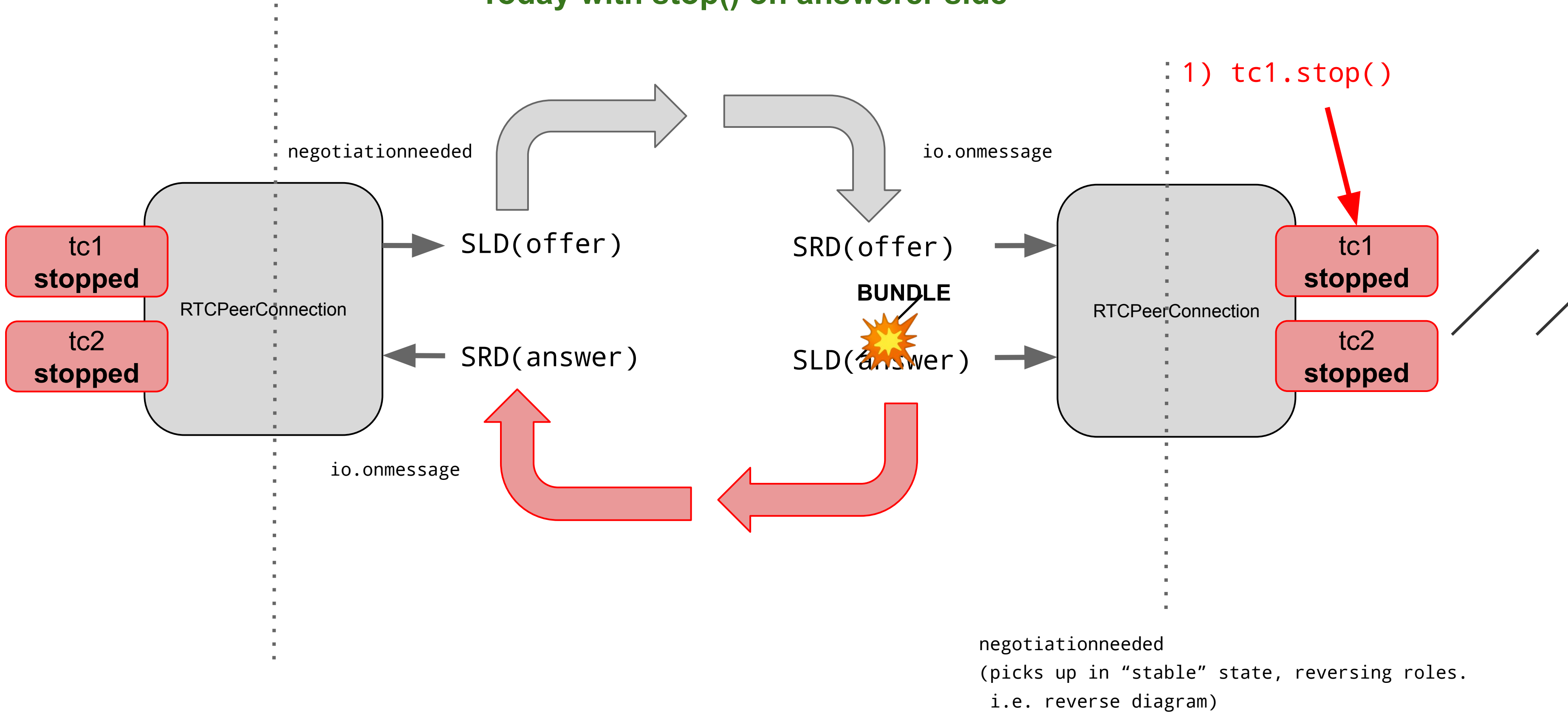


Issues 2176 & 2150: stop()'s BUNDLE problem illustrated



Issues 2176 & 2150: stop()'s BUNDLE problem illustrated

Today with stop() on answerer side



Issues 2176 & 2150 / PR 2220: 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();
  }
}
```

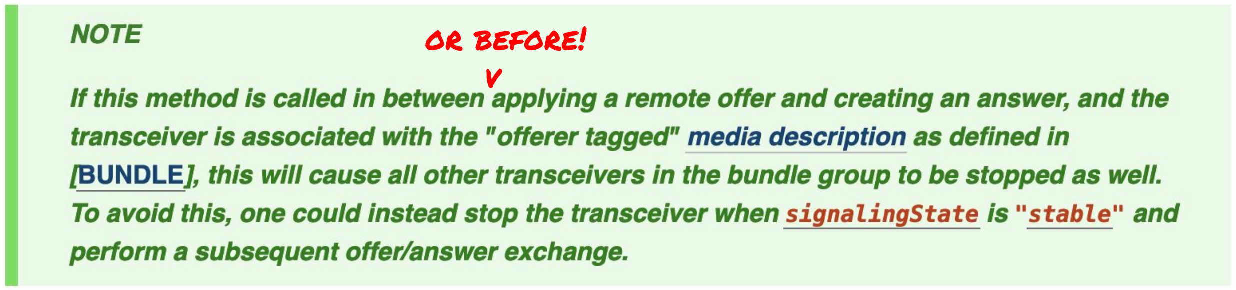
⚠️ The above code will work 95% of the time, but once in a blue moon it will stop all transceivers, just because we happened to hit the time-window where signalingState == "have-remote-offer". A footgun!

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

Issues 2176 & 2150 / PR 2220: It's worse than we thought (jan-ivar)

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

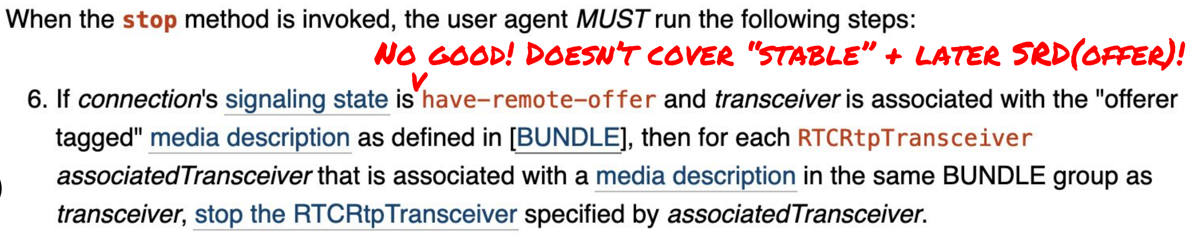
The problem is equally present if stop() is called any time *before* SRD(offer), as long as what ultimately ends up being called next is SRD(offer) rather than SLD(offer). Only then is the bundle transport toast (not everyone uses negotiationneeded)



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). Booh!~~
2. A safer stop() that only affects createOffer, not createAnswer. A separate reject() method would work like old stop() but throw InvalidStateError outside "have-local-offer". (solves both #2176 and #2150).
   Defies JSEP a bit, which doesn't anticipate creating/setting a stopped offer wo/entering stopped state.

Inherent JSEP problem: Once stopped in "stable", no way to know what comes next, SLD(offer) or SRD(offer).

Issues 2176 & 2150 / PR 2220: Do we need stop()? Yes! (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 the transceiver remains!

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

```
const sender = pc.addTrack(track, stream);                        // participant enters
pc.removeTrack(sender);                                           // participant drops
pc.getTransceivers().find(t => t.sender == sender).stop(); // drop resources
```

⚠️ The above code will work 95% 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: stop() workaround comes up short (jan-ivar)

There's a seemingly simple workaround:

```
    const pc = new RTCPeerConnection(config);
    pc.addTransceiver("audio");                    // Add an offerer-tagged dummy!
    /* Rest of your WebRTC code goes here */
```

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

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

Web developers would have to avoid addTrack outright, and learn to only use addTransceiver, which 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).

But ultimately, this doesn't solve the footgun, because people would need to actively know about it and avoid it.

People need to use stop() but want to not have to think about all this.

PR 2220: stop() sets new tc.stopping, affecting createOffer only

SOLUTION:

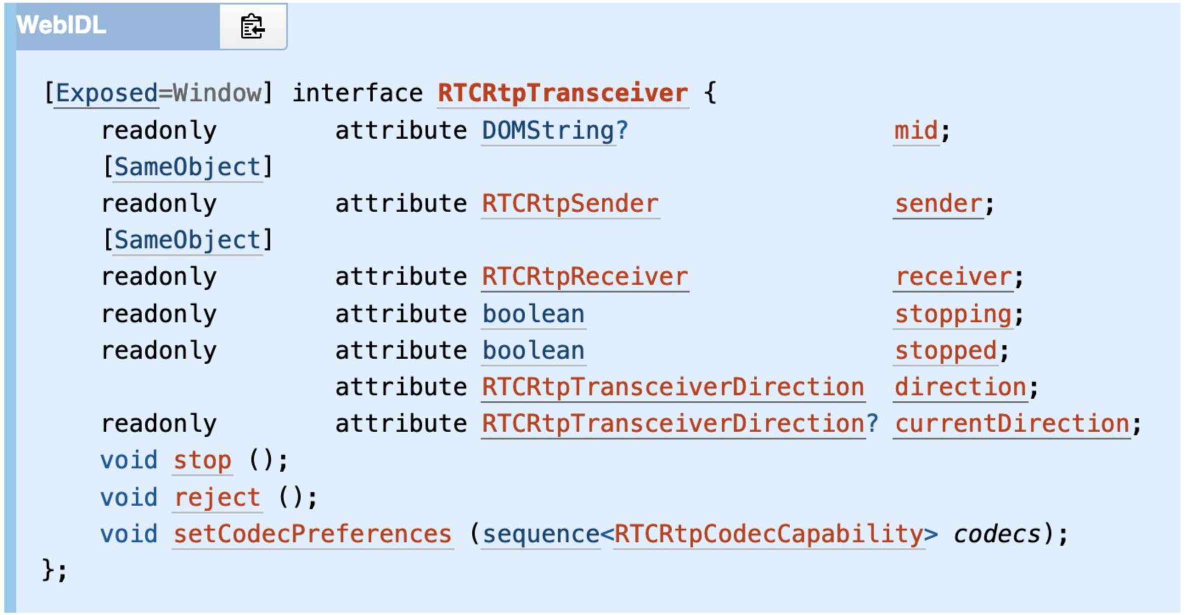
New `stopping` attribute.

`stop()` sets `stopping`, not `stopped`, but otherwise *works the same locally*.

`stopping` affects createOffer only, not createAnswer, by tricking JSEP.

SRD(offer/answer) still sets `stopped`.

We sidestep createAnswer, and that sidesteps the BUNDLE problem.



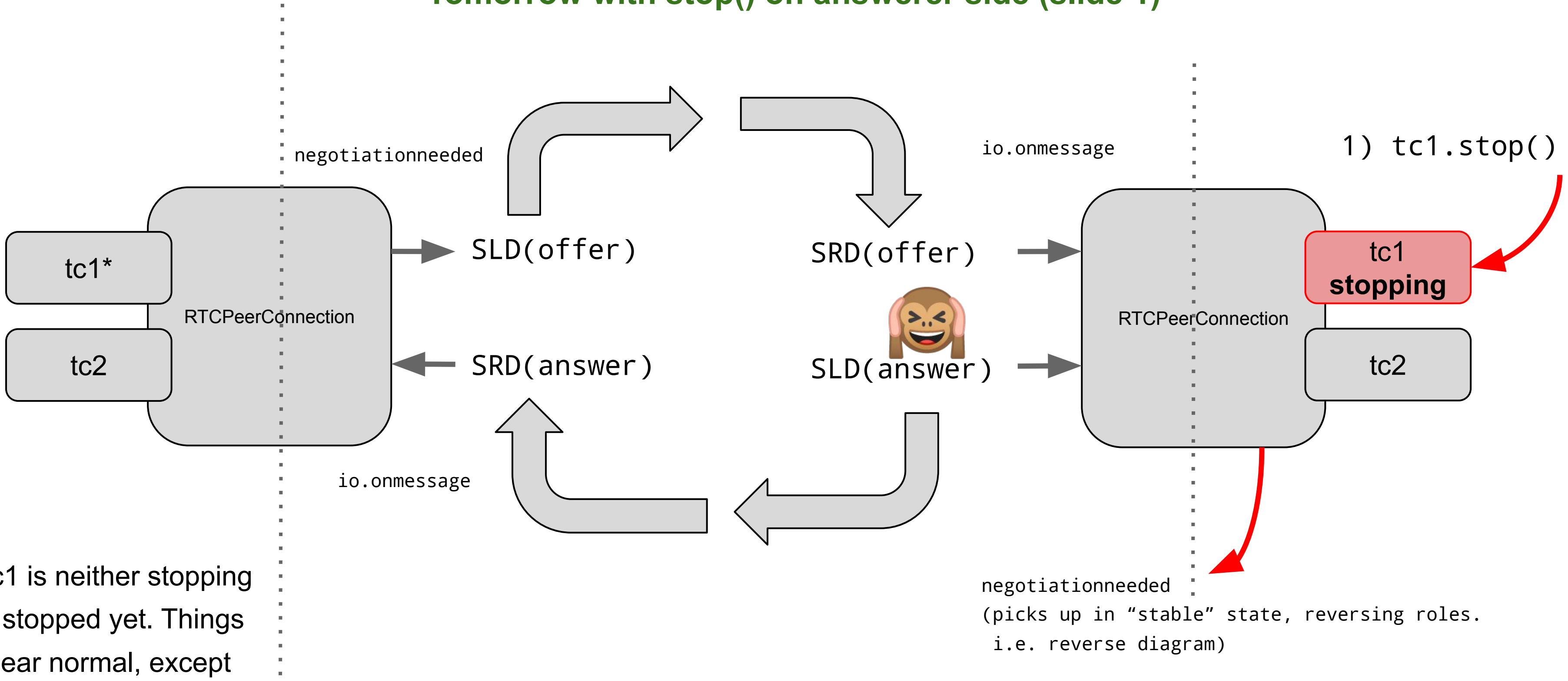
WebIDL

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

New `reject()` method sets `stopped`, just like old `stop()` did, but it only works in "have-remote-offer" state.

PR 2220: stopping solution to BUNDLE problem illustrated (1/2)

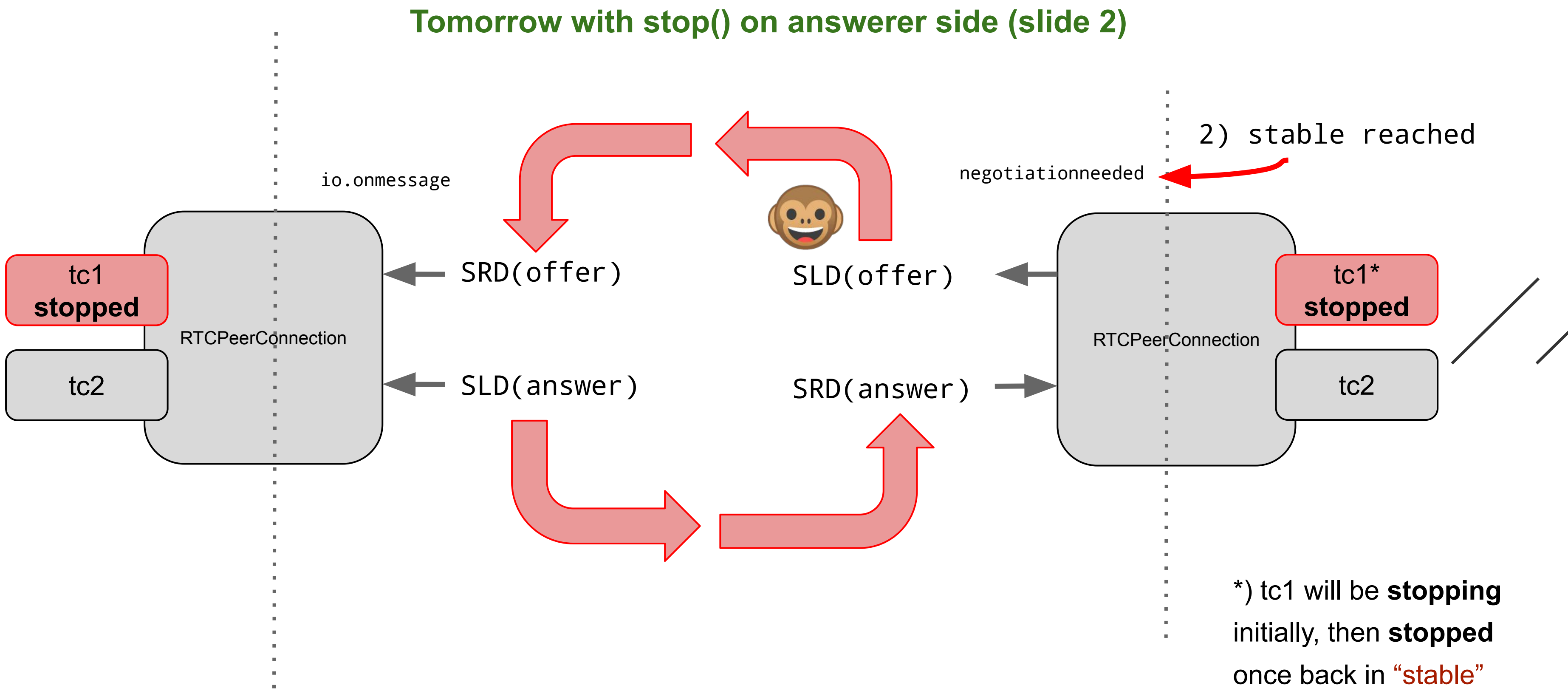
Tomorrow with stop() on answerer side (slide 1)



*) tc1 is neither stopping nor stopped yet. Things appear normal, except tc1.receiver.track has ended from RTCP BYE.

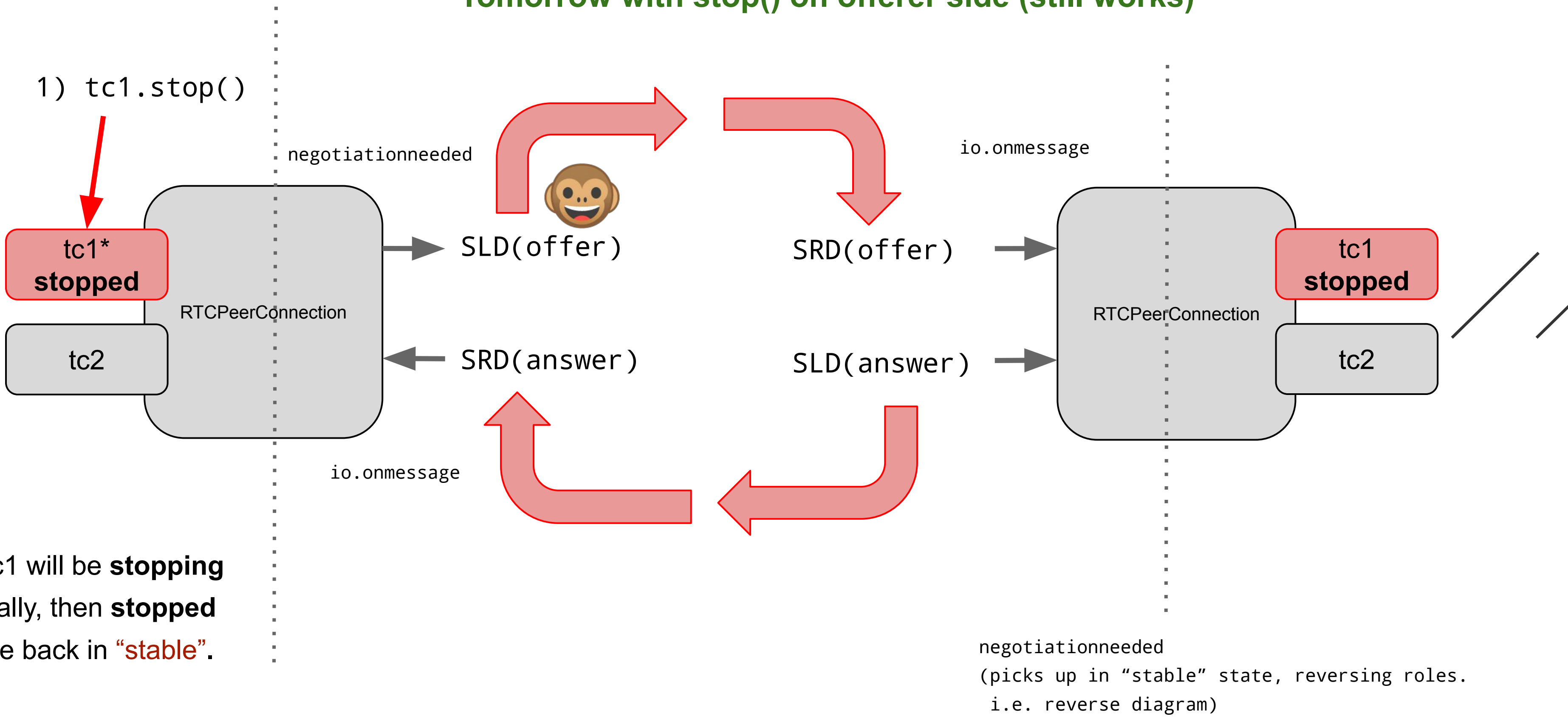
Immediately followed by...

PR 2220: stopping solution to BUNDLE problem illustrated (2/2)



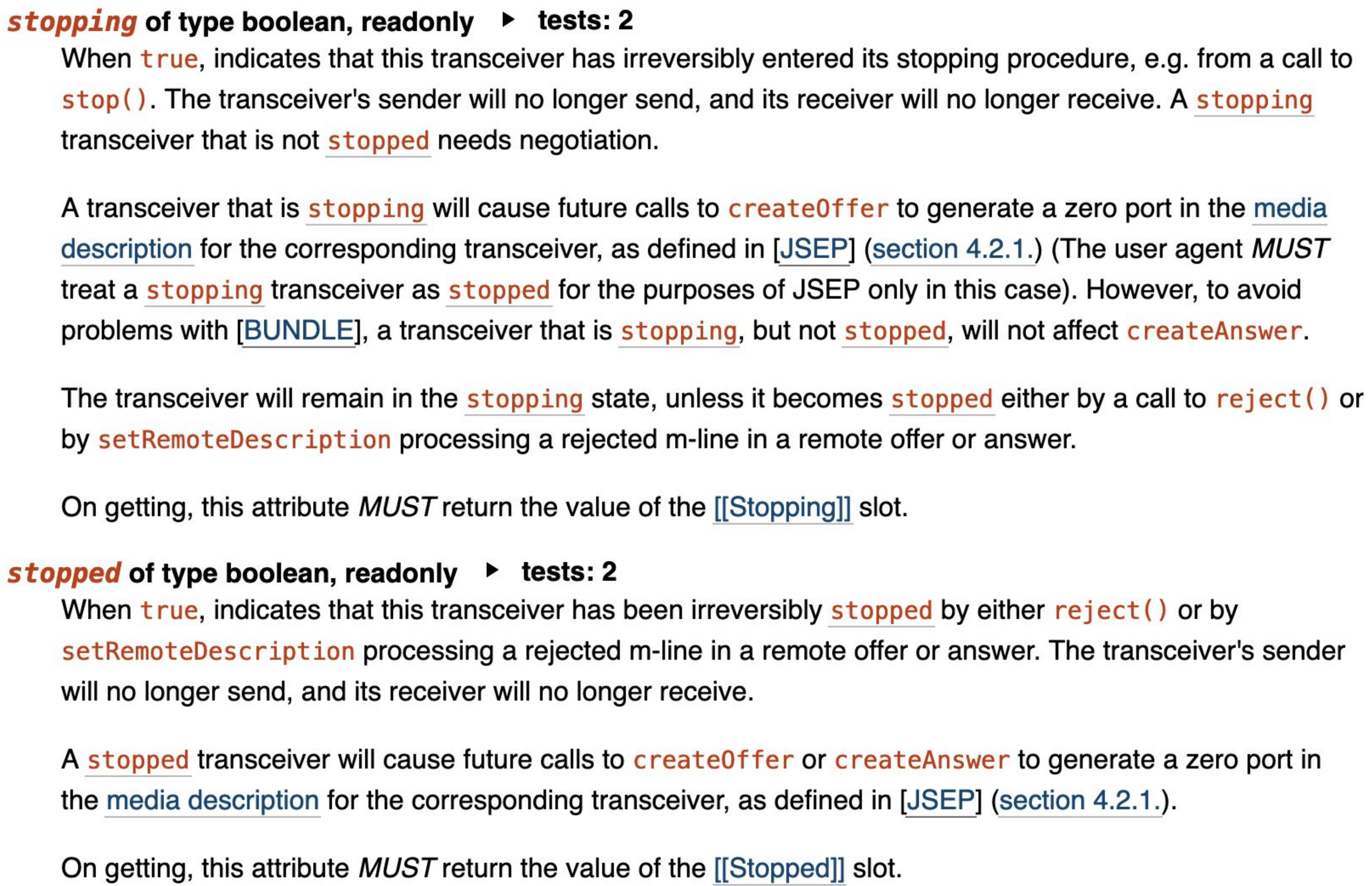
*) tc1 will be **stopping** initially, then **stopped** once back in "stable" yet again.

PR 2220: stopping solution in simple case illustrated (1/1)



*) tc1 will be **stopping** initially, then **stopped** once back in “stable”.

PR 2220: stop() sets new tc.stopping, affecting createOffer only



`stopping` **of type boolean, readonly** ▸ **tests: 2**

When `true`, indicates that this transceiver has irreversibly entered its stopping procedure, e.g. from a call to `stop()`. The transceiver's sender will no longer send, and its receiver will no longer receive. A `stopping` transceiver that is not `stopped` needs negotiation.

A transceiver that is `stopping` will cause future calls to `createOffer` to generate a zero port in the media description for the corresponding transceiver, as defined in [JSEP] (section 4.2.1.) (The user agent *MUST* treat a `stopping` transceiver as `stopped` for the purposes of JSEP only in this case). However, to avoid problems with [BUNDLE], a transceiver that is `stopping`, but not `stopped`, will not affect `createAnswer`.

The transceiver will remain in the `stopping` state, unless it becomes `stopped` either by a call to `reject()` or by `setRemoteDescription` processing a rejected m-line in a remote offer or answer.

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

`stopped` **of type boolean, readonly** ▸ **tests: 2**

When `true`, indicates that this transceiver has been irreversibly `stopped` by either `reject()` or by `setRemoteDescription` processing a rejected m-line in a remote offer or answer. The transceiver's sender will no longer send, and its receiver will no longer receive.

A `stopped` transceiver will cause future calls to `createOffer` or `createAnswer` to generate a zero port in the media description for the corresponding transceiver, as defined in [JSEP] (section 4.2.1.).

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

`stop` ▸ **tests: 1**

Irreversibly marks the transceiver as `stopping`, unless it is already `stopped`. This will immediately cause the transceiver's sender to no longer send, and its receiver to no longer receive. Calling `stop()` also updates the negotiation-needed flag for the `RTCRtpTransceiver`'s associated `RTCPeerConnection`.

> ***NOTE***
>
> ***A transceiver that is `stopping` but not `stopped` will always need negotiation. In practice, this means that calling `stop()` on a transceiver will cause the transceiver to become `stopped` eventually, provided negotiation is allowed to complete on both ends.***

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

1. Let *transceiver* be the `RTCRtpTransceiver` object on which the method is invoked.
2. Let *connection* be the `RTCPeerConnection` object associated with *transceiver*.
3. If *connection*'s [[IsClosed]] slot is `true`, throw an `InvalidStateError`.
4. If *transceiver*'s [[Stopping]] slot is `true`, abort these steps.
5. Stop sending and receiving given *transceiver*, and update the negotiation-needed flag for *connection*.

`reject` ▶ **tests: 1**

Only callable in the `have-remote-offer` signaling state of the transceiver's associated `RTCPeerConnection`. Irreversibly stops the transceiver, marking it as `stopped` immediately. The sender of this transceiver will no longer send, and the receiver will no longer receive.

> ***NOTE***
>
> ***This method may only be called between applying a remote offer and creating an answer, to reject m-lines. But if 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 rejected as well. To avoid this, users should instead consider using the safer `stop()` method, which only affects offers, not answers.***

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

1. Let *transceiver* be the `RTCRtpTransceiver` object on which the method is invoked.
2. Let *connection* be the `RTCPeerConnection` object associated with *transceiver*.
3. If *connection*'s signaling state is not `have-remote-offer`, or *connection*'s [[IsClosed]] slot is `true`, throw an `InvalidStateError`.
4. If *transceiver*'s [[Stopped]] slot is `true`, abort these steps.
5. Stop the RTCRtpTransceiver specified by *transceiver*.
6. If *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*.

Backup plan: Improve the stop() workarounds (jan-ivar)

(Only if we fail to agree to fix the problem)

WORKAROUND PROPOSAL A: `pc.addTransceiver(“audio”, {ineligible: true});` with the following special property:

1. This transceiver is ineligible for reuse by `pc.addTrack`.

This most minimal solution fixes the footgun with the workaround for the first footgun: subsequent addTrack use is POLA again. But if both ends do this, you still end up with two dummies (wasted m-lines, confusion over order of transceivers and which one is used).

WORKAROUND PROPOSAL B: `pc.addTransceiver(“audio”, {offererTagged: true});` with the following special properties:

2. This transceiver is ineligible for reuse by `pc.addTrack`.
3. If any other transceiver exists already, throw `InvalidStateError`.
4. This transceiver is somehow paired with remote’s construct if remote also uses `{offererTagged: true}`.
5. This transceiver is otherwise a regular transceiver (e.g. can be stop()ed).

Issue 2167/PR 2169: {iceRestart: true} works poorly with ONN

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.

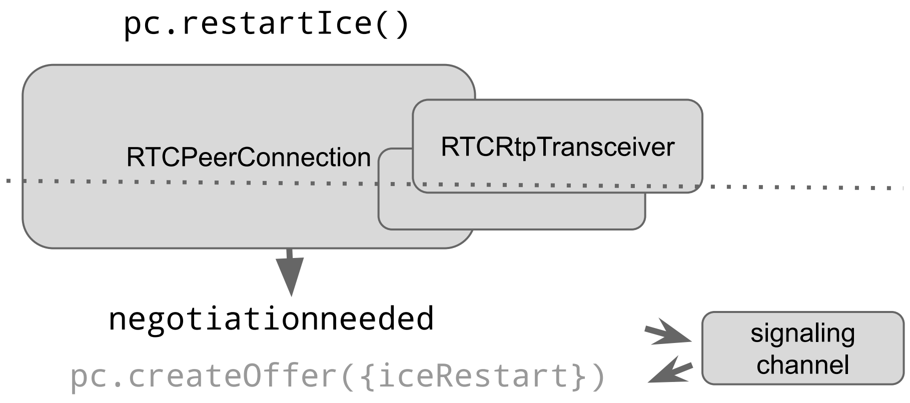
This is hard to polyfill in a way that catches all known corner cases that lead to races in common apps. PR next:

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).

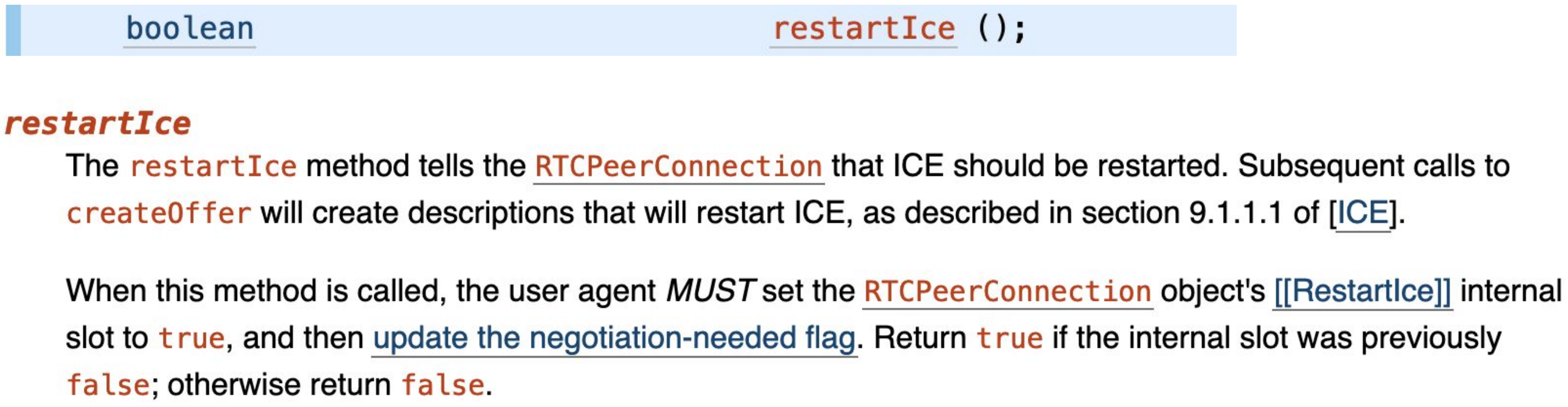
First-class high-level application method.
Sets [[RestartIce]] internal slot, and fires ONN.
Only cleared in SRD(answer) or by full remote ICE restart.
Flips createOffer's {iceRestart} to default to true.



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");

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

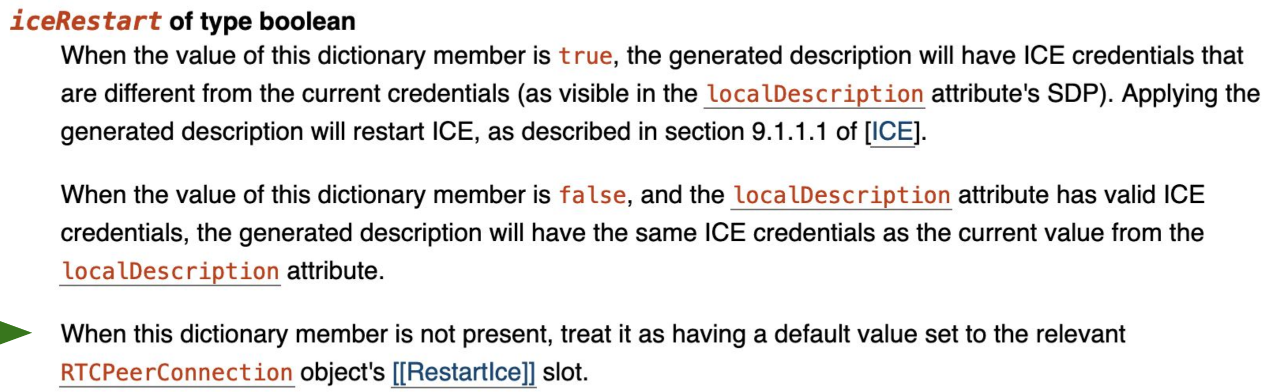


```
    boolean                          restartIce ();
```

restartIce

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

When this method is called, the user agent *MUST* set the RTCPeerConnection object's [[RestartIce]] internal slot to true, and then update the negotiation-needed flag. Return true if the internal slot was previously false; otherwise return false.



iceRestart **of type boolean**

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

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

When this dictionary member is not present, treat it as having a default value set to the relevant RTCPeerConnection object's [[RestartIce]] slot.

Issue 2165/6/7: Perfect negotiation exposes racy APIs (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/PR 2212: A simpler non-racy rollback (jan-ivar)

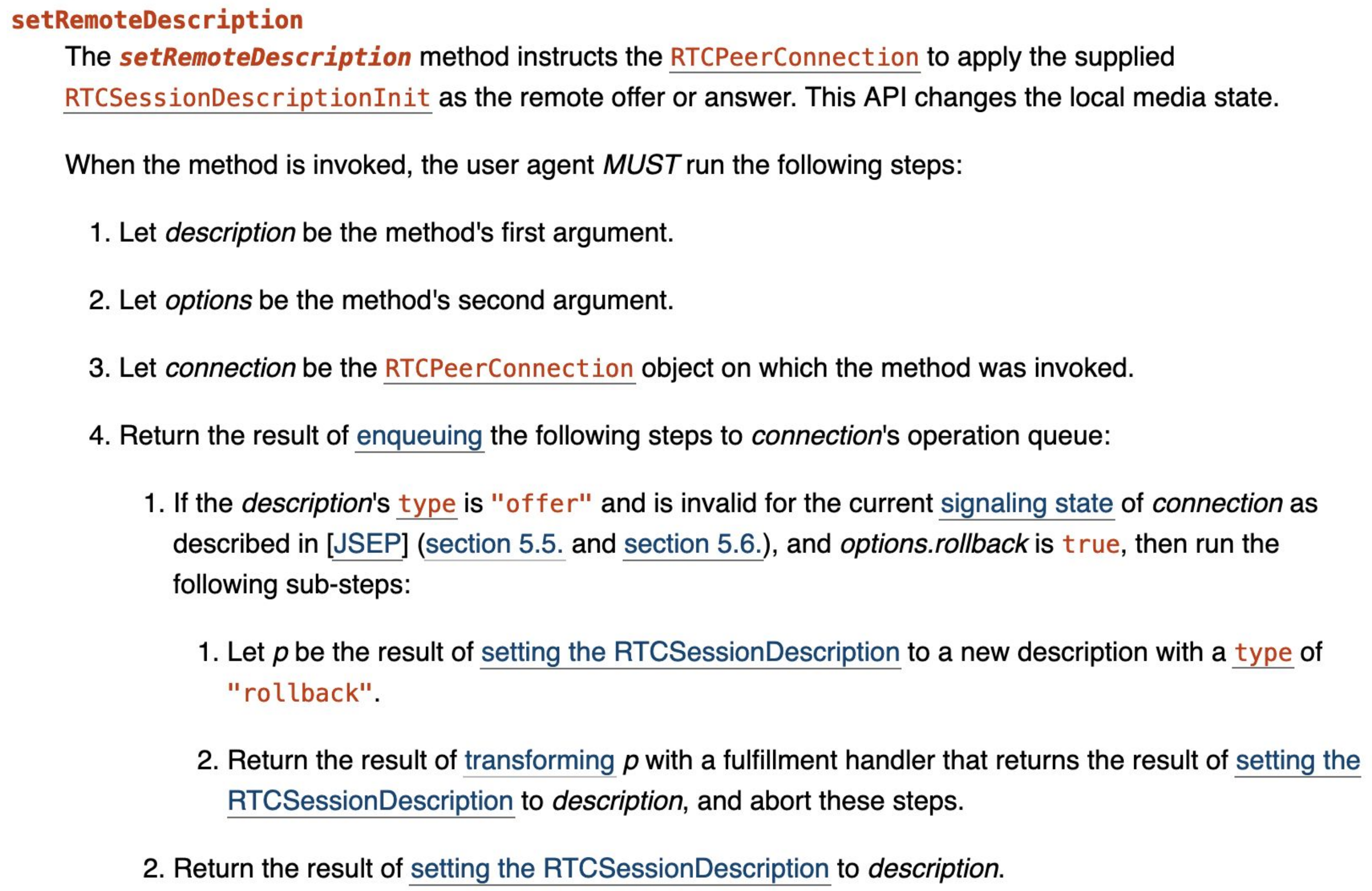
If remote candidates come in between rollback & SRD, don't miss them! Promise.all + PeerConnection queue avoid this, enqueueing methods ahead of addIceCandidate. But intermittents == Bad API. A simpler/safe API:

```
// 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 surface change is needed.

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



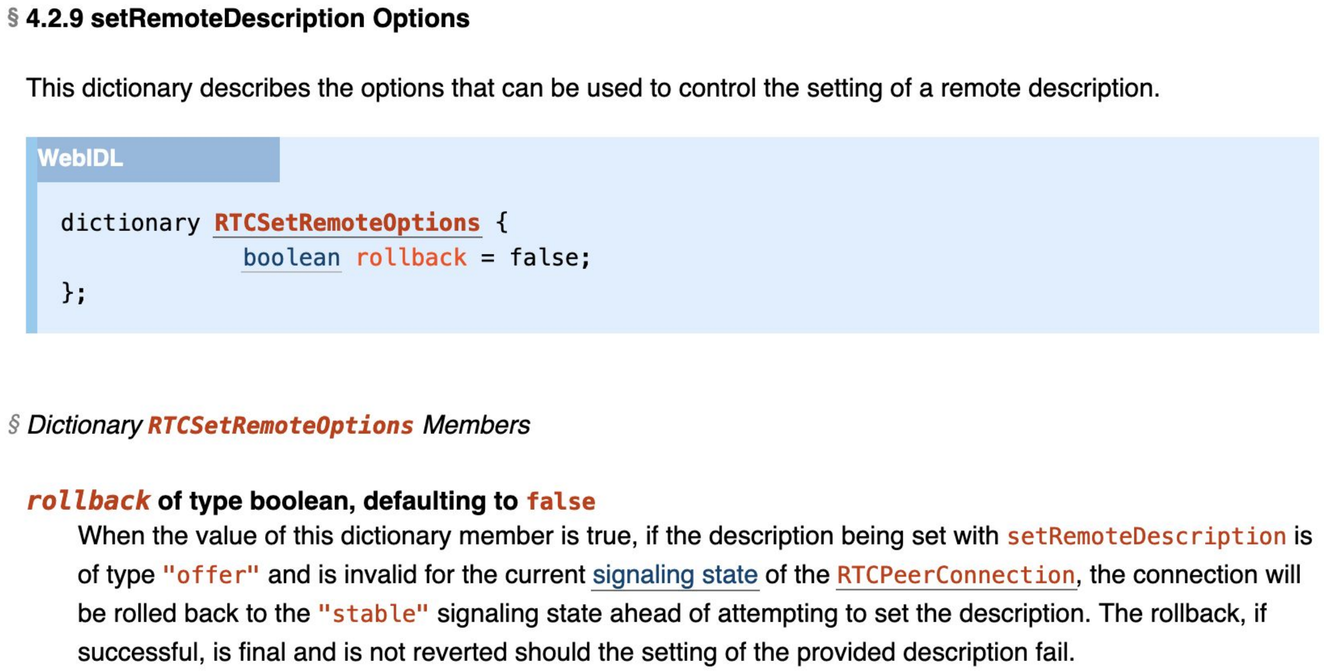
`setRemoteDescription`

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



§ 4.2.9 setRemoteDescription Options

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

WebIDL

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

§ *Dictionary* ***RTCSetRemoteOptions*** *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 the `"stable"` signaling state ahead of attempting to set the description. The rollback, if successful, is final and is 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 A: SLD without `{sdp}` implicitly calls createOffer/Answer, if needed, instead of `InvalidStateError`.
Proposal B: 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.

Issue 2221: Negotiation methods are racy 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 the "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});
```

Issue 2221: Negotiation methods are racy with a pushy SFU

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.

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();
```

Tweaking Jan-Ivar’s stop() - if time allows (1/2)

Starting point: Jan-Ivar’s slides. We have [[Stopping]] and [[Stopped]].

```
enum RTCRtpTransceiverDirection {
    "sendrecv",
    "sendonly",
    "recvonly",
    "inactive"
};

interface RTCRtpTransciever {
    ...
    readonly attribute boolean stopping;
    readonly attribute boolean stopped;
             attribute RTCRtpTransceiverDirection  direction;
    readonly attribute RTCRtpTransceiverDirection? currentDirection;
    void stop();
    void reject();
}
```

Problem: We have two new APIs, and we rarely care about the difference between “stopping” and “stopped”. We just want to avoid the BUNDLE footgun.

Tweaking Jan-Ivar’s stop() - if time allows (2/2)

Solution: Treat “stopping” as a direction and remove reject().

```
enum RTCRtpTransceiverDirection {
    "sendrecv",
    "sendonly",
    "recvonly",
    "inactive",
    "stopped"
};
```

We can remove both the `stopped` and `stopping` attributes, now covered by `direction`/`currentDirection`.

We can remove `stop()` too in favor of:

```
  t.direction = "stopped";
```

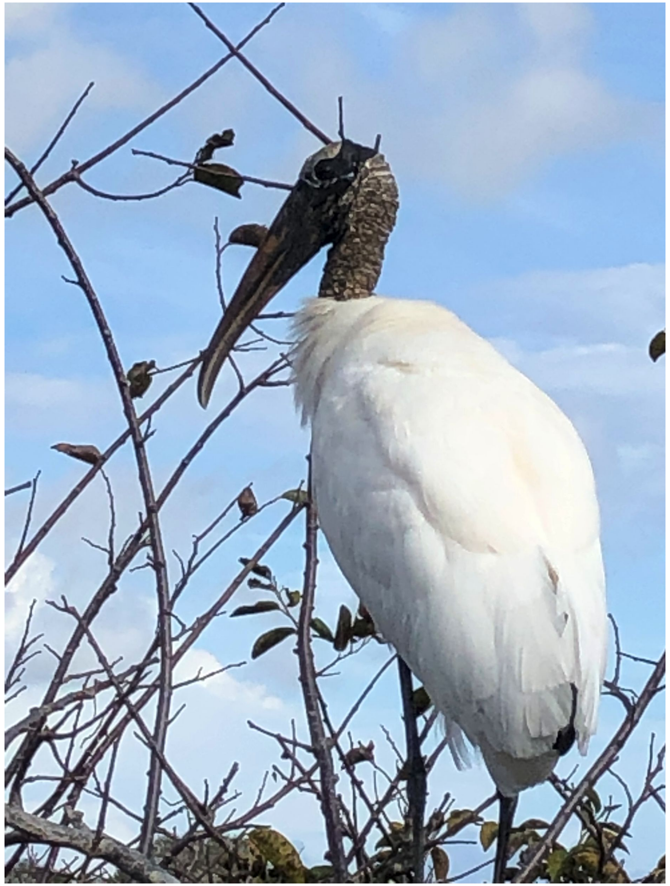
```
interface RTCRtpTransciever {
             attribute RTCRtpTransceiverDirection  direction;
    readonly attribute RTCRtpTransceiverDirection? currentDirection;
}
```

- Note that [[Direction]] is not a control surface we need once we become stopping.

- While [[Stopping]], `direction` is "stopped".
- Once [[Stopped]], both `direction` and `currentDirection` are "stopped".

- reject() optimizes away an “SDP-ping” at the cost of being a BUNDLE footgun. **It’s not worth it**. stop() is “good enough”!

For extra credit



Name that bird!

Thank you

Special thanks to:

WG Participants, Editors & Chairs

The bird