Deprecation Trial: RTCPeerConnection Plan B SDP Semantics

Author: hbos@chromium.org

Background

In Chromium-based browsers (like Google Chrome and Microsoft Edge), the RTCPeerConnection can be constructed using {sdpSemantics:"plan-b"}. This makes it speak with the Plan B dialect of the Session Description Protocol. If this argument is not specified (or if {sdpSemantics:"unified-plan"} is used) then it will speak the Unified Plan SDP dialect.

- Unified Plan is the spec-compliant SDP that is cross-browser compatible and supports the RTCRtpTransceiver APIs.
- Plan B is non-standard and targeted to be removed in M93 (see Schedule for release dates), after which constructing an RTCPeerConnection with {sdpSemantics:"plan-b"} will throw an exception. This means that applications using Plan B need to migrate their application to Unified Plan in order to function properly in M93+ unless they register for the Deprecation Trial.

By signing up to the Deprecation Trial “RTCPeerConnection Plan B SDP Semantics”, applications that need more time to successfully migrate may continue to use Plan B for a few more milestones. When the Deprecation Trial is over, Plan B will go away permanently.

I need more time. How do I sign up?

See Origin Trials Guide for Web Developers.

Once the Deprecation Trial is enabled, the RTCPeerConnection can continue to be constructed using {sdpSemantics:"plan-b"} for a limited period of time.

I have issues with migration. Where do I file feedback?

Please file feedback on the public chromium issue [Origin Trial] Developer Feedback: RTCPeerConnection Plan B SDP Semantics or write an email to hbos@chromium.org.

How is Unified Plan different from Plan B?

The Mozilla blog post The evolution of WebRTC 1.0 was written back in 2017 *before* Chrome implemented Unified Plan, so the article's comments about Chrome's implementation status is no longer accurate, but it is nevertheless a great introduction to how WebRTC has evolved from a stream-based set of APIs to a transceiver-based set of APIs. Now all major browsers support Unified Plan, so we should all get comfortable with transceivers.

Here's a rundown of differences between Plan B and Unified Plan.

SDP differences

The main difference between Plan B SDP and Unified Plan SDP is that in Plan B, all audio tracks are listed under a single audio m= section (as a=ssrc lines) and all video tracks are listed under a single video m= section. But in Unified Plan, there is one m= section per track, so if you have 10 video tracks you need to have 10 video m= sections. In Plan B, tracks can be added or removed quite freely by adding or removing ssrcs. But in Unified Plan, while the direction of m= sections can change freely (sendrecv, sendonly, recvonly or inactive), the number of m= sections have to be established by SDP offers which are used to correlate RTCRtpTransceiver objects to m= sections.

In Unified Plan, the RTCRtpTransceiver is essentially the control surface for an m= section. Because of this, the lifetime of RTCRtpSender and RTCRtpReceiver objects are dramatically different, including the lifetime of the RTCRtpReceiver's MediaStreamTrack.

API differences

In Plan B, you can have a different number of RTCRtpSenders than RTCRtpReceivers that are added or removed depending on if they are in-use. Overview:

- Calling RTCPeerConnection.addTrack() returns a new RTCRtpSender that is added to RTCPeerConnection.getSenders().
- Calling RTCPeerConnection.removeTrack(sender) removes the sender, it is no longer returned in RTCPeerConnection.getSenders().
- Calling RTCPeerConnection.setRemoteDescription() with an SDP offer or answer might add or remove RTCRtpReceiver objects from RTCPeerConnection.getReceivers().
    - The answer can contain any number of remote tracks.
    - The remote track IDs are predictable since they are contained within the SDP.
- RTCRtpTransceiver objects do not exist.
- It is impossible to roll back a local offer in order to set a remote offer.

In Unified Plan, RTCRtpTransceiver is an object consisting of an RTCRtpSender-RTCRtpReceiver pair. If you want a sender or a receiver, you need a transceiver. This means that there must necessarily always exist the same number of senders and receivers as transceivers. If you want to send a different number of tracks than you want to receive, you can make the transceiver

"sendonly" or "recvonly". After the transceivers have been associated with m= sections in the SDP there exists a 1:1 relationship between transceivers and m= sections (associated by mid).

- Calling RTCPeerConnection.addTrack() will either find an existing RTCRtpTransceiver that has never been in use or create a new RTCRtpTransceiver and return the “transceiver.sender”.
- Calling RTCPeerConnection.addTransceiver() creates a new transceiver with the desired direction.
    - RTCRtpTransceiver.direction is what you are willing to negotiate (SDP offer), and RTCRtpTransceiver.currentDirection is what is currently negotiated (SDP answer).
    - addTransceiver() can be used to configure multiple encodings (simulcast).
- Calling RTCPeerConnection.removeTrack(sender) will change the direction of the corresponding transceiver so that it is no longer sending and set transceiver.sender.track to null. Note that the sender continues to be returned in RTCPeerConnection.getSenders() and that you are allowed to make the same transceiver sending again by changing its transceiver’s direction and renegotiating.
    - You also can make it send or not send using RTCRtpTransceiver.direction and transceiver.sender.replaceTrack().
- Calling RTCPeerConnection.setLocalDescription() or RTCPeerConnection.setRemoteDescription() configures the transceivers for sending, receiving, both or neither.
    - createOffer() will generate SDP with one m= section per RTCRtpTransceiver.
    - SDP offers sets the RTCRtpTransceiver.mid value so that both endpoints have a transceiver with the same MID. This is how sender tracks on one endpoint and receiver tracks on another endpoint can be correlated. The sender and receiver track IDs on the other hand may be different!
        - Remote offer SDP will create new RTCRtpTransceiver objects for new m= sections on offer. However if there exists RTCRtpTransceiver that were created using the addTrack() API that don’t have a mid yet, these get associated with the m= sections on offer first.
    - createAnswer() will generate SDP for each m= section on offer. Note that the answer cannot add new m= sections, so if you want the answerer to send media to the offerer, the offerer needs to have allocated transceivers (m= sections) capable of receiving before creating the offer, and the answerer need to make use of these before creating the answer.
    - SDP answers update RTCRtpTransceiver.currentDirection with the negotiated direction. This is when the transceivers start to send/receive accordingly.
    - When setRemoteDescription() offers or answers to receive, RTCPeerConnection.ontrack fires with a transceiver and its receiver/track.
- Transceivers can be stopped using RTCRtpTransceiver.stop(). Once this has been negotiated, the sender, receiver and transceiver is removed and its m= section can be recycled by new transceivers.

- It *is* possible to roll back offers which makes Perfect Negotiation possible (a negotiation pattern that allows both endpoints to create offers).

Note that because receivers are not removed when an m= section stops receiving (since they might be used for receiving again in the future), the MediaStreamTrack.onmute event is what is fired in Unified Plan, compared to the MediaStreeamTrack.onended event in Plan B.

Chromium implementation of RTCPeerConnection.iceConnectionState

The RTCIceConnectionState is a compound state representing the states of the RTCIceTransports that are used by the RTCPeerConnection. It is supposed to be implemented according to this table. If we are “disconnected” then there may exist more ICE candidates to attempt to connect with and there is still hope of becoming “connected”. If we are “failed”, then not only are we “disconnected”, but we’ve attempted to connect every ICE candidate pair without success and it is not possible to become “connected” without performing an ICE restart.

There is thus a relevant distinction between “disconnected” and “failed”, but at the time of writing, Chromium has not implemented “end-of-candidates” and is unable to tell if we should be “disconnected” or “failed”. Plan B and Unified Plan worked around the lack of “end-of-candidates” differently.

In Plan B, a heuristic was implemented and “failed” would be reported based on a timeout - it was incorrect but useful (Plan B *over*-reports “failed”). In Unified Plan, this heuristic has been removed in favor of predictability which means your application has to be prepared for “disconnected” even in cases where “failed” should be reported (Unified Plan *under*-reports “failed”).