Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 25, 2013
J. Uberti
Google
C. Jennings
Cisco
October 22, 2012

Javascript Session Establishment Protocol
draft-ietf-rtcweb-jsep-02

Abstract

This document proposes a mechanism for allowing a Javascript application to fully control the signaling plane of a multimedia session, and discusses how this would work with existing signaling protocols.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

The thinking behind WebRTC call setup has been to fully specify and control the media plane, but to leave the signaling plane up to the application as much as possible. The rationale is that different applications may prefer to use different protocols, such as the existing SIP or Jingle call signaling protocols, or something custom to the particular application, perhaps for a novel use case. In this approach, the key information that needs to be exchanged is the multimedia session description, which specifies the necessary transport and media configuration information necessary to establish the media plane.

The browser environment also has its own challenges that cause problems for an embedded signaling state machine. One of these is that the user may reload the web page at any time. If this happens, and the state machine is being run at a server, the server can simply push the current state back down to the page and resume the call where it left off.

This document describes the Javascript Session Establishment Protocol (JSEP) that pulls the signaling state machine out of the browser and into Javascript. This mechanism effectively removes the browser almost completely from the core signaling flow; the only interface needed is a way for the application to pass in the local and remote session descriptions negotiated by whatever signaling mechanism is used, and a way to interact with the ICE state machine.

JSEP's handling of session descriptions is simple and straightforward. Whenever an offer/answer exchange is needed, the initiating side creates an offer by calling a createOffer() API. The application optionally modifies that offer, and then uses it to set up its local config via the setLocalDescription() API. The offer is then sent off to the remote side over its preferred signaling mechanism (e.g., WebSockets); upon receipt of that offer, the remote party installs it using the setRemoteDescription() API.

When the call is accepted, the callee uses the createAnswer() API to generate an appropriate answer, applies it using setLocalDescription(), and sends the answer back to the initiator over the signaling channel. When the offerer gets that answer, it installs it using setRemoteDescription(), and initial setup is complete. This process can be repeated for additional offer/answer exchanges.

Regarding ICE, JSEP decouples the ICE state machine from the overall signaling state machine, as the ICE state machine must remain in the browser, because only the browser has the necessary knowledge of

candidates and other transport info. Performing this separation also provides additional flexibility; in protocols that decouple session descriptions from transport, such as Jingle, the transport information can be sent separately; in protocols that don't, such as SIP, the information can be used in the aggregated form. Sending transport information separately can allow for faster ICE and DTLS startup, since the necessary roundtrips can occur while waiting for the remote side to accept the session.

The JSEP approach does come with a minor downside. As the application now is responsible for driving the signaling state machine, slightly more application code is necessary to perform call setup; the application must call the right APIs at the right times, and convert the session descriptions and ICE information into the defined messages of its chosen signaling protocol, instead of simply forwarding the messages emitted from the browser.

One way to mitigate this is to provide a Javascript library that hides this complexity from the developer, which would implement the state machine and serialization of the desired signaling protocol. For example, this library could convert easily adapt the JSEP API into the exact ROAP API [I-D.jennings-rtcweb-signaling], thereby implementing the ROAP signaling protocol. Such a library could of course also implement other popular signaling protocols, including SIP or Jingle. In this fashion we can enable greater control for the experienced developer without forcing any additional complexity on the novice developer.

2. Other Approaches Considered

Another approach that was considered for JSEP was to move the mechanism for generating offers and answers out of the browser as well. Instead of providing createOffer/createAnswer methods within the browser, this approach would instead expose a getCapabilities API which would provide the application with the information it needed in order to generate its own session descriptions. This increases the amount of work that the application needs to do; it needs to know how to generate session descriptions from capabilities, and especially how to generate the correct answer from an arbitrary offer and the supported capabilities. While this could certainly be addressed by using a library like the one mentioned above, it basically forces the use of said library even for a simple example. Exposing createOffer/createAnswer avoids that problem, but still allows applications to generate their own offers/answers if they choose, using the description generated by createOffer as an indication of the browser's capabilities.

Note also that while JSEP transfers more control to Javascript, it is not intended to be an example of a "low-level" API. The general argument against a low-level API is that there are too many necessary API points, and they can be called in any order, leading to something that is hard to specify and test. In the approach proposed here, control is performed via session descriptions; this requires only a few APIs to handle these descriptions, and they are evaluated in a specific fashion, which reduces the number of possible states and interactions.

3. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

4. Semantics and Syntax

4.1. Signaling Model

JSEP does not specify a particular signaling model or state machine, other than the generic need to exchange RFC 3264 offers and answers in order for both sides of the session to know how to conduct the session. JSEP provides mechanisms to create offers and answers, as well as to apply them to a session. However, the actual mechanism by which these offers and answers are communicated to the remote side, including addressing, retransmission, forking, and glare handling, is left entirely up to the application.

```
    +-----------+                               +-----------+
    |  Web App  |<--- App-Specific Signaling -->|  Web App  |
    +-----------+                               +-----------+
          ^                                            ^
          |  SDP                                       |  SDP
          V                                            V
    +-----------+                                +-----------+
    |  Browser  |<----------- Media ------------>|  Browser  |
    +-----------+                                +-----------+
```

Figure 1: JSEP Signaling Model

4.2. Session Descriptions and State Machine

In order to establish the media plane, the user agent needs specific parameters to indicate what to transmit to the remote side, as well as how to handle the media that is received. These parameters are determined by the exchange of session descriptions in offers and answers, and there are certain details to this process that must be handled in the JSEP APIs.

Whether a session description was sent or received affects the meaning of that description. For example, the list of codecs sent to a remote party indicates what the local side is willing to decode, and what the remote party should send. Not all parameters follow this rule; for example, the SRTP parameters [RFC4568] sent to a remote party indicate what the local side will use to encrypt, and thereby how the remote party should expect to receive.

In addition, various RFCs put different conditions on the format of offers versus answers. For example, a offer may propose multiple SRTP configurations, but an answer may only contain a single SRTP configuration.

Lastly, while the exact media parameters are only known only after a

offer and an answer have been exchanged, it is possible for the offerer to receive media after they have sent an offer and before they have received an answer. To properly process incoming media in this case, the offerer's media handler must be aware of the details of the offerer before the answer arrives.

Therefore, in order to handle session descriptions properly, the user agent needs:

1. To know if a session description pertains to the local or remote side.

2. To know if a session description is an offer or an answer.

3. To allow the offer to be specified independently of the answer.

JSEP addresses this by adding both a setLocalDescription and a setRemoteDescription method and having session description objects contain a type field indicating the type of session description being supplied. This satisfies the requirements listed above for both the offerer, who first calls setLocalDescription(sdp [offer]) and then later setRemoteDescription(sdp [answer]), as well as for the answerer, who first calls setRemoteDescription(sdp [offer]) and then later setLocalDescription(sdp [answer]). While it could be possible to implicitly determine the value of the offer/answer argument, requiring it to be specified explicitly is more robust, allowing invalid combinations (i.e. an answer before an offer) to generate an appropriate error.

JSEP also allows for an answer to be treated as provisional by the application. Provisional answers provide a way for an answerer to communicate initial session parameters back to the offerer, in order to allow the session to begin, while allowing a final answer to be specified later. This concept of a final answer is important to the offer/answer model; when such an answer is received, any extra resources allocated by the caller can be released, now that the exact session configuration is known. These "resources" can include things like extra ICE components, TURN candidates, or video decoders. Provisional answers, on the other hand, do no such deallocation results; as a result, multiple dissimilar provisional answers can be received and applied during call setup.

In [RFC3264], the constraints at the signaling level is that only one offer can be outstanding for a given session but from the media stack level, a new offer can be generated at any point. For example, when using SIP for signaling, if one offer is sent, then cancelled using a SIP CANCEL, another offer can be generated even though no answer was received for the first offer. To support this, the JSEP media layer

can provide an offer whenever the Javascript application needs one for the signaling. The answerer can send back zero or more provisional answers, and finally end the offer-answer exchange by sending a final answer. The state machine for this is as follows:

```
      +-----------+
      |           |
      |           |
      |  Stable   |<---------------\
      |           |                |
      |           |                |
      +-----------+                |
          ^   |                    |
          |   | OFFER              |
   ANSWER |   |                    | ANSWER
          |   V                    |
      +-----------+          +-----------+
      |           |          |           |
      |           | PRANSWER |           |
      |   Offer   |-------- >| Pranswer  |
      |           |          |           |
      |           |----\     |           |----\
      +-----------+    |     +-----------+    |
                 ^     |                ^     |
                 |     |                |     |
                 \-----/                \-----/
                  OFFER                 PRANSWER
```

Figure 2: JSEP State Machine

Aside from these state transitions, there is no other difference between the handling of provisional ("pranswer") and final ("answer") answers.

4.3. Session Description Format

In the WebRTC specification, session descriptions are formatted as SDP messages. While this format is not optimal for manipulation from Javascript, it is widely accepted, and frequently updated with new features. Any alternate encoding of session descriptions would have to keep pace with the changes to SDP, at least until the time that this new encoding eclipsed SDP in popularity. As a result, JSEP continues to use SDP as the internal representation for its session descriptions.

However, to simplify Javascript processing, and provide for future flexibility, the SDP syntax is encapsulated within a SessionDescription object, which can be constructed from SDP, and be

serialized out to SDP. If future specifications agree on a JSON format for session descriptions, we could easily enable this object to generate and consume that JSON.

Other methods may be added to SessionDescription in the future to simplify handling of SessionDescriptions from Javascript. Though it is unclear exactly what manipulations developer will commonly want to do to SDP, it would be simple to write a Javascript library to perform these manipulations.

4.4. ICE

When a new ICE candidate is available, the ICE Agent will notify the application via a callback; these candidates will automatically be added to the local session description. When all candidates have been gathered, the callback will also be invoked to signal that the gathering process is complete.

4.4.1. ICE Candidate Trickling

Candidate trickling is a technique through which a caller may incrementally provide candidates to the callee after the initial offer has been dispatched; the semantics of "Trickle ICE" are defined in [I-D.rescorla-mmusic-ice-trickle]. This process allows the callee to begin acting upon the call and setting up the ICE (and perhaps DTLS) connections immediately, without having to wait for the caller to gather all possible candidates. This results in faster call startup in cases where gathering is not performed prior to initating the call.

JSEP supports optional candidate trickling by providing APIs that provide control and feedback on the ICE candidate gathering process. Applications that support candidate trickling can send the initial offer immediately and send individual candidates when they get the notified of a new candidate; applications that do not support this feature can simply wait for the indication that gathering is complete, and then create and send their offer, with all the candidates, at this time.

Upon receipt of trickled candidates, the receiving application will supply them to its ICE Agent. This triggers the ICE Agent to start using the new remote candidates for connectivity checks.

4.4.1.1. ICE Candidate Format

As with session descriptions, the syntax of the IceCandidate object provides some abstraction, but can be easily converted to and from the SDP a=candidate lines.

The a=candidate lines are the only SDP information that is contained within IceCandidate, as they represent the only information needed that is not present in the initial offer (i.e. for trickle candidates). This information is carried with the same syntax as the "a=candidate" line in SDP. For example:

```
a=candidate:1 1 UDP 1694498815 192.0.2.33 10000 typ host
```

The IceCandidate object also contains fields to indicate which m= line it should be associated with. The m line can be identified in one of two ways; either by a m-line index, or a MID. The m-line index is a zero-based index, referring to the Nth m-line in the SDP. The MID uses the "media stream identification", as defined in [RFC 3388], to identify the m-line. WebRTC implementations creating an ICE Candidate object MUST populate both of these fields. Implementations receiving an ICE Candidate object SHOULD use the MID if they implement that functionality, or the m-line index, if not.

4.5. Interactions With Forking

Some call signaling systems allow various types of forking where an SDP Offer may be provided to more than one device. For example, SIP RFC 3261 defines both a "Parallel Search" and "Sequential Search". Although these are primarily signaling level issues that are outside the scope of JSEP, they do have some impact on the configuration of the media plane, which is relevant. When forking is happening at the signaling layer, the Javascript application responsible for the signaling needs to make the decisions about what media should be sent or received at any point of time and which remote endpoint it should communicate with. JSEP is used to make sure the media engine can make the RTP and media perform as required by the application. The basic operations that the applications can have the media engine do are:

Start exchanging media to a given remote peer but keep all the resources reserved in the offer.

Start exchanging media with a given remote peer and free any resources in the offer that are not being used.

4.5.1. Sequential Forking

Sequential forking involves a call being dispatched to multiple remote callees, where each callee can accept the call, but only one active session ever exists at a time; no mixing of received media is performed.

JSEP handles serial forking well, allowing the application to easily

control the policy for selecting the desired remote endpoint. When an answer arrives from one of the callees, the application can choose to apply it either as a provisional answer, leaving open the possibility of using a different answer in the future, or apply it as a final answer, ending the setup flow.

In a "first-one-wins" situation, the first answer will be applied as a final answer, and the application will reject any subsequent answers. In SIP parlance, this would be ACK + BYE.

In a "last-one-wins" situation, all answers would be applied as provisional answers, and any previous call leg will be terminated. At some point, the application will end the setup process, perhaps with a timer; at this point, the application could reapply the existing remote description as a final answer.

4.5.2. Parallel Forking

Parallel forking involves a call being dispatched to multiple remote callees, where each callee can accept the call, and multiple simultaneous active signaling sessions can be established as a result. If multiple callees send media at the same time, the possibilities for handling this are described in Section 3.1 of RFC 3960. Most SIP devices today only support exchanging media with a single device at a time, and do not try to mix multiple early media audio sources, as that could result in a confusing situation. For example. consider having a European ringback tone mixed together with the North American ringback tone - the resulting sound would not be like either tone, and would confuse the user. If the signaling application wishes to only exchange media with one of the remote endpoints at a time, then from a media engine point of view, this is exactly like the sequential forking case.

In the parallel forking case where the Javascript application wishes to simultaneously exchange media with multiple peers, the flow is slightly more complex, but the Javascript application can follow the strategy that RFC 3960 describes using UPDATE. (It is worth noting that use cases where this is the desired behavior are very unusual.) The UPDATE approach allows the signaling to set up a separate media flow for each peer that it wishes to exchange media with. In JSEP, this offer used in the UPDATE would be formed by simply creating a new PeerConnection and making sure that the same local media streams have been added into this new PeerConnection. Then the new PeerConnection object would produce a SDP offer that could be used by the signaling to perform the UPDATE strategy discussed in RFC 3690.

As a result of sharing the media streams, the application will end up with N parallel PeerConnection sessions, each with a local and remote

description and their own local and remote addresses. The media flow from these sessions can be managed by specifying SDP direction attributes in the descriptions, or the application can choose to play out the media from all sessions mixed together. Of course, if the application wants to only keep a single session, it can simply terminate the sessions that it no longer needs.

4.6. Session Rehydration

In the event that the local application state is reinitialized, either due to a user reload of the page, or a decision within the application to reload itself (perhaps to update to a new version), it is possible to keep an existing session alive via a process called "rehydration".

With rehydration, the current signaling state is persisted somewhere outside of the page, perhaps on the application server, or in browser local storage. The page is then reloaded, and a new session object is created in Javascript. The saved signaling state is now retrieved, and a new PeerConnection object is created for the session. At this point a new offer can be generated by the new PeerConnection, with new ICE and SDES credentials. This can then be used to re-initiate the session with the existing remote endpoint, who simply sees the new offer as an in-call renegotiation, and will reply with an answer that can be supplied to setRemoteDescription. ICE processing proceeds as usual, and as soon as connectivity is established, the session will be back up and running again.

Open Issue: EKR proposed an alternative rehydration approach where the actual internal PeerConnection object in the browser was kept alive for some time after the web page was killed and provided some way for a new page to acquire the old PeerConnection object.

5. Interface

This section details the basic operations that must be present to implement JSEP functionality. The actual API exposed in the W3C API may have somewhat different syntax, but should map easily to these concepts.

5.1. SDP Requirements

Note: The text in this section may not represent working group consensus and is put here so that the working group can discuss it and find out how to change it such that it does have consensus.

When generating SDP blobs, either for offers or answers, the generated SDP needs to conform to the following specifications. Similarly, in order to properly process received SDP blobs, implementations need to implement the functionality described in the following specifications. This list is derived from [I-D.ietf-rtcweb-rtp-usage].

RFC4566 is the base SDP specification and MUST be implemented.

RFC5124 MUST be supported for signaling RTP/SAVPF RTP profile.

RFC5104 MUST be implemented to signal RTCP based feedback.

RFC5761 MUST be implemented to signal multiplexing of RTP and RTCP.

RFC5245 MUST be implemented for signaling the ICE candidate lines corresponding to each media stream.

RFC3264 MUST be implemented to signal information about media direction.

The RFC5888 grouping framework MUST be implemented for signaling the grouping information.

RFC5506 MAY be implemented to signal Reduced-Size RTCP messages.

RFC5576 MAY be implemented to signal RTP SSRC values.

RFC3556 with bandwidth modifiers MAY be supported for specifying RTCP bandwidth as a fraction of the media bandwidth, RTCP fraction allocated to the senders and setting maximum media bit-rate boundaries.

As required by RFC 4566 Section 5.13 JSEP implementations MUST ignore

unknown attributes (a=) lines.

Example SDP for RTCWeb call flows can be found in [I-D.nandakumar-rtcweb-sdp].

5.2. Methods

5.2.1. createOffer

The createOffer method generates a blob of SDP that contains a RFC 3264 offer with the supported configurations for the session, including descriptions of the local MediaStreams attached to this PeerConnection, the codec/RTP/RTCP options supported by this implementation, and any candidates that have been gathered by the ICE Agent. A constraints parameters may be supplied to provide additional control over the generated offer, e.g. to get a full set of session capabilities, or to request a new set of ICE credentials.

In the initial offer, the generated SDP will contain all desired functionality for the session (certain parts that are supported but not desired by default may be omitted); for each SDP line, the generation of the SDP must follow the appropriate process for generating an offer. In the event createOffer is called after the session is established, createOffer will generate an offer that is compatible with the current session, incorporating any changes that have been made to the session since the last complete offer-answer exchange, such as addition or removal of streams. If no changes have been made, the offer will be identical to the current local description.

Session descriptions generated by createOffer must be immediately usable by setLocalDescription; if a system has limited resources (e.g. a finite number of decoders), createOffer should return an offer that reflects the current state of the system, so that setLocalDescription will succeed when it attempts to acquire those resources. Because this method may need to inspect the system state to determine the currently available resources, it may be implemented as an async operation.

Calling this method may do things such as generate new ICE credentials, but does not change media state.

5.2.2. createAnswer

The createAnswer method generates a blob of SDP that contains a RFC 3264 SDP answer with the supported configuration for the session that is compatible with the parameters supplied in the offer. Like createOffer, the returned blob contains descriptions of the local

MediaStreams attached to this PeerConnection, the codec/RTP/RTCP options negotiated for this session, and any candidates that have been gathered by the ICE Agent. A constraints parameter may be supplied to provide additional control over the generated answer.

As an answer, the generated SDP will contain a specific configuration that specifies how the media plane should be established.

Session descriptions generated by createAnswer must be immediately usable by setLocalDescription; like createOffer, the returned description should reflect the current state of the system. Because this method may need to inspect the system state to determine the currently available resources, it may need to be implemented as an async operation.

Calling this method may do things such as generate new ICE credentials, but does not change media state.

5.2.3. SessionDescriptionType

Session description objects (RTCSessionDescription) may be of type "offer", "pranswer", and "answer". These types provide information as to how the description parameter should be parsed, and how the media state should be changed.

"offer" indicates that a description should be parsed as an offer; said description may include many possible media configurations. A description used as an "offer" may be applied anytime the PeerConnection is in a stable state, or as an update to a previously sent but unanswered "offer".

"pranswer" indicates that a description should be parsed as an answer, but not a final answer, and so should not result in the freeing of allocated resources. It may result in the start of media transmission, if the answer does not specify an inactive media direction. A description used as a "pranswer" may be applied as a response to an "offer", or an update to a previously sent "answer".

"answer" indicates that a description should be parsed as an answer, the offer-answer exchange should be considered complete, and any resources (decoders, candidates) that are no longer needed can be released. A description used as an "answer" may be applied as a response to a "offer", or an update to a previously sent "pranswer".

The application can use some discretion on whether an answer should be applied as provisional or final. For example, in a serial forking scenario, an application may receive multiple "final" answers, one from each remote endpoint. The application could accept the initial

answers as provisional answers, and only apply an answer as final when it receives one that meets its criteria (e.g. a live user instead of voicemail).

5.2.3.1. Creating Answers

Most web applications will not need to create answers using the "pranswer" type. The general recommendation for a web application would be to create an answer more or less immediately after receiving the offer, instead of waiting for a human user to provide input. Later when the human input is received, the applications can create a new offer to update the previous offer/answer pair. Some applications may not be able to do this, particularly ones that Some application may not be able to do this, particular ones that are attempting to gateway to other signaling protocols.

Consider a typical web application that will set up a data channel, an audio channel, and a video channel. When an endpoint receives an offer with these channels, it could send an answer accepting the data channel for two-way data, and accepting the audio and video tracks as receive-only. It could then ask the user if they wanted to transmit audio and video to the far end, acquire the local media streams, and send a new offer to the remote side moving the audio and video to be two-way media. By the time the human has authorized sending media, it is likely that the ICE and DTLS handshaking with the remote side will already be set up.

5.2.4. setLocalDescription

The setLocalDescription method instructs the PeerConnection to apply the supplied SDP blob as its local configuration. The type field indicates whether the blob should be processed as an offer, provisional answer, or final answer; offers and answers are checked differently, using the various rules that exist for each SDP line.

This API changes the local media state; among other things, it sets up local resources for receiving and decoding media. In order to successfully handle scenarios where the application wants to offer to change from one media format to a different, incompatible format, the PeerConnection must be able to simultaneously support use of both the old and new local descriptions (e.g. support codecs that exist in both descriptions) until a final answer is received, at which point the PeerConnection can fully adopt the new local description, or roll back to the old description if the remote side denied the change.

If setRemoteDescription was previous called with an offer, and setLocalDescription is called with an answer (provisional or final), and the media directions are compatible, this will result in the

starting of media transmission.

5.2.5. setRemoteDescription

The setRemoteDescription method instructs the PeerConnection to apply the supplied SDP blob as the desired remote configuration. As in setLocalDescription, the type field of the indicates how the blob should be processed.

This API changes the local media state; among other things, it sets up local resources for sending and encoding media.

If setRemoteDescription was previous called with an offer, and setLocalDescription is called with an answer (provisional or final), and the media directions are compatible, this will result in the starting of media transmission.

5.2.6. localDescription

The localDescription method returns a copy of the current local configuration, i.e. what was most recently passed to setLocalDescription, plus any local candidates that have been generated by the ICE Agent.

A null object will be returned if the local description has not yet been established.

5.2.7. remoteDescription

The remoteDescription method returns a copy of the current remote configuration, i.e. what was most recently passed to setRemoteDescription, plus any remote candidates that have been supplied via processIceMessage.

A null object will be returned if the remote description has not yet been established.

5.2.8. updateIce

The updateIce method allows the configuration of the ICE Agent to be changed during the session, primarily for changing which types of local candidates are provided to the application and used for connectivity checks. A callee may initially configure the ICE Agent to use only relay candidates, to avoid leaking location information, but update this configuration to use all candidates once the call is accepted.

Regardless of the configuration, the gathering process collects all

available candidates, but excluded candidates will not be surfaced in onicecallback or used for connectivity checks.

This call may result in a change to the state of the ICE Agent, and may result in a change to media state if it results in connectivity being established.

5.2.9. addIceCandidate

The addIceCandidate method provides a remote candidate to the ICE Agent, which will be added to the remote description. Connectivity checks will be sent to the new candidate.

This call will result in a change to the state of the ICE Agent, and may result in a change to media state if it results in connectivity being established.

6. Configurable SDP Parameters

Note: This section is still very early and is likely to significantly change as we get a better understanding of the a) the use cases for this b) the implications at the protocol level c) feedback from implementors on what they can do.

The following is a partial list of SDP parameters that an application may want to control, in either local or remote descriptions, using this API.

- o remove or reorder codecs (m=)

- o change codec attributes (a=fmtp; ptime)

- o enable/disable BUNDLE (a=group)

- o enable/disable RTCP mux (a=rtcp-mux)

- o change send resolution or framerate (TBD)

- o change desired recv resolution or framerate (TBD)

- o change total bandwidth (b=)

- o remove desired AVPF mechanisms (a=rtcp-fb)

- o remove RTP header extensions (a=rtphdr-ext)

- o add/change SSRC grouping (e.g. FID, RTX, etc) (a=ssrc-group)

- o add SSRC attributes (a=ssrc)

- o change media send/recv state (a=sendonly/recvonly/inactive)

For example, an application could implement call hold by adding an a=inactive attribute to its local description, and then applying and signaling that description.

7. Security Considerations

TODO

8. IANA Considerations

This document requires no actions from IANA.

9. Acknowledgements

Harald Alvestrand, Dan Burnett, Neil Stratford, Eric Rescorla, Anant Narayanan, and Adam Bergkvist all provided valuable feedback on this proposal. Suhas Nandakumar provided text and input for SDP requirements. Matthew Kaufman provided the observation that keeping state out of the browser allows a call to continue even if the page is reloaded.

10. References

10.1. Normative References

[I-D.rescorla-mmusic-ice-trickle]
Rescorla, E., Uberti, J., and E. Ivov, "Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol", draft-rescorla-mmusic-ice-trickle-00 (work in progress), October 2012.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, June 2002.

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, July 2006.

10.2. Informative References

[I-D.ietf-rtcweb-rtp-usage]
Perkins, C., Westerlund, M., and J. Ott, "Web Real-Time Communication (WebRTC): Media Transport and Use of RTP", draft-ietf-rtcweb-rtp-usage-04 (work in progress), July 2012.

[I-D.jennings-rtcweb-signaling]
Jennings, C., Rosenberg, J., and R. Jesup, "RTCWeb Offer/Answer Protocol (ROAP)", draft-jennings-rtcweb-signaling-01 (work in progress), October 2011.

[I-D.nandakumar-rtcweb-sdp]
Nandakumar, S. and C. Jennings, "SDP for the WebRTC", draft-nandakumar-rtcweb-sdp-00 (work in progress), October 2012.

[RFC4568] Andreasen, F., Baugher, M., and D. Wing, "Session Description Protocol (SDP) Security Descriptions for Media Streams", RFC 4568, July 2006.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.

[W3C.WD-webrtc-20111027]
Bergkvist, A., Burnett, D., Narayanan, A., and C. Jennings, "WebRTC 1.0: Real-time Communication Between Browsers", World Wide Web Consortium WD WD-webrtc-20111027, October 2011, <http://www.w3.org/TR/2011/WD-webrtc-20111027>.

Appendix A. JSEP Implementation Examples

A.1. Example API Flows

Below are several sample flows for the new PeerConnection and library APIs, demonstrating when the various APIs are called in different situations and with various transport protocols. For clarity and simplicity, the createOffer/createAnswer calls are assumed to be synchronous in these examples, whereas the actual APIs are async.

A.1.1. Call using ROAP

This example demonstrates a ROAP call, without the use of trickle candidates.

```
// Call is initiated toward Answerer
OffererJS->OffererUA:   pc = new PeerConnection();
OffererJS->OffererUA:   pc.addStream(localStream, null);
OffererUA->OffererJS:   iceCallback(candidate);
OffererJS->OffererUA:   offer = pc.createOffer(null);
OffererJS->OffererUA:   pc.setLocalDescription("offer", offer);
OffererJS->AnswererJS:  {"type":"OFFER", "sdp":offer }

// OFFER arrives at Answerer
AnswererJS->AnswererUA: pc = new PeerConnection();
AnswererJS->AnswererUA: pc.setRemoteDescription("offer", msg.sdp);
AnswererUA->AnswererJS: onaddstream(remoteStream);
AnswererUA->OffererUA:  iceCallback(candidate);

// Answerer accepts call
AnswererJS->AnswererUA: peer.addStream(localStream, null);
AnswererJS->AnswererUA: answer = peer.createAnswer(msg.sdp, null);
AnswererJS->AnswererUA: peer.setLocalDescription("answer", answer);
AnswererJS->OffererJS:  {"type":"ANSWER","sdp":answer }

// ANSWER arrives at Offerer
OffererJS->OffererUA:   peer.setRemoteDescription("answer", answer);
OffererUA->OffererJS:   onaddstream(remoteStream);

// ICE Completes (at Answerer)
AnswererUA->AnswererJS: onopen();
AnswererUA->OffererUA:  Media

// ICE Completes (at Offerer)
OffererUA->OffererJS:   onopen();
OffererJS->AnswererJS:  {"type":"OK" }
OffererUA->AnswererUA:  Media
```

A.1.2. Call using XMPP

This example demonstrates an XMPP call, making use of trickle candidates.

```
// Call is initiated toward Answerer
OffererJS->OffererUA:   pc = new PeerConnection();
OffererJS->OffererUA:   pc.addStream(localStream, null);
OffererJS->OffererUA:   offer = pc.createOffer(null);
OffererJS->OffererUA:   pc.setLocalDescription("offer", offer);
OffererJS:              xmpp = createSessionInitiate(offer);
OffererJS->AnswererJS:  <jingle action="session-initiate"/>

OffererJS->OffererUA:   pc.startIce();
OffererUA->OffererJS:   onicecandidate(cand);
OffererJS:              createTransportInfo(cand);
OffererJS->AnswererJS:  <jingle action="transport-info"/>

// session-initiate arrives at Answerer
AnswererJS->AnswererUA: pc = new PeerConnection();
AnswererJS:             offer = parseSessionInitiate(xmpp);
AnswererJS->AnswererUA: pc.setRemoteDescription("offer", offer);
AnswererUA->AnswererJS: onaddstream(remoteStream);

// transport-infos arrive at Answerer
AnswererJS->AnswererUA: candidate = parseTransportInfo(xmpp);
AnswererJS->AnswererUA: pc.addIceCandidate(candidate);
AnswererUA->AnswererJS: onicecandidate(cand)
AnswererJS:             createTransportInfo(cand);
AnswererJS->OffererJS:  <jingle action="transport-info"/>

// transport-infos arrive at Offerer
OffererJS->OffererUA:   candidates = parseTransportInfo(xmpp);
OffererJS->OffererUA:   pc.addIceCandidate(candidates);

// Answerer accepts call
AnswererJS->AnswererUA: peer.addStream(localStream, null);
AnswererJS->AnswererUA: answer = peer.createAnswer(offer, null);
AnswererJS:             xmpp = createSessionAccept(answer);
AnswererJS->AnswererUA: pc.setLocalDescription("answer", answer);
AnswererJS->OffererJS:  <jingle action="session-accept"/>

// session-accept arrives at Offerer
OffererJS:              answer = parseSessionAccept(xmpp);
OffererJS->OffererUA:   peer.setRemoteDescription("answer", answer);
OffererUA->OffererJS:   onaddstream(remoteStream);

// ICE Completes (at Answerer)
```

```
AnswererUA->AnswererJS: onopen();
AnswererUA->OffererUA:  Media

// ICE Completes (at Offerer)
OffererUA->OffererJS:   onopen();
OffererUA->AnswererUA:  Media
```

A.1.3. Adding video to a call, using XMPP

This example demonstrates an XMPP call, where the XMPP content-add mechanism is used to add video media to an existing session. For simplicity, candidate exchange is not shown.

Note that the offerer for the change to the session may be different than the original call offerer.

```
// Offerer adds video stream
OffererJS->OffererUA:   pc.addStream(videoStream)
OffererJS->OffererUA:   offer = pc.createOffer(null);
OffererJS:              xmpp = createContentAdd(offer);
OffererJS->OffererUA:   pc.setLocalDescription("offer", offer);
OffererJS->AnswererJS:  <jingle action="content-add"/>

// content-add arrives at Answerer
AnswererJS:             offer = parseContentAdd(xmpp);
AnswererJS->AnswererUA: pc.setRemoteDescription("offer", offer);
AnswererJS->AnswererUA: answer = pc.createAnswer(offer, null);
AnswererJS->AnswererUA: pc.setLocalDescription("answer", answer);
AnswererJS:             xmpp = createContentAccept(answer);
AnswererJS->OffererJS:  <jingle action="content-accept"/>

// content-accept arrives at Offerer
OffererJS:              answer = parseContentAccept(xmpp);
OffererJS->OffererUA:   pc.setRemoteDescription("answer", answer);
```

A.1.4. Simultaneous add of video streams, using XMPP

This example demonstrates an XMPP call, where new video sources are added at the same time to a call that already has video; since adding these sources only affects one side of the call, there is no conflict. The XMPP description-info mechanism is used to indicate the new sources to the remote side.

```
// Offerer and "Answerer" add video streams at the same time
OffererJS->OffererUA:   pc.addStream(offererVideoStream2)
OffererJS->OffererUA:   offer = pc.createOffer(null);
OffererJS:              xmpp = createDescriptionInfo(offer);
OffererJS->OffererUA:   pc.setLocalDescription("offer", offer);
OffererJS->AnswererJS:  <jingle action="description-info"/>

AnswererJS->AnswererUA: pc.addStream(answererVideoStream2)
AnswererJS->AnswererUA: offer = pc.createOffer(null);
AnswererJS:             xmpp = createDescriptionInfo(offer);
AnswererJS->AnswererUA: pc.setLocalDescription("offer", offer);
AnswererJS->OffererJS:  <jingle action="description-info"/>

// description-info arrives at "Answerer", and is acked
AnswererJS:             offer = parseDescriptionInfo(xmpp);
AnswererJS->OffererJS:  <iq type="result"/>  // ack

// description-info arrives at Offerer, and is acked
OffererJS:              offer = parseDescriptionInfo(xmpp);
OffererJS->AnswererJS:  <iq type="result"/>  // ack

// ack arrives at Offerer; remote offer is used as an answer
OffererJS->OffererUA:   pc.setRemoteDescription("answer", offer);

// ack arrives at "Answerer"; remote offer is used as an answer
AnswererJS->AnswererUA: pc.setRemoteDescription("answer", offer);
```

A.1.5. Call using SIP

This example demonstrates a simple SIP call (e.g. where the client talks to a SIP proxy over WebSockets).

```
// Call is initiated toward Answerer
OffererJS->OffererUA:   pc = new PeerConnection();
OffererJS->OffererUA:   pc.addStream(localStream, null);
OffererUA->OffererJS:   onicecandidate(candidate);
OffererJS->OffererUA:   offer = pc.createOffer(null);
OffererJS->OffererUA:   pc.setLocalDescription("offer", offer);
OffererJS:              sip = createInvite(offer);
OffererJS->AnswererJS:  SIP INVITE w/ SDP

// INVITE arrives at Answerer
AnswererJS->AnswererUA: pc = new PeerConnection();
AnswererJS:             offer = parseInvite(sip);
AnswererJS->AnswererUA: pc.setRemoteDescription("offer", offer);
AnswererUA->AnswererJS: onaddstream(remoteStream);
AnswererUA->OffererUA:  onicecandidate(candidate);

// Answerer accepts call
AnswererJS->AnswererUA: peer.addStream(localStream, null);
AnswererJS->AnswererUA: answer = peer.createAnswer(offer, null);
AnswererJS:             sip = createResponse(200, answer);
AnswererJS->AnswererUA: peer.setLocalDescription("answer", answer);
AnswererJS->OffererJS:  200 OK w/ SDP

// 200 OK arrives at Offerer
OffererJS:              answer = parseResponse(sip);
OffererJS->OffererUA:   peer.setRemoteDescription("answer", answer);
OffererUA->OffererJS:   onaddstream(remoteStream);
OffererJS->AnswererJS:  ACK

// ICE Completes (at Answerer)
AnswererUA->AnswererJS: onopen();
AnswererUA->OffererUA:  Media

// ICE Completes (at Offerer)
OffererUA->OffererJS:   onopen();
OffererUA->AnswererUA:  Media
```

A.1.6. Handling early media (e.g. 1-800-GO FEDEX), using SIP

This example demonstrates how early media could be handled; for simplicity, only the offerer side of the call is shown.

```
// Call is initiated toward Answerer
OffererJS->OffererUA:   pc = new PeerConnection();
OffererJS->OffererUA:   pc.addStream(localStream, null);
OffererUA->OffererJS:   onicecandidate(candidate);
OffererJS->OffererUA:   offer = pc.createOffer(null);
OffererJS->OffererUA:   pc.setLocalDescription("offer", offer);
OffererJS:              sip = createInvite(offer);
OffererJS->AnswererJS:  SIP INVITE w/ SDP

// 180 Ringing is received by offerer, w/ SDP
OffererJS:              answer = parseResponse(sip);
OffererJS->OffererUA:   pc.setRemoteDescription("pranswer", answer);
OffererUA->OffererJS:   onaddstream(remoteStream);

// ICE Completes (at Offerer)
OffererUA->OffererJS:   onopen();
OffererUA->AnswererUA:  Media

// 200 OK arrives at Offerer
OffererJS:              answer = parseResponse(sip);
OffererJS->OffererUA:   pc.setRemoteDescription("answer", answer);
OffererJS->AnswererJS:  ACK
```

Appendix B. Change log

Changes in draft -02:

- o Converted from nroff
- o Removed comparisons to old approaches abandoned by the working group
- o Removed stuff that has moved to W3C specificaiton
- o Align SDP handling with W3C draft
- o Clarified section on forking.

Changes in draft -01:

- o Added diagrams for architecture and state machine.
- o Added sections on forking and rehydration.
- o Clarified meaning of "pranswer" and "answer".
- o Reworked how ICE restarts and media directions are controlled.
- o Added list of parameters that can be changed in a description.
- o Updated suggested API and examples to match latest thinking.
- o Suggested API and examples have been moved to an appendix.

Changes in draft -00:

- o Migrated from draft-uberti-rtcweb-jsep-02.

Authors' Addresses

Justin Uberti
Google
747 6th Ave S
Kirkland, WA 98033
USA

Email: justin@uberti.name

Cullen Jennings
Cisco
170 West Tasman Drive
San Jose, CA 95134
USA

Email: fluffy@iii.ca