Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 5, 2015

J. Uberti
Google
C. Jennings
Cisco
E. Rescorla, Ed.
Mozilla

July 4, 2014

Javascript Session Establishment Protocol
draft-ietf-rtcweb-jsep-07

Abstract

This document describes the mechanisms for allowing a Javascript application to control the signaling plane of a multimedia session via the interface specified in the W3C RTCPeerConnection API, and discusses how this relates to existing signaling protocols.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes how the W3C WEBRTC RTCPeerConnection interface[W3C.WD-webrtc-20140617] is used to control the setup, management and teardown of a multimedia session.

1.1. General Design of JSEP

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.

With these considerations in mind, this document describes the Javascript Session Establishment Protocol (JSEP) that allows for full control of the signaling state machine from Javascript. JSEP removes the browser almost entirely from the core signaling flow, which is instead handled by the Javascript making use of two interfaces: (1) passing in local and remote session descriptions and (2) interacting with the ICE state machine.

In this document, the use of JSEP is described as if it always occurs between two browsers. Note though in many cases it will actually be between a browser and some kind of server, such as a gateway or MCU. This distinction is invisible to the browser; it just follows the instructions it is given via the API.

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 [RFC5245], 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 session description can be sent immediately and the transport information can be sent when available. 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 ICE checks can start as soon as any transport information is available rather than waiting for all of it.

Through its abstraction of signaling, the JSEP approach does require the application to be aware of the signaling process. While the application does not need to understand the contents of session descriptions to set up a call, the application must call the right APIs at the right times, convert the session descriptions and ICE information into the defined messages of its chosen signaling protocol, and perform the reverse conversion on the messages it receives from the other side.

One way to mitigate this is to provide a Javascript library that hides this complexity from the developer; said library would implement a given signaling protocol along with its state machine and serialization code, presenting a higher level call-oriented interface to the application developer. For example, libraries exist to adapt the JSEP API into an API suitable for a SIP or XMPP. Thus, JSEP provides greater control for the experienced developer without forcing any additional complexity on the novice developer.

1.2. Other Approaches Considered

One approach that was considered instead of JSEP was to include a lightweight signaling protocol. Instead of providing session descriptions to the API, the API would produce and consume messages from this protocol. While providing a more high-level API, this put more control of signaling within the browser, forcing the browser to have to understand and handle concepts like signaling glare. In addition, it prevented the application from driving the state machine to a desired state, as is needed in the page reload case.

A second approach that was considered but not chosen was to decouple the management of the media control objects from session descriptions, instead offering APIs that would control each component directly. This was rejected based on a feeling that requiring

exposure of this level of complexity to the application programmer would not be beneficial; it would result in an API where even a simple example would require a significant amount of code to orchestrate all the needed interactions, as well as creating a large API surface that needed to be agreed upon and documented. In addition, these API points could be called in any order, resulting in a more complex set of interactions with the media subsystem than the JSEP approach, which specifies how session descriptions are to be evaluated and applied.

One variation on JSEP that was considered was to keep the basic session description-oriented API, but to move the mechanism for generating offers and answers out of the browser. 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. Providing createOffer/createAnswer avoids this problem, but still allows applications to generate their own offers/answers (to a large extent) if they choose, using the description generated by createOffer as an indication of the browser's capabilities.

2. 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 [RFC2119].

3. Semantics and Syntax

3.1. Signaling Model

JSEP does not specify a particular signaling model or state machine, other than the generic need to exchange SDP media descriptions in the fashion described by [RFC3264] (offer/answer) 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 browser is totally decoupled from the actual mechanism by which these offers and answers are communicated to the remote side, including addressing, retransmission, forking, and glare

handling. These issues are left entirely up to the application; the application has complete control over which offers and answers get handed to the browser, and when.

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

Figure 1: JSEP Signaling Model

3.2. 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 applies to the local side or the remote side 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 receive, which, when intersected with the set of codecs the remote side supports, specifies what the remote side should send. However, 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 what the remote party should expect to receive; the remote party will have to accept these parameters, with no option to choose a different value. [[OPEN ISSUE: This is not correct because we removed SDES (https://github.com/rtcweb-wg/jsep/issues/10)]]

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. [[OPEN ISSUE: See issue 10 above.]]

Lastly, while the exact media parameters are only known only after an 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 offer 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 setLocalDescription and setRemoteDescription methods 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]).

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 constraint at the signaling level is that only one offer can be outstanding for a given session, but at 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 via the createOffer() method 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:

```
                       setRemote(OFFER)               setLocal(PRANSWER)
                           /-----\                               /-----\
                           |     |                               |     |
                           v     |                               v     |
            +---------------+    |                +---------------+    |
            |               |----/                |               |----/
            |               | setLocal(PRANSWER)  |               |
            |  Remote-Offer |------------------- >| Local-Pranswer|
            |               |                     |               |
            |               |                     |               |
            +---------------+                     +---------------+
                 ^   |                                   |
                 |   | setLocal(ANSWER)                  |
   setRemote(OFFER)  |                                   |
                 |   V                  setLocal(ANSWER) |
            +---------------+                            |
            |               |                            |
            |               |<---------------------------+
            |    Stable     |
            |               |<---------------------------+
            |               |                            |
            +---------------+          setRemote(ANSWER) |
                 ^   |                                   |
                 |   | setLocal(OFFER)                   |
   setRemote(ANSWER) |                                   |
                 |   V                                   |
            +---------------+                     +---------------+
            |               |                     |               |
            |               | setRemote(PRANSWER) |               |
            |  Local-Offer  |------------------- >|Remote-Pranswer|
            |               |                     |               |
            |               |----\                |               |----\
            +---------------+    |                +---------------+    |
                           ^     |                               ^     |
                           |     |                               |     |
                           \-----/                               \-----/
                       setLocal(OFFER)               setRemote(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.

3.3. 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 currently uses 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. In the meantime, Javascript libraries can be used to perform these manipulations.

Note that most applications should be able to treat the SessionDescriptions produced and consumed by these various API calls as opaque blobs; that is, the application will not need to read or change them. The W3C WebRTC API specification will provide appropriate APIs to allow the application to control various session parameters, which will provide the necessary information to the browser about what sort of SessionDescription to produce.

3.4. 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.

3.4.1. 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.ietf-mmusic-trickle-ice]. 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 media setup in cases where gathering is not performed prior to initiating 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.

3.4.1.1. 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 candidate lines.

The 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 "candidate-attribute" field defined for ICE. For example:

```
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, with index N referring to the N+1th m-line in the SDP sent by the entity which sent the IceCandidate. The MID uses the "media stream identification", as defined in [RFC5888], 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.

3.4.2. ICE Candidate Pool

JSEP applications typically inform the browser to begin ICE gathering via the information supplied to setLocalDescription, as this is where the app specifies the number of media streams for which to gather candidates. However, to accelerate cases where the application knows the number of media streams to use ahead of time, it MAY ask the browser to gather a pool of potential ICE candidates to help ensure rapid media setup. When setLocalDescription is eventually called, and the browser goes to gather the needed ICE candidates, it SHOULD start by checking if any candidates are available in the pool. If

there are candidates in the pool, they SHOULD be handed to the application immediately via the ICE candidate callback. If the pool becomes depleted, either because a larger-than-expected number of media streams is used, or because the pool has not had enough time to gather candidates, the remaining candidates are gathered as usual.

3.5. 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 [RFC3261] 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 that is relevant. When forking happens 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, as well as 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:

- o Start exchanging media to a given remote peer, but keep all the resources reserved in the offer.
- o Start exchanging media with a given remote peer, and free any resources in the offer that are not being used.

3.5.1. 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 sequential 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.

3.5.2. 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 [RFC3960]. 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 [RFC3960] describes using UPDATE. 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 [RFC3960].

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. 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.

4.1. Methods

4.1.1. Constructor

The PeerConnection constructor allows the application to specify global parameters for the media session, such as the STUN/TURN servers and credentials to use when gathering candidates. The size of the ICE candidate pool can also be set, if desired; this indicates the number of ICE components to pre-gather candidates for. If the application does not indicate a candidate pool size, the browser may select any default candidate pool size.

In addition, the application can specify its preferred policy regarding use of BUNDLE, the multiplexing mechanism defined in [I-D.ietf-mmusic-sdp-bundle-negotiation]. By specifying a policy from the list below, the application can control how aggressively it will try to BUNDLE media streams together. The set of available policies is as follows:

balanced: The application will BUNDLE all media streams of the same type together. That is, if there are multiple audio and multiple video MediaStreamTracks attached to a PeerConnection, all but the first audio and video tracks will be marked as bundle-only, and candidates will only be gathered for N media streams, where N is the number of distinct media types. When talking to a non-BUNDLE-aware endpoint, only the non-bundle-only streams will be negotiated. This policy balances desire to multiplex with the need to ensure basic audio and video still works in legacy cases. Data channels will be in a separate bundle group.

max-bundle: The application will BUNDLE all of its media streams, including data channels, on a single transport. All streams other than the first will be marked as bundle-only. This policy aims to minimize candidate gathering and maximize multiplexing, at the cost of less compatibility with legacy endpoints.

max-compat: The application will offer BUNDLE, but mark none of its streams as bundle-only. This policy will allow all streams to be received by non-BUNDLE-aware endpoints, but require separate candidates to be gathered for each media stream.

4.1.2. createOffer

The createOffer method generates a blob of SDP that contains a [RFC3264] 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. An options parameter may be supplied to provide additional

control over the generated offer. This options parameter should allow for the following manipulations to be performed:

- o To indicate support for a media type even if no MediaStreamTracks of that type have been added to the session (e.g., an audio call that wants to receive video.)
- o To trigger an ICE restart, for the purpose of reestablishing connectivity.

In the initial offer, the generated SDP will contain all desired functionality for the session (functionality that is supported but not desired by default may be omitted); for each SDP line, the generation of the SDP will follow the process defined for generating an initial offer from the document that specifies the given SDP line. The exact handling of initial offer generation is detailed in Section 5.2.1 below.

In the event createOffer is called after the session is established, createOffer will generate an offer to modify the current session based on any changes that have been made to the session, e.g. adding or removing MediaStreams, or requesting an ICE restart. For each existing stream, the generation of each SDP line must follow the process defined for generating an updated offer from the RFC that specifies the given SDP line. For each new stream, the generation of the SDP must follow the process of generating an initial offer, as mentioned above. If no changes have been made, or for SDP lines that are unaffected by the requested changes, the offer will only contain the parameters negotiated by the last offer-answer exchange. The exact handling of subsequent offer generation is detailed in Section 5.2.2. below.

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 result in candidate gathering, or cause media to start or stop flowing.

4.1.3. createAnswer

The createAnswer method generates a blob of SDP that contains a [RFC3264] SDP answer with the supported configuration for the session

that is compatible with the parameters supplied in the most recent call to setRemoteDescription, which MUST have been called prior to calling createAnswer. 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. An options 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; for each SDP line, the generation of the SDP must follow the process defined for generating an answer from the document that specifies the given SDP line. The exact handling of answer generation is detailed in Section 5.3. below.

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 trigger candidate gathering or change media state.

4.1.4. 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 supplied 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 "pranswer".

"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 only difference between a provisional and final answer is that the final answer results in the freeing of any unused resources that were allocated as a result of the offer. As such, the application can use some discretion on whether an answer should be applied as provisional or final, and can change the type of the session description as needed. For example, in a serial forking scenario, an application may receive multiple "final" answers, one from each remote endpoint. The application could choose to 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).

"rollback" is a special session description type implying that the state machine should be rolled back to the previous state, as described in Section 4.1.4.2. The contents MUST be empty.

4.1.4.1. Use of Provisional Answers

Most web applications will not need to create answers using the "pranswer" type. While it is good practice to send an immediate response to an "offer", in order to warm up the session transport and prevent media clipping, the preferred handling for a web application would be to create and send an "inactive" final answer immediately after receiving the offer. Later, when the called user actually accepts the call, the application can create a new "sendrecv" offer to update the previous offer/answer pair and start the media flow. While this could also be done with an inactive "pranswer", followed by a sendrecv "answer", the initial "pranswer" leaves the offer-answer exchange open, which means that neither side can send an updated offer during this time.

As an example, 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 inactive or receive-only. It could then ask the user to accept the call, 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 accepted the call and triggered the new offer, it is likely that the ICE and DTLS handshaking for all the channels will already have finished.

Of course, some applications may not be able to perform this double

offer-answer exchange, particularly ones that are attempting to gateway to legacy signaling protocols. In these cases, "pranswer" can still provide the application with a mechanism to warm up the transport.

4.1.4.2. Rollback

In certain situations it may be desirable to "undo" a change made to setLocalDescription or setRemoteDescription. Consider a case where a call is ongoing, and one side wants to change some of the session parameters; that side generates an updated offer and then calls setLocalDescription. However, the remote side, either before or after setRemoteDescription, decides it does not want to accept the new parameters, and sends a reject message back to the offerer. Now, the offerer, and possibly the answerer as well, need to return to a stable state and the previous local/remote description. To support this, we introduce the concept of "rollback".

A rollback discards any proposed changes to the session, returning the state machine to the stable state, and setting the modified local and/or remote description back to their previous values. Any resources or candidates that were allocated by the abandoned local description are discarded; any media that is received will be processed according to the previous local and remote descriptions. Rollback can only be used to cancel proposed changes; there is no support for rolling back from a stable state to a previous stable state. Note that this implies that once the answerer has performed setLocalDescription with his answer, this cannot be rolled back.

A rollback is performed by supplying a session description of type "rollback" with empty contents to either setLocalDescription or setRemoteDescription, depending on which was most recently used (i.e. if the new offer was supplied to setLocalDescription, the rollback should be done using setLocalDescription as well).

4.1.5. 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.

This API indirectly controls the candidate gathering process. When a local description is supplied, and the number of transports currently in use does not match the number of transports needed by the local description, the PeerConnection will create transports as needed and begin gathering candidates for them.

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

4.1.6. 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 previously called with an offer, and setLocalDescription is called with an answer (provisional or final), and the media directions are compatible, and media are available to send, this will result in the starting of media transmission.

4.1.7. 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.

[[OPEN ISSUE: Do we need to expose accessors for both the current and proposed local description? https://github.com/rtcweb-wg/jsep/issues/16]]

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

4.1.8. 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.

[[OPEN ISSUE: Do we need to expose accessors for both the current and proposed remote description? https://github.com/rtcweb-wg/jsep/issues/16]]

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

4.1.9. 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 onicecandidate callback 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.

4.1.10. addIceCandidate

The addIceCandidate method provides a remote candidate to the ICE Agent, which, if parsed successfully, will be added to the remote description according to the rules defined for Trickle ICE. 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.

5. SDP Interaction Procedures

This section describes the specific procedures to be followed when creating and parsing SDP objects.

5.1. Requirements Overview

JSEP implementations must comply with the specifications listed below that govern the creation and processing of offers and answers.

The first set of specifications is the "mandatory-to-implement" set. All implementations must support these behaviors, but may not use all of them if the remote side, which may not be a JSEP endpoint, does not support them.

The second set of specifications is the "mandatory-to-use" set. The local JSEP endpoint and any remote endpoint must indicate support for these specifications in their session descriptions.

5.1.1. Implementation Requirements

This list of mandatory-to-implement specifications is derived from the requirements outlined in [I-D.ietf-rtcweb-rtp-usage].

R-1 [RFC4566] is the base SDP specification and MUST be implemented.

R-2 [RFC5764] MUST be supported for signaling the UDP/TLS/RTP/SAVPF RTP profile.

R-3 [RFC5245] MUST be implemented for signaling the ICE credentials and candidate lines corresponding to each media stream. The ICE implementation MUST be a Full implementation, not a Lite implementation.

R-4 [RFC5763] MUST be implemented to signal DTLS certificate fingerprints.

R-5 [RFC4568] MUST NOT be implemented to signal SDES SRTP keying information.

R-6 The [RFC5888] grouping framework MUST be implemented for signaling grouping information, and MUST be used to identify m= lines via the a=mid attribute.

R-7 [I-D.ietf-mmusic-msid] MUST be supported, in order to signal associations between RTP objects and W3C MediaStreams and MediaStreamTracks in a standard way.

R-8 The bundle mechanism in [I-D.ietf-mmusic-sdp-bundle-negotiation] MUST be supported to signal the ability to multiplex RTP streams on a single UDP port, in order to avoid excessive use of port number resources.

R-9 The SDP attributes of "sendonly", "recvonly", "inactive", and "sendrecv" from [RFC4566] MUST be implemented to signal information about media direction.

R-10 [RFC5576] MUST be implemented to signal RTP SSRC values.

R-11 [RFC4585] MUST be implemented to signal RTCP based feedback.

R-12 [RFC5761] MUST be implemented to signal multiplexing of RTP and RTCP.

R-13 [RFC5506] MUST be implemented to signal reduced-size RTCP messages.

R-14 [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 [RFC4566], Section 5.13, JSEP implementations MUST ignore unknown attribute (a=) lines.

5.1.2. Usage Requirements

All session descriptions handled by JSEP endpoints, both local and remote, MUST indicate support for the following specifications. If any of these are absent, this omission MUST be treated as an error.

R-1 Either the UDP/TLS/RTP/SAVP or the UDP/TLS/RTP/SAVPF RTP profile, as specified in [RFC5764], MUST be used.

R-2 ICE, as specified in [RFC5245], MUST be used. Note that the remote endpoint may use a Lite implementation; implementations MUST properly handle remote endpoints which do ICE-Lite.

R-3 DTLS-SRTP, as specified in [RFC5763], MUST be used.

5.2. Constructing an Offer

When createOffer is called, a new SDP description must be created that includes the functionality specified in [I-D.ietf-rtcweb-rtp-usage]. The exact details of this process are explained below.

5.2.1. Initial Offers

When createOffer is called for the first time, the result is known as the initial offer.

The first step in generating an initial offer is to generate session-level attributes, as specified in [RFC4566], Section 5. Specifically:

- o The first SDP line MUST be "v=0", as specified in [RFC4566], Section 5.1
- o The second SDP line MUST be an "o=" line, as specified in [RFC4566], Section 5.2. The value of the <username> field SHOULD be "-". The value of the <sess-id> field SHOULD be a cryptographically random number. To ensure uniqueness, this number SHOULD be at least 64 bits long. The value of the <sess-version> field SHOULD be zero. The value of the <nettype>

<addrtype> <unicast-address> tuple SHOULD be set to a non-meaningful address, such as IN IP4 0.0.0.0, to prevent leaking the local address in this field. As mentioned in [RFC4566], the entire o= line needs to be unique, but selecting a random number for <sess-id> is sufficient to accomplish this.

- o The third SDP line MUST be a "s=" line, as specified in [RFC4566], Section 5.3; to match the "o=" line, a single dash SHOULD be used as the session name, e.g. "s=-". Note that this differs from the advice in [RFC4566] which proposes a single space, but as both "o=" and "s=" are meaningless, having the same meaningless value seems clearer.
- o Session Information ("i="), URI ("u="), Email Address ("e="), Phone Number ("p="), Bandwidth ("b="), Repeat Times ("r="), and Time Zones ("z=") lines are not useful in this context and SHOULD NOT be included.
- o Encryption Keys ("k=") lines do not provide sufficient security and MUST NOT be included.
- o A "t=" line MUST be added, as specified in [RFC4566], Section 5.9; both <start-time> and <stop-time> SHOULD be set to zero, e.g. "t=0 0".
- o An "a=msid-semantic:WMS" line MUST be added, as specified in [I-D.ietf-mmusic-msid], Section 4.

The next step is to generate m= sections, as specified in [RFC4566] Section 5.14, for each MediaStreamTrack that has been added to the PeerConnection via the addStream method. (Note that this method takes a MediaStream, which can contain multiple MediaStreamTracks, and therefore multiple m= sections can be generated even if addStream is only called once.) m=sections MUST be sorted first by the order in which the MediaStreams were added to the PeerConnection, and then by the alphabetical ordering of the media type for the MediaStreamTrack. For example, if a MediaStream containing both an audio and a video MediaStreamTrack is added to a PeerConnection, the resultant m=audio section will precede the m=video section. If a second MediaStream containing an audio MediaStreamTrack was added, it would follow the m=video section.

Each m= section, provided it is not being bundled into another m= section, MUST generate a unique set of ICE credentials and gather its own unique set of ICE candidates. Otherwise, it MUST use the same ICE credentials and candidates as the m= section into which it is being bundled. Note that this means that for offers, any m= sections which are not bundle-only MUST have unique ICE credentials and candidates, since it is possible that the answerer will accept them without bundling them.

For DTLS, all m= sections MUST use the certificate for the identity that has been specified for the PeerConnection; as a result, they

MUST all have the same [RFC4572] fingerprint value, or this value MUST be a session-level attribute.

Each m= section should be generated as specified in [RFC4566], Section 5.14. For the m= line itself, the following rules MUST be followed:

- o The port value is set to the port of the default ICE candidate for this m= section; if this m= section is not being bundled into another m= section, the port value MUST be unique. If no candidates have yet been gathered, and a 'null' port value is being used, as indicated in [I-D.ietf-mmusic-trickle-ice], Section 5.1., this port MUST still be unique.
- o To properly indicate use of DTLS, the <proto> field MUST be set to "UDP/TLS/RTP/SAVPF", as specified in [RFC5764], Section 8.

Each m= section MUST include the following attribute lines:

- o An "a=mid" line, as specified in [RFC5888], Section 4.
- o An "a=msid" line, as specified in [I-D.ietf-mmusic-msid], Section 2.
- o [OPEN ISSUE: Use of AppID]
- o An "a=sendrecv" line, as specified in [RFC3264], Section 5.1.
- o For each supported codec, "a=rtpmap" and "a=fmtp" lines, as specified in [RFC4566], Section 6. For audio, the codecs specified in [I-D.ietf-rtcweb-audio], Section 3, MUST be be supported.
- o For each primary codec where RTP retransmission should be used, a corresponding "a=rtpmap" line indicating "rtx" with the clock rate of the primary codec and an "a=fmtp" line that references the payload type of the primary codec, as specified in [RFC4588], Section 8.1.
- o For each supported FEC mechanism, a corresponding "a=rtpmap" line indicating the desired FEC codec.
- o "a=ice-ufrag" and "a=ice-passwd" lines, as specified in [RFC5245], Section 15.4.
- o An "a=ice-options" line, with the "trickle" option, as specified in [I-D.ietf-mmusic-trickle-ice], Section 4.
- o For each candidate that has been gathered during the most recent gathering phase, an "a=candidate" line, as specified in [RFC5245], Section 4.3., paragraph 3.
- o For the current default candidate, a "c=" line, as specified in [RFC5245], Section 4.3., paragraph 6. If no candidates have been gathered yet, the default candidate should be set to the 'null' value defined in [I-D.ietf-mmusic-trickle-ice], Section 5.1.
- o An "a=fingerprint" line, as specified in [RFC4572], Section 5; the algorithm used for the fingerprint MUST match that used in the certificate signature.

- An "a=setup" line, as specified in [RFC4145], Section 4, and clarified for use in DTLS-SRTP scenarios in [RFC5763], Section 5. The role value in the offer MUST be "actpass".
- An "a=rtcp-mux" line, as specified in [RFC5761], Section 5.1.1.
- An "a=rtcp-rsize" line, as specified in [RFC5506], Section 5.
- For each supported RTP header extension, an "a=extmap" line, as specified in [RFC5285], Section 5. The list of header extensions that SHOULD/MUST be supported is specified in [I-D.ietf-rtcweb-rtp-usage], Section 5.2. Any header extensions that require encryption MUST be specified as indicated in [RFC6904], Section 4.
- For each supported RTCP feedback mechanism, an "a=rtcp-fb" mechanism, as specified in [RFC4585], Section 4.2. The list of RTCP feedback mechanisms that SHOULD/MUST be supported is specified in [I-D.ietf-rtcweb-rtp-usage], Section 5.1.
- An "a=ssrc" line, as specified in [RFC5576], Section 4.1, indicating the SSRC to be used for sending media, along with the mandatory "cname" source attribute, as specified in Section 6.1, indicating the CNAME for the source. The CNAME must be generated in accordance with [RFC7022]. [OPEN ISSUE: How are CNAMEs specified for MSTs? Are they randomly generated for each MediaStream? If so, can two MediaStreams be synced? See: https://github.com/rtcweb-wg/jsep/issues/4]
- If RTX is supported for this media type, another "a=ssrc" line with the RTX SSRC, and an "a=ssrc-group" line, as specified in [RFC5576], section 4.2, with semantics set to "FID" and including the primary and RTX SSRCs.
- If FEC is supported for this media type, another "a=ssrc" line with the FEC SSRC, and an "a=ssrc-group" line, as specified in [RFC5576], section 4.2, with semantics set to "FEC" and including the primary and FEC SSRCs.
- [OPEN ISSUE: Handling of a=imageattr]
- If the BUNDLE policy for this PeerConnection is set to "max-bundle", and this is not the first m= section, or the BUNDLE policy is set to "default", and this is not the first m= section for this media type, an "a=bundle-only" line.

Lastly, if a data channel has been created, a m= section MUST be generated for data. The <media> field MUST be set to "application" and the <proto> field MUST be set to "DTLS/SCTP", as specified in [I-D.ietf-mmusic-sctp-sdp], Section 3; the "fmt" value MUST be set to the SCTP port number, as specified in Section 4.1.

Within the data m= section, the "a=mid", "a=ice-ufrag", "a=ice-passwd", "a=ice-options", "a=candidate", "a=fingerprint", and "a=setup" lines MUST be included as mentioned above, along with an "a=sctpmap" line referencing the SCTP port number and specifying the application protocol indicated in [I-D.ietf-rtcweb-data-protocol].

[OPEN ISSUE: the -01 of this document is missing this information.]

Once all m= sections have been generated, a session-level "a=group" attribute MUST be added as specified in [RFC5888]. This attribute MUST have semantics "BUNDLE", and MUST include the mid identifiers of each m= section. The effect of this is that the browser offers all m= sections as one BUNDLE group. However, whether the m= sections are bundle-only or not depends on the BUNDLE policy.

Attributes which SDP permits to either be at the session level or the media level SHOULD generally be at the media level even if they are identical. This promotes readability, especially if one of a set of initially identical attributes is subsequently changed.

Attributes other than the ones specified above MAY be included, except for the following attributes which are specifically incompatible with the requirements of [I-D.ietf-rtcweb-rtp-usage], and MUST NOT be included:

- o "a=crypto"
- o "a=key-mgmt"
- o "a=ice-lite"

Note that when BUNDLE is used, any additional attributes that are added MUST follow the advice in [I-D.ietf-mmusic-sdp-mux-attributes] on how those attributes interact with BUNDLE.

Note that these requirements are in some cases stricter than those of SDP. Implementations MUST be prepared to accept compliant SDP even if it would not conform to the requirements for generating SDP in this specification.

5.2.2. Subsequent Offers

When createOffer is called a second (or later) time, or is called after a local description has already been installed, the processing is somewhat different than for an initial offer.

If the initial offer was not applied using setLocalDescription, meaning the PeerConnection is still in the "stable" state, the steps for generating an initial offer should be followed, subject to the following restriction:

- o The fields of the "o=" line MUST stay the same except for the <session-version> field, which MUST increment if the session description changes in any way, including the addition of ICE candidates.

If the initial offer was applied using setLocalDescription, but an answer from the remote side has not yet been applied, meaning the

PeerConnection is still in the "local-offer" state, an offer is generated by following the steps in the "stable" state above, along with these exceptions:

- o The "s=" and "t=" lines MUST stay the same.
- o Each "a=mid" line MUST stay the same.
- o Each "a=ice-ufrag" and "a=ice-pwd" line MUST stay the same unless the "IceRestart" option (Section 5.2.3 was specified. Note that it's not clear why you would actually want to do this, since at this point ICE has not yet started and is thus unlikely to need a restart.
- o For MediaStreamTracks that are still present, the "a=msid", "a=ssrc", and "a=ssrc-group" lines MUST stay the same.
- o If any MediaStreamTracks have been removed, either through the removeStream method or by removing them from an added MediaStream, their m= sections MUST be marked as recvonly by changing the value of the [RFC3264] directional attribute to "a=recvonly". The "a=msid", "a=ssrc", and "a=ssrc-group" lines MUST be removed from the associated m= sections.
- o If any MediaStreamTracks have been added, and there exist m= sections of the appropriate media type with no associated MediaStreamTracks (i.e. as described in the preceding paragraph), those m= sections MUST be recycled by adding the new MediaStreamTrack to the m= section. This is done by adding the necessary "a=msid", "a=ssrc", and "a=ssrc-group" lines to the recycled m= section, and removing the "a=recvonly" attribute.

If the initial offer was applied using setLocalDescription, and an answer from the remote side has been applied using setRemoteDescription, meaning the PeerConnection is in the "remote-pranswer" or "stable" states, an offer is generated based on the negotiated session descriptions by following the steps mentioned for the "local-offer" state above, along with these exceptions: [OPEN ISSUE: should this be permitted in the remote-pranswer state?]

- o If a m= section exists in the current local description, but does not have an associated local MediaStreamTrack (possibly because said MediaStreamTrack was removed since the last exchange), a m= section MUST still be generated in the new offer, as indicated in [RFC3264], Section 8. The disposition of this section will depend on the state of the remote MediaStreamTrack associated with this m= section. If one exists, and it is still in the "live" state, the new m= section MUST be marked as "a=recvonly", with no "a=msid" or related attributes present. If no remote MediaStreamTrack exists, or it is in the "ended" state, the m= section MUST be marked as rejected, by setting the port to zero, as indicated in [RFC3264], Section 8.2.
- o If any MediaStreamTracks have been added, and there exist recvonly m= sections of the appropriate media type with no associated MediaStreamTracks, or rejected m= sections of any media type,

those m= sections MUST be recycled, and a local MediaStreamTrack associated with these recycled m= sections until all such existing m= sections have been used. This includes any recvonly or rejected m= sections created by the preceding paragraph.

In addition, for each non-recycled, non-rejected m= section in the new offer, the following adjustments are made based on the contents of the corresponding m= section in the current remote description:

- o The m= line and corresponding "a=rtpmap" and "a=fmtp" lines MUST only include codecs present in the remote description.
- o The RTP header extensions MUST only include those that are present in the remote description.
- o The RTCP feedback extensions MUST only include those that are present in the remote description.
- o The "a=rtcp-mux" line MUST only be added if present in the remote description.
- o The "a=rtcp-rsize" line MUST only be added if present in the remote description.

The "a=group:BUNDLE" attribute MUST include the mid identifiers specified in the BUNDLE group in the most recent answer, minus any m= sections that have been marked as rejected, plus any newly added or re-enabled m= sections. In other words, the BUNDLE attribute must contain all m= sections that were previously bundled, as long as they are still alive, as well as any new m= sections.

5.2.3. Options Handling

The createOffer method takes as a parameter an RTCOfferOptions object. Special processing is performed when generating a SDP description if the following constraints are present.

5.2.3.1. OfferToReceiveAudio

If the "OfferToReceiveAudio" option is specified, with an integer value of N, the offer MUST include N non-rejected m= sections with media type "audio", even if fewer than N audio MediaStreamTracks have been added to the PeerConnection. This allows the offerer to receive audio, including multiple independent streams, even when not sending it; accordingly, the directional attribute on the audio m= sections without associated MediaStreamTracks MUST be set to recvonly. If this option is specified in the case where at least N audio MediaStreamTracks have already been added to the PeerConnection, or N non-rejected m= sections with media type "audio" would otherwise be generated, it has no effect. For backwards compatibility, a value of "true" is interpreted as equivalent to N=1.

5.2.3.2. OfferToReceiveVideo

If the "OfferToReceiveVideo" option is specified, with an integer value of N, the offer MUST include N non-rejected m= sections with media type "video", even if fewer than N video MediaStreamTracks have been added to the PeerConnection. This allows the offerer to receive video, including multiple independent streams, even when not sending it; accordingly, the directional attribute on the video m= sections without associated MediaStreamTracks MUST be set to recvonly. If this option is specified in the case where at least N video MediaStreamTracks have already been added to the PeerConnection, or N non-rejected m= sections with media type "video" would otherwise be generated, it has no effect. For backwards compatibility, a value of "true" is interpreted as equivalent to N=1.

5.2.3.3. VoiceActivityDetection

If the "VoiceActivityDetection" option is specified, with a value of "true", the offer MUST indicate support for silence suppression in the audio it receives by including comfort noise ("CN") codecs for each offered audio codec, as specified in [RFC3389], Section 5.1, except for codecs that have their own internal silence suppression support. For codecs that have their own internal silence suppression support, the appropriate fmtp parameters for that codec MUST be specified to indicate that silence suppression for received audio is desired. For example, when using the Opus codec, the "usedtx=1" parameter would be specified in the offer. This option allows the endpoint to significantly reduce the amount of audio bandwidth it receives, at the cost of some fidelity, depending on the quality of the remote VAD algorithm.

5.2.3.4. IceRestart

If the "IceRestart" option is specified, with a value of "true", the offer MUST indicate an ICE restart by generating new ICE ufrag and pwd attributes, as specified in RFC5245, Section 9.1.1.1. If this option is specified on an initial offer, it has no effect (since a new ICE ufrag and pwd are already generated). This option is useful for reestablishing connectivity in cases where failures are detected.

5.3. Generating an Answer

When createAnswer is called, a new SDP description must be created that is compatible with the supplied remote description as well as the requirements specified in [I-D.ietf-rtcweb-rtp-usage]. The exact details of this process are explained below.

5.3.1. Initial Answers

When createAnswer is called for the first time after a remote description has been provided, the result is known as the initial answer. If no remote description has been installed, an answer cannot be generated, and an error MUST be returned.

Note that the remote description SDP may not have been created by a JSEP endpoint and may not conform to all the requirements listed in Section 5.2. For many cases, this is not a problem. However, if any mandatory SDP attributes are missing, or functionality listed as mandatory-to-use above is not present, this MUST be treated as an error, and MUST cause the affected m= sections to be marked as rejected.

The first step in generating an initial answer is to generate session-level attributes. The process here is identical to that indicated in the Initial Offers section above.

The next step is to generate m= sections for each m= section that is present in the remote offer, as specified in [RFC3264], Section 6. For the purposes of this discussion, any session-level attributes in the offer that are also valid as media-level attributes SHALL be considered to be present in each m= section.

The next step is to go through each offered m= section. If there is a local MediaStreamTrack of the same type which has been added to the PeerConnection via addStream and not yet associated with a m= section, and the specific m= section is either sendrecv or recvonly, the MediaStreamTrack will be associated with the m= section at this time. MediaStreamTracks are assigned to m= sections using the canonical order described in Section 5.2.1. If there are more m= sections of a certain type than MediaStreamTracks, some m= sections will not have an associated MediaStreamTrack. If there are more MediaStreamTracks of a certain type than compatible m= sections, only the first N MediaStreamTracks will be able to be associated in the constructed answer. The remainder will need to be associated in a subsequent offer.

For each offered m= section, if the associated remote MediaStreamTrack has been stopped, and is therefore in state "ended", and no local MediaStreamTrack has been associated, the corresponding m= section in the answer MUST be marked as rejected by setting the port in the m= line to zero, as indicated in [RFC3264], Section 6., and further processing for this m= section can be skipped.

Provided that is not the case, each m= section in the answer should then be generated as specified in [RFC3264], Section 6.1. Because

use of DTLS is mandatory, the <proto> field MUST be set to "UDP/TLS/RTP/SAVPF". If the offer supports BUNDLE, all m= sections to be BUNDLEd must use the same ICE credentials and candidates; all m= sections not being BUNDLEd must use unique ICE credentials and candidates. Each m= section MUST include the following:

- o If present in the offer, an "a=mid" line, as specified in [RFC5888], Section 9.1. The "mid" value MUST match that specified in the offer.
- o If a local MediaStreamTrack has been associated, an "a=msid" line, as specified in [I-D.ietf-mmusic-msid], Section 2.
- o [OPEN ISSUE: Use of AppID]
- o Depending on the directionality of the offer, the disposition of any associated remote MediaStreamTrack, and the presence of an associated local MediaStreamTrack, the appropriate directionality attribute, as specified in [RFC3264], Section 6.1. If the offer was sendrecv, and the remote MediaStreamTrack is still "live", and there is a local MediaStreamTrack that has been associated, the directionality MUST be set as sendrecv. If the offer was sendonly, and the remote MediaStreamTrack is still "live", the directionality MUST be set as recvonly. If the offer was recvonly, and a local MediaStreamTrack has been associated, the directionality MUST be set as sendonly. If the offer was inactive, the directionality MUST be set as inactive.
- o For each supported codec that is present in the offer, "a=rtpmap" and "a=fmtp" lines, as specified in [RFC4566], Section 6, and [RFC3264], Section 6.1. For audio, the codecs specified in [I-D.ietf-rtcweb-audio], Section 3, MUST be supported. Note that for simplicity, the answerer MAY use different payload types for codecs than the offerer, as it is not prohibited by Section 6.1.
- o If "rtx" is present in the offer, for each primary codec where RTP retransmission should be used, a corresponding "a=rtpmap" line indicating "rtx" with the clock rate of the primary codec and an "a=fmtp" line that references the payload type of the primary codec, as specified in [RFC4588], Section 8.1.
- o For each supported FEC mechanism that is present in the offer, a corresponding "a=rtpmap" line indicating the desired FEC codec.
- o "a=ice-ufrag" and "a=ice-passwd" lines, as specified in [RFC5245], Section 15.4.
- o If the "trickle" ICE option is present in the offer, an "a=ice-options" line, with the "trickle" option, as specified in [I-D.ietf-mmusic-trickle-ice], Section 4.
- o For each candidate that has been gathered during the most recent gathering phase, an "a=candidate" line, as specified in [RFC5245], Section 4.3., paragraph 3.
- o For the current default candidate, a "c=" line, as specified in [RFC5245], Section 4.3., paragraph 6. If no candidates have been gathered yet, the default candidate should be set to the 'null' value defined in [I-D.ietf-mmusic-trickle-ice], Section 5.1.

- o An "a=fingerprint" line, as specified in [RFC4572], Section 5; the algorithm used for the fingerprint MUST match that used in the certificate signature.
- o An "a=setup" line, as specified in [RFC4145], Section 4, and clarified for use in DTLS-SRTP scenarios in [RFC5763], Section 5. The role value in the answer MUST be "active" or "passive"; the "active" role is RECOMMENDED.
- o If present in the offer, an "a=rtcp-mux" line, as specified in [RFC5761], Section 5.1.1.
- o If present in the offer, an "a=rtcp-rsize" line, as specified in [RFC5506], Section 5.
- o For each supported RTP header extension that is present in the offer, an "a=extmap" line, as specified in [RFC5285], Section 5. The list of header extensions that SHOULD/MUST be supported is specified in [I-D.ietf-rtcweb-rtp-usage], Section 5.2. Any header extensions that require encryption MUST be specified as indicated in [RFC6904], Section 4.
- o For each supported RTCP feedback mechanism that is present in the offer, an "a=rtcp-fb" mechanism, as specified in [RFC4585], Section 4.2. The list of RTCP feedback mechanisms that SHOULD/ MUST be supported is specified in [I-D.ietf-rtcweb-rtp-usage], Section 5.1.
- o If a local MediaStreamTrack has been associated, an "a=ssrc" line, as specified in [RFC5576], Section 4.1, indicating the SSRC to be used for sending media.
- o If a local MediaStreamTrack has been associated, and RTX has been negotiated for this m= section, another "a=ssrc" line with the RTX SSRC, and an "a=ssrc-group" line, as specified in [RFC5576], section 4.2, with semantics set to "FID" and including the primary and RTX SSRCs.
- o If a local MediaStreamTrack has been associated, and FEC has been negotiated for this m= section, another "a=ssrc" line with the FEC SSRC, and an "a=ssrc-group" line, as specified in [RFC5576], section 4.2, with semantics set to "FEC" and including the primary and FEC SSRCs.
- o [OPEN ISSUE: Handling of a=imageattr]

If a data channel m= section has been offered, a m= section MUST also be generated for data. The <media> field MUST be set to "application" and the <proto> field MUST be set to "DTLS/SCTP", as specified in [I-D.ietf-mmusic-sctp-sdp], Section 3; the "fmt" value MUST be set to the SCTP port number, as specified in Section 4.1.

Within the data m= section, the "a=mid", "a=ice-ufrag", "a=ice-passwd", "a=ice-options", "a=candidate", "a=fingerprint", and "a=setup" lines MUST be included as mentioned above, along with an "a=sctpmap" line referencing the SCTP port number and specifying the application protocol indicated in [I-D.ietf-rtcweb-data-protocol].

[OPEN ISSUE: the -01 of this document is missing this information.]

If "a=group" attributes with semantics of "BUNDLE" are offered, corresponding session-level "a=group" attributes MUST be added as specified in [RFC5888]. These attributes MUST have semantics "BUNDLE", and MUST include the all mid identifiers from the offered BUNDLE groups that have not been rejected. Note that regardless of the presence of "a=bundle-only" in the offer, no m= sections in the answer should have an "a=bundle-only" line.

Attributes that are common between all m= sections MAY be moved to session-level, if explicitly defined to be valid at session-level.

The attributes prohibited in the creation of offers are also prohibited in the creation of answers.

5.3.2. Subsequent Answers

5.3.3. Options Handling

5.4. Parsing an Offer

5.5. Parsing an Answer

5.6. Applying a Local Description

5.7. Applying a Remote Description

6. Configurable SDP Parameters

It is possible to change elements in the SDP returned from createOffer before passing it to setLocalDescription. When an implementation receives modified SDP it MUST either:

- o Accept the changes and adjust its behavior to match the SDP.
- o Reject the changes and return an error via the error callback.

Changes MUST NOT be silently ignored.

The following elements of the SDP media description MUST NOT be changed between the createOffer and the setLocalDescription, since they reflect transport attributes that are solely under browser control, and the browser MUST NOT honor an attempt to change them:

- o The number, type and port number of m-lines.

- The generated ICE credentials (a=ice-ufrag and a=ice-pwd).
- The set of ICE candidates and their parameters (a=candidate).

The following modifications, if done by the browser to a description between createOffer/createAnswer and the setLocalDescription, MUST be honored by the browser:

- Remove or reorder codecs (m=)

The following parameters may be controlled by constraints passed into createOffer/createAnswer. As an open issue, these changes may also be be performed by manipulating the SDP returned from createOffer/ createAnswer, as indicated above, as long as the capabilities of the endpoint are not exceeded (e.g. asking for a resolution greater than what the endpoint can encode):

- [[OPEN ISSUE: This is a placeholder for other modifications, which we may continue adding as use cases appear.]]

Implementations MAY choose to either honor or reject any elements not listed in the above two categories, but must do so explicitly as described at the beginning of this section. Note that future standards may add new SDP elements to the list of elements which must be accepted or rejected, but due to version skew, applications must be prepared for implementations to accept changes which must be rejected and vice versa.

The application can also modify the SDP to reduce the capabilities in the offer it sends to the far side or the offer that it installs from the far side in any way the application sees fit, as long as it is a valid SDP offer and specifies a subset of what was in the original offer. This is safe because the answer is not permitted to expand capabilities and therefore will just respond to what is actually in the offer.

As always, the application is solely responsible for what it sends to the other party, and all incoming SDP will be processed by the browser to the extent of its capabilities. It is an error to assume that all SDP is well-formed; however, one should be able to assume that any implementation of this specification will be able to process, as a remote offer or answer, unmodified SDP coming from any other implementation of this specification.

7. Security Considerations

The IETF has published separate documents [I-D.ietf-rtcweb-security-arch] [I-D.ietf-rtcweb-security] describing

the security architecture for WebRTC as a whole. The remainder of this section describes security considerations for this document.

While formally the JSEP interface is an API, it is better to think of it is an Internet protocol, with the JS being untrustworthy from the perspective of the browser. Thus, the threat model of [RFC3552] applies. In particular, JS can call the API in any order and with any inputs, including malicious ones. This is particularly relevant when we consider the SDP which is passed to setLocalDescription(). While correct API usage requires that the application pass in SDP which was derived from createOffer() or createAnswer() (perhaps suitably modified as described in Section 6, there is no guarantee that applications do so. The browser MUST be prepared for the JS to pass in bogus data instead.

Conversely, the application programmer MUST recognize that the JS does not have complete control of browser behavior. One case that bears particular mention is that editing ICE candidates out of the SDP or suppressing trickled candidates does not have the expected behavior: implementations will still perform checks from those candidates even if they are not sent to the other side. Thus, for instance, it is not possible to prevent the remote peer from learning your public IP address by removing server reflexive candidates. Applications which wish to conceal their public IP address should instead configure the ICE agent to use only relay candidates.

8. IANA Considerations

This document requires no actions from IANA.

9. Acknowledgements

Significant text incorporated in the draft as well and review was provided by Harald Alvestrand and Suhas Nandakumar. Dan Burnett, Neil Stratford, Eric Rescorla, Anant Narayanan, Andrew Hutton, Richard Ejzak, Adam Bergkvist and Matthew Kaufman all provided valuable feedback on this proposal.

10. References

10.1. Normative References

[I-D.ietf-mmusic-msid]
Alvestrand, H., "Cross Session Stream Identification in the Session Description Protocol",

draft-ietf-mmusic-msid-01 (work in progress), August 2013.

[I-D.ietf-mmusic-sctp-sdp]
Loreto, S. and G. Camarillo, "Stream Control Transmission Protocol (SCTP)-Based Media Transport in the Session Description Protocol (SDP)", draft-ietf-mmusic-sctp-sdp-04 (work in progress), June 2013.

[I-D.ietf-mmusic-sdp-bundle-negotiation]
Holmberg, C., Alvestrand, H., and C. Jennings, "Multiplexing Negotiation Using Session Description Protocol (SDP) Port Numbers", draft-ietf-mmusic-sdp-bundle-negotiation-04 (work in progress), June 2013.

[I-D.ietf-mmusic-sdp-mux-attributes]
Nandakumar, S., "A Framework for SDP Attributes when Multiplexing", draft-ietf-mmusic-sdp-mux-attributes-01 (work in progress), February 2014.

[I-D.ietf-rtcweb-audio]
Valin, J. and C. Bran, "WebRTC Audio Codec and Processing Requirements", draft-ietf-rtcweb-audio-02 (work in progress), August 2013.

[I-D.ietf-rtcweb-data-protocol]
Jesup, R., Loreto, S., and M. Tuexen, "WebRTC Data Channel Protocol", draft-ietf-rtcweb-data-protocol-04 (work in progress), February 2013.

[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-09 (work in progress), September 2013.

[I-D.ietf-rtcweb-security]
Rescorla, E., "Security Considerations for WebRTC", draft-ietf-rtcweb-security-06 (work in progress), January 2014.

[I-D.ietf-rtcweb-security-arch]
Rescorla, E., "WebRTC Security Architecture", draft-ietf-rtcweb-security-arch-09 (work in progress), February 2014.

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

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

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

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

[RFC4145] Yon, D. and G. Camarillo, "TCP-Based Media Transport in the Session Description Protocol (SDP)", RFC 4145, September 2005.

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

[RFC4572] Lennox, J., "Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP)", RFC 4572, July 2006.

[RFC4585] Ott, J., Wenger, S., Sato, N., Burmeister, C., and J. Rey, "Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF)", RFC 4585, July 2006.

[RFC5124] Ott, J. and E. Carrara, "Extended Secure RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/SAVPF)", RFC 5124, February 2008.

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

[RFC5285] Singer, D. and H. Desineni, "A General Mechanism for RTP Header Extensions", RFC 5285, July 2008.

[RFC5761] Perkins, C. and M. Westerlund, "Multiplexing RTP Data and Control Packets on a Single Port", RFC 5761, April 2010.

[RFC5888] Camarillo, G. and H. Schulzrinne, "The Session Description Protocol (SDP) Grouping Framework", RFC 5888, June 2010.

[RFC6904] Lennox, J., "Encryption of Header Extensions in the Secure

Real-time Transport Protocol (SRTP)", RFC 6904, April 2013.

[RFC7022] Begen, A., Perkins, C., Wing, D., and E. Rescorla, "Guidelines for Choosing RTP Control Protocol (RTCP) Canonical Names (CNAMEs)", RFC 7022, September 2013.

10.2. Informative References

[I-D.ietf-mmusic-trickle-ice] Ivov, E., Rescorla, E., and J. Uberti, "Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol", draft-ietf-mmusic-trickle-ice-00 (work in progress), March 2013.

[I-D.nandakumar-rtcweb-sdp] Nandakumar, S. and C. Jennings, "SDP for the WebRTC", draft-nandakumar-rtcweb-sdp-02 (work in progress), July 2013.

[RFC3389] Zopf, R., "Real-time Transport Protocol (RTP) Payload for Comfort Noise (CN)", RFC 3389, September 2002.

[RFC3556] Casner, S., "Session Description Protocol (SDP) Bandwidth Modifiers for RTP Control Protocol (RTCP) Bandwidth", RFC 3556, July 2003.

[RFC3960] Camarillo, G. and H. Schulzrinne, "Early Media and Ringing Tone Generation in the Session Initiation Protocol (SIP)", RFC 3960, December 2004.

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

[RFC4588] Rey, J., Leon, D., Miyazaki, A., Varsa, V., and R. Hakenberg, "RTP Retransmission Payload Format", RFC 4588, July 2006.

[RFC5506] Johansson, I. and M. Westerlund, "Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences", RFC 5506, April 2009.

[RFC5576] Lennox, J., Ott, J., and T. Schierl, "Source-Specific Media Attributes in the Session Description Protocol (SDP)", RFC 5576, June 2009.

[RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, May 2010.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, May 2010.

[W3C.WD-webrtc-20140617] Bergkvist, A., Burnett, D., Narayanan, A., and C. Jennings, "WebRTC 1.0: Real-time Communication Between Browsers", World Wide Web Consortium WD WD-webrtc-20140617, June 2014, <http://www.w3.org/TR/2011/WD-webrtc-20140617>.

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

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

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

   // ICE Completes (at Offerer)
   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: pc.addStream(localStream, null);
   AnswererJS->AnswererUA: answer = pc.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:   pc.setRemoteDescription("answer", answer);
   OffererUA->OffererJS:   onaddstream(remoteStream);

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

   // ICE Completes (at Offerer)
   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: pc.addStream(localStream, null);
   AnswererJS->AnswererUA: answer = pc.createAnswer(offer, null);
   AnswererJS:             sip = createResponse(200, answer);
   AnswererJS->AnswererUA: pc.setLocalDescription("answer", answer);
   AnswererJS->OffererJS:  200 OK w/ SDP

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

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

   // ICE Completes (at Offerer)
   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->AnswererUA:  Media

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

A.2. Example Session Descriptions

A.2.1. createOffer

This SDP shows a typical initial offer, created by createOffer for a PeerConnection with a single audio MediaStreamTrack, a single video MediaStreamTrack, and a single data channel. Host candidates have also already been gathered. Note some lines have been broken into two lines for formatting reasons.

```
   v=0
   o=- 4962303333179871722 1 IN IP4 0.0.0.0
   s=-
   t=0 0
   a=msid-semantic:WMS
   a=group:BUNDLE audio video data
   m=audio 56500 UDP/TLS/RTP/SAVPF 111 0 8 126
   c=IN IP4 192.0.2.1
   a=rtcp:56501 IN IP4 192.0.2.1
   a=candidate:3348148302 1 udp 2113937151 192.0.2.1 56500
               typ host generation 0
   a=candidate:3348148302 2 udp 2113937151 192.0.2.1 56501
               typ host generation 0
   a=ice-ufrag:ETEn1v9DoTMB9J4r
   a=ice-pwd:OtSK0WpNtpUjkY4+86js7ZQl
   a=ice-options:trickle
```

```
   a=mid:audio
   a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
   a=sendrecv
   a=rtcp-mux
   a=rtcp-rsize
   a=fingerprint:sha-256
                 19:E2:1C:3B:4B:9F:81:E6:B8:5C:F4:A5:A8:D8:73:04
                :BB:05:2F:70:9F:04:A9:0E:05:E9:26:33:E8:70:88:A2
   a=setup:actpass
   a=rtpmap:111 opus/48000/2
   a=fmtp:111 minptime=10
   a=rtpmap:0 PCMU/8000
   a=rtpmap:8 PCMA/8000
   a=rtpmap:126 telephone-event/8000
   a=maxptime:60
   a=ssrc:1732846380 cname:EocUG1f0fcg/yvY7
   a=msid:47017fee-b6c1-4162-929c-a25110252400
          f83006c5-a0ff-4e0a-9ed9-d3e6747be7d9
   m=video 56502 UDP/TLS/RTP/SAVPF 100 115 116 117
   c=IN IP4 192.0.2.1
   a=rtcp:56503 IN IP4 192.0.2.1
   a=candidate:3348148302 1 udp 2113937151 192.0.2.1 56502
               typ host generation 0
   a=candidate:3348148302 2 udp 2113937151 192.0.2.1 56503
               typ host generation 0
   a=ice-ufrag:BGKkWnG5GmiUpdIV
   a=ice-pwd:mqyWsAjvtKwTGnvhPztQ9mIf
   a=ice-options:trickle
   a=mid:video
   a=extmap:2 urn:ietf:params:rtp-hdrext:toffset
   a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
   a=sendrecv
   a=rtcp-mux
   a=rtcp-rsize
   a=fingerprint:sha-256
                 19:E2:1C:3B:4B:9F:81:E6:B8:5C:F4:A5:A8:D8:73:04
                :BB:05:2F:70:9F:04:A9:0E:05:E9:26:33:E8:70:88:A2
   a=setup:actpass
   a=rtpmap:100 VP8/90000
   a=rtcp-fb:100 ccm fir
   a=rtcp-fb:100 nack
   a=rtcp-fb:100 goog-remb
   a=rtpmap:115 rtx/90000
   a=fmtp:115 apt=100
   a=rtpmap:116 red/90000
   a=rtpmap:117 ulpfec/90000
   a=ssrc:1366781083 cname:EocUG1f0fcg/yvY7
   a=ssrc:1366781084 cname:EocUG1f0fcg/yvY7
```

```
   a=ssrc:1366781085 cname:EocUG1f0fcg/yvY7
   a=ssrc-group:FID 1366781083 1366781084
   a=ssrc-group:FEC 1366781083 1366781085
   a=msid:61317484-2ed4-49d7-9eb7-1414322a7aae
          f30bdb4a-5db8-49b5-bcdc-e0c9a23172e0
   m=application 56504 DTLS/SCTP 5000
   c=IN IP4 192.0.2.1
   a=candidate:3348148302 1 udp 2113937151 192.0.2.1 56504
               typ host generation 0
   a=ice-ufrag:VD5v2BnbZm3mgP3d
   a=ice-pwd:+Jlkuox+VVIUDqxcfIDuTZMH
   a=ice-options:trickle
   a=mid:data
   a=fingerprint:sha-256 19:E2:1C:3B:4B:9F:81:E6:B8:5C:F4:A5:A8:D8:73:04
                         :BB:05:2F:70:9F:04:A9:0E:05:E9:26:33:E8:70:88:A2
   a=setup:actpass
   a=sctpmap:5000 webrtc-datachannel 16
```

A.2.2. createAnswer

This SDP shows a typical initial answer to the above offer, created by createAnswer for a PeerConnection with a single audio MediaStreamTrack, a single video MediaStreamTrack, and a single data channel. Host candidates have also already been gathered. Note some lines have been broken into two lines for formatting reasons.

```
   v=0
   o=- 6729291447651054566 1 IN IP4 0.0.0.0
   s=-
   t=0 0
   a=msid-semantic:WMS
   a=group:BUNDLE audio video data
   m=audio 20000 UDP/TLS/RTP/SAVPF 111 0 8 126
   c=IN IP4 192.0.2.2
   a=candidate:2299743422 1 udp 2113937151 192.0.2.2 20000
               typ host generation 0
   a=ice-ufrag:6sFvz2gdLkEwjZEr
   a=ice-pwd:cOTZKZNVlO9RSGsEGM63JXT2
   a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F:46:1B:35
                         :DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1:2C:19:08
   a=setup:active
   a=mid:audio
   a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
   a=sendrecv
   a=rtcp-mux
   a=rtpmap:111 opus/48000/2
   a=fmtp:111 minptime=10
   a=rtpmap:0 PCMU/8000
```

```
   a=rtpmap:8 PCMA/8000
   a=rtpmap:126 telephone-event/8000
   a=maxptime:60
   a=ssrc:3429951804 cname:Q/NWs1ao1HmN4Xa5
   a=msid:PI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1
          PI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1a0
   m=video 20000 UDP/TLS/RTP/SAVPF 100 115 116 117
   c=IN IP4 192.0.2.2
   a=candidate:2299743422 1 udp 2113937151 192.0.2.2 20000
               typ host generation 0
   a=ice-ufrag:6sFvz2gdLkEwjZEr
   a=ice-pwd:cOTZKZNVlO9RSGsEGM63JXT2
   a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F:46:1B:35
                        :DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1:2C:19:08
   a=setup:active
   a=mid:video
   a=extmap:2 urn:ietf:params:rtp-hdrext:toffset
   a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
   a=sendrecv
   a=rtcp-mux
   a=rtpmap:100 VP8/90000
   a=rtcp-fb:100 ccm fir
   a=rtcp-fb:100 nack
   a=rtcp-fb:100 goog-remb
   a=rtpmap:115 rtx/90000
   a=fmtp:115 apt=100
   a=rtpmap:116 red/90000
   a=rtpmap:117 ulpfec/90000
   a=ssrc:3229706345 cname:Q/NWs1ao1HmN4Xa5
   a=ssrc:3229706346 cname:Q/NWs1ao1HmN4Xa5
   a=ssrc:3229706347 cname:Q/NWs1ao1HmN4Xa5
   a=ssrc-group:FID 3229706345 3229706346
   a=ssrc-group:FEC 3229706345 3229706347
   a=msid:PI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1
          PI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1v0
   m=application 20000 DTLS/SCTP 5000
   c=IN IP4 192.0.2.2
   a=candidate:2299743422 1 udp 2113937151 192.0.2.2 20000
               typ host generation 0
   a=ice-ufrag:6sFvz2gdLkEwjZEr
   a=ice-pwd:cOTZKZNVlO9RSGsEGM63JXT2
   a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F:46:1B:35
                        :DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1:2C:19:08
   a=setup:active
   a=mid:data
   a=sctpmap:5000 webrtc-datachannel 16
```

A.2.3. Call Flows

Example SDP for WebRTC call flows can be found in [I-D.nandakumar-rtcweb-sdp]. [TODO: should these call flows be merged into this section?]

Appendix B. Change log

Changes in draft-06:

- o Reworked handling of m= line recycling.
- o Added handling of BUNDLE and bundle-only.
- o Clarified handling of rollback.
- o Added text describing the ICE Candidate Pool and its behavior.
- o Allowed OfferToReceiveX to create multiple recvonly m= sections.

Changes in draft-05:

- o Fixed several issues identified in the createOffer/Answer sections during document review.
- o Updated references.

Changes in draft-04:

- o Filled in sections on createOffer and createAnswer.
- o Added SDP examples.
- o Fixed references.

Changes in draft-03:

- o Added text describing relationship to W3C specification

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 specification
- 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:

- 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

Eric Rescorla (editor)
Mozilla
331 Evelyn Ave
Mountain View, CA 94041
USA

Email: ekr@rtfm.com