Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 6, 2012

J. Uberti
Google
C. Jennings
Cisco Systems, Inc.
June 4, 2012

Javascript Session Establishment Protocol
draft-ietf-rtcweb-jsep-01

Abstract

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

This document is an input document for discussion. It should be discussed in the RTCWEB WG list, rtcweb@ietf.org.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 26, 2012.

Table of Contents

1. Introduction

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

The original spec for WebRTC attempted to implement this protocol-agnostic signaling by providing a mechanism to exchange session descriptions in the form of SDP blobs. Upon starting a session, the browser would generate a SDP blob, which would be passed to the application for transport over its preferred signaling protocol. On the remote side, this blob would be passed into the browser from the application, and the browser would then generate a blob of its own in response. Upon transmission back to the initiator, this blob would be plugged into their browser, and the handshake would be complete.

Experimentation with this mechanism turned up several shortcomings, which generally stemmed from there being insufficient context at the browser to fully determine the meaning of a SDP blob. For example, determining whether a blob is an offer or an answer, or differentiating a new offer from a retransmit.

The ROAP proposal, specified in [I-D.draft-jennings-rtcweb-signaling-01], attempted to resolve these issues by providing additional structure in the messaging - in essence, to create a generic signaling protocol that specifies how the browser signaling state machine should operate. However, even though the protocol is abstracted, the state machine forces a least-common-denominator approach on the signaling interactions. For example, in Jingle, the call initiator can provide additional ICE candidates even after the initial offer has been sent, which allows the offer to be sent immediately for quicker call startup. However, in the browser state machine, there is no notion of sending an updated offer before the initial offer has been responded to, rendering this functionality impossible.

While specific concerns like this could be addressed by modifying the generic protocol, others would likely be discovered later. The main reason this mechanism is inflexible is because it embeds a signaling state machine within the browser. Since the browser generates the session descriptions on its own, and fully controls the possible

states and advancement of the signaling state machine, modification of the session descriptions or use of alternate state machines becomes difficult or impossible.

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

If instead the state machine is run at the browser end, and is instantiated within, for example, the PeerConnection object, that state machine will be reinitialized when the page is reloaded and the JavaScript re-executed. This actually complicates the design of any interoperability service, as all cases where an offer or answer has already been generated but is now "forgotten" must now be handled by trying to move the client state machine forward to the same state it had been in previously in order to match what has already been delivered to and/or answered by the far side, or handled by ensuring that aborts are cleanly handled from every state and the negotiation rapidly restarted.

1.1. Terminology

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

2. JSEP Approach

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

JSEP's handling of session descriptions is simple and straightforward. Whenever an offer/answer exchange is needed, the initiating side creates an offer by calling a createOffer() API. The application can do massaging of that offer, if it wants to, and then uses it to set up its local config via a 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 a setRemoteDescription() API.

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

Regarding ICE, JSEP decouples the ICE state machine from the overall signaling state machine, as the ICE state machine must remain in the browser, since only the browser has the necessary knowledge of candidates and other transport info. Performing this separation it provides additional flexibility; in protocols that decouple session descriptions from transport, such as Jingle, the transport information can be sent separately; in protocols that don't, such as SIP, the information can be easily aggregated and recombined. Sending transport information separately can allow for faster ICE and DTLS startup, since the necessary roundtrips can occur while waiting for the remote side to accept the session.

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

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

3. Other Approaches Considered

Another approach that was considered for JSEP was to move the mechanism for generating offers and answers out of the browser as well. Instead of providing createOffer/createAnswer methods within the browser, this approach would instead expose a getCapabilities API which would provide the application with the information it needed in order to generate its own session descriptions. This increases the amount of work that the application needs to do; it needs to know how to generate session descriptions from capabilities, and especially

how to generate the correct answer from an arbitrary offer and the supported capabilities. While this could certainly be addressed by using a library like the one mentioned above, it basically forces the use of said library even for a simple example. Exposing createOffer/createAnswer avoids that problem, but still allows applications to generate their own offers/answers if they choose, using the description generated by createOffer as an indication of the browser's capabilities.

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

4. Semantics and Syntax

4.1. Signaling Model

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

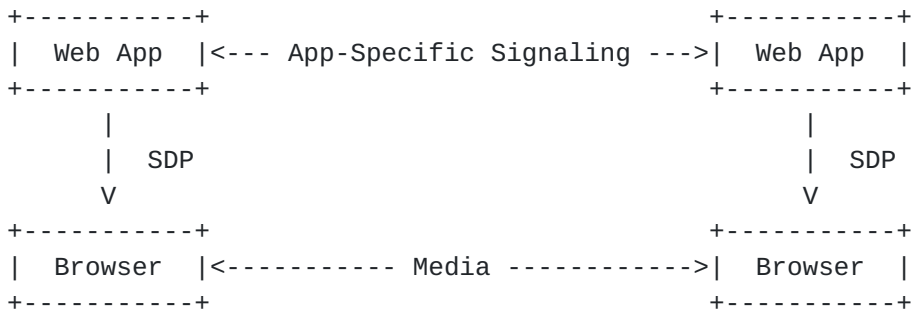

Figure 1: JSEP Signaling Model

4.2. Session Descriptions and State Machine

In order to establish the media plane, the user agent needs specific parameters to indicate what to transmit to the remote side, as well

as how to handle the media that is received. These parameters are determined by the exchange of session descriptions in offers and answers, and there are certain details to this process that must be handled in the JSEP APIs.

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

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

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

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

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

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

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

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

It also allows for an answer to be treated as provisional by the application. Provisional answers provide a way for an answerer to communicate session parameters back to the offerer, in order for 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; as a result, multiple dissimilar provisional answers can be received and applied during call setup.

As in [RFC3264], an offerer can send an offer, and update it as long as it has not been answered. 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:

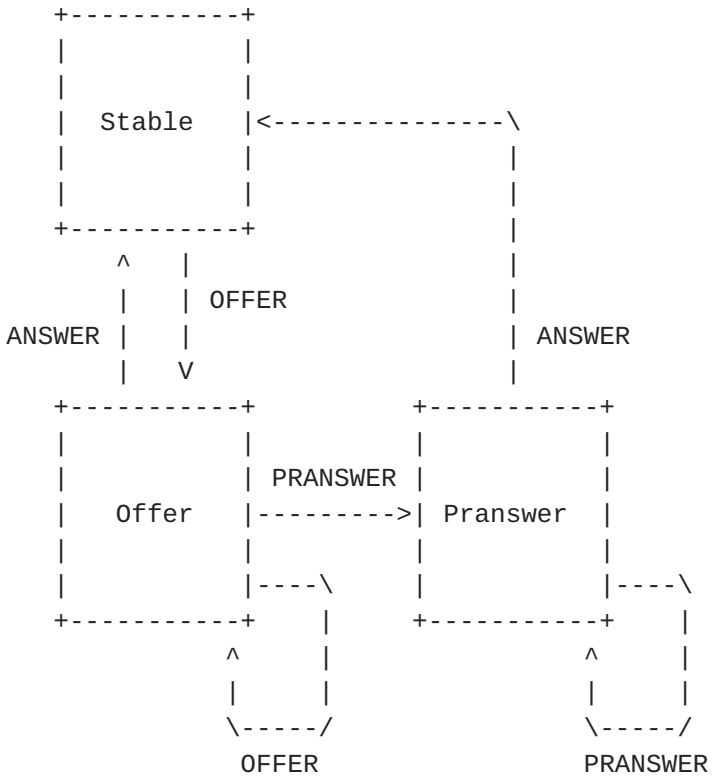

Figure 2: JSEP State Machine

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

4.3. Session Description Format

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

However, to simplify Javascript processing, and provide for future flexibility, the SDP syntax is encapsulated within a SessionDescription object, which can be constructed from SDP, and be serialized out to SDP. If we were able to agree on a JSON format for session descriptions, we could easily enable this object to generate/expect JSON.

Other methods may be added to SessionDescription in the future to simplify handling of SessionDescriptions from Javascript.

4.4. Separation of Signaling and ICE State Machines

JSEP does away with the SDP Agent within the browser, and this functionality is now controlled directly by the application, which uses the setLocalDescription and setRemoteDescription APIs to tell the browser what SDP has been negotiated. The ICE Agent remains in the browser, as it still needs to drive the process of gathering candidates, connectivity checks, and related ICE functionality.

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

4.5. 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. This 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 allocate all possible candidates, resulting in faster call startup in many cases.

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 onicecandidate callback with a new candidate; applications that do

not support this feature can simply wait for the final onicecandidate callback that indicates gathering is complete, and create and send their offer, with all the candidates, at this time.

Upon receipt of trickled candidates, the receiving application can supply them to its ICE Agent by calling an addIceCandidate method. This triggers the ICE Agent to start using this remote candidate for connectivity checks. Applications that do not make use of candidate tricking can ignore addIceCandidate entirely, and use the onicecandidate callback solely to indicate when candidate gathering is complete.

4.6. ICE Candidate Format

As with session descriptions, we choose to provide an IceCandidate object that provides some abstraction, but can be easily converted to/from SDP a=candidate lines.

The IceCandidate object has fields to indicate which m= line it should be associated with, and a method to convert to a SDP representation, ex:

```
a=candidate:1 1 UDP 1694498815 66.77.88.99 10000 typ host
```

Currently, a=candidate lines are the only SDP information that is contained within IceCandidate, as they represent the only information needed that is not present in the initial offer (i.e. for trickle candidates).

4.7. Interactions With Forking

4.7.1. Serial Forking

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

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

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

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

4.7.2. Parallel Forking

Parallel forking involves a call being dispatched to multiple remote callees, where each callee can accept the call, and multiple simultaneous active sessions can be established as a result. If multiple callees send media, this media is mixed and played out at the caller side.

JSEP can handle parallel forking by "cloning" the session when needed to create multiple parallel sessions. When the first answer is received, the caller can clone the existing session, and then apply the answer as a final answer to the original session. Upon receiving the next answer, the cloned session is cloned again, and the received answer is applied as a final answer to the first clone. This process repeats until the caller decides to end the setup flow, and closes the final cloned session.

Cloned sessions inherit the local session description and candidates from their parent, and an empty remote description; only sessions that have not yet applied an answer can be cloned. Each cloned session may discover new peer-reflexive candidates; these candidates will be supplied via the onicecandidate callback to that specific session. Since the clone uses the same local description as its parent, creating a clone will fail if it is not possible to reserve the same resources for the clone as have already been reserved by the parent.

As a result of this cloning, the application will end up with N parallel 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.

4.8. Session Rehydration

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

"rehydration".

With rehydration, the current local session description is persisted somewhere outside of the page, perhaps on the application server, or in browser local storage. The page is then reloaded, and a new session object is created in Javascript. The saved local session is now retrieved, but the previous ICE candidates will no longer be valid in this case, so we will need to perform an ICE restart; to do so, we simply generate a new ICE ufrag/pwd combo for the local description.

The modified local description is then installed via setLocalDescription, and sent off as an offer to the remote side, who will reply with an answer that can be supplied to setRemoteDescription. ICE processing proceeds as usual, and as soon as connectivity is established, the session will be back up and running again.

5. Interface

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

5.1. Methods

5.1.1. createOffer

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

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

description.

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

Calling this method does not change state; its use is not required.

5.1.2. createAnswer

The createAnswer method generates a blob of SDP that contains a RFC 3264 SDP answer with the supported configuration for the session that is compatible with the parameters supplied in |offer|. Like createOffer, the returned blob contains descriptions of the local MediaStreams attached to this PeerConnection, the codec/RTP/RTCP options negotiated for this session, and any candidates that have been gathered by the ICE Agent. A constraints parameter may be supplied to provide additional control over the generated answer.

As an answer, the generated SDP will contain a specific configuration that specifies how the media plane should be established. For each SDP line, the generation of the SDP must follow the appropriate process for generating an answer.

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 does not change state; its use is not required.

5.1.3. SessionDescriptionType

The strings "offer", "pranswer", and "answer" serve as type arguments to setLocalDescription and setRemoteDescription. They provide information as to how the description parameter should be parsed, and how the media state should be changed.

"offer" indicates that a description should be parsed as an offer; said description may include many possible media configurations. A description used as an "offer" may be applied anytime the

PeerConnection is in a stable state, or as an update to a previously sent but unanswered "offer".

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

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

The application can use some discretion on whether an answer should be applied as provisional or final. For example, in a serial forking scenario, an application may receive multiple "final" answers, one from each remote endpoint. The application could accept the initial answers as provisional answers, and only apply an answer as final when it receives one that meets its criteria (e.g. a live user instead of voicemail).

5.1.4. setLocalDescription

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

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

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

5.1.5. setRemoteDescription

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

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

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

5.1.6. localDescription

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

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

5.1.7. remoteDescription

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

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

5.1.8. updateIce

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

Regardless of the configuration, the gathering process collects all available candidates, but excluded candidates will not be surfaced in onicecallback or used for connectivity checks.

This call may result in a change to the state of the ICE Agent, and

may result in a change to media state if it results in connectivity being established.

5.1.9. addIceCandidate

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

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

5.2. Configurable SDP Parameters

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

- remove or reorder codecs (m=)
- change codec attributes (a=fmtp; ptime)
- enable/disable BUNDLE (a=group)
- enable/disable RTCP mux (a=rtcp-mux)
- remove or reorder SRTP crypto-suites (a=crypto)
- change SRTP parameters or keys (a=crypto)
- change send resolution or framerate (TBD)
- change desired recv resolution or framerate (TBD)
- change total bandwidth (b=)
- remove desired AVPF mechanisms (a=rtcp-fb)
- remove RTP header extensions (a=rtphdr-ext)
- add/change SSRC grouping (e.g. FID, RTX, etc) (a=ssrc-group)
- add SSRC attributes (a=ssrc)
- change ICE ufrag/password (a=ice-ufrag/pwd)
- change media send/recv state (a=sendonly/recvonly/inactive)

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

6. Media Setup Overview

The example here shows a typical call setup using the JSEP model, indicating the functions that are called and the state changes that occur. We assume the following architecture in this example, where UA is synonymous with "browser", and JS is synonymous with "web application":

OffererUA <-> OffererJS <-> WebServer <-> AnswererJS <-> AnswererUA

6.1. Initiating the Session

The initiator creates a PeerConnection, hooks up to its ICE callback, and adds the desired MediaStreams (presumably obtained via getUserMedia). The ICE gathering process begins to gather candidates for a default number of streams, as the exact number will not be known until the local description is applied. The PeerConnection is in the NEW state.

```
OffererJS->OffererUA: var pc = new PeerConnection(config, null);
OffererJS->OffererUA: pc.onicecandidate = onIceCandidate;
OffererJS->OffererUA: pc.addStream(stream);
```

6.1.1. Generating An Offer

The initiator then creates a session description to offer to the callee. This description includes the codecs and other necessary session parameters, as well as information about each of the streams that has been added (e.g. SSRC, CNAME, etc.) The created description includes all parameters that the offerer's UA supports; if the initiator wants to influence the created offer, they can pass in a MediaConstraints object to createOffer that allows for customization (e.g. if the initiator wants to receive but not send video). The initiator can also directly manipulate the created session description as well, perhaps if it wants to change the priority of the offered codecs.

```
OffererJS->OffererUA: var offer = pc.createOffer(null);
```

6.1.2. Applying the Offer

The initiator then instructs the PeerConnection to use this offer as the local description for this session, i.e. what codecs it will use for received media, what SRTP keys it will use for sending media (if using SDES), etc. In order that the UA handle the description properly, the initiator marks it as an offer when calling setLocalDescription; this indicates to the UA that multiple capabilities have been offered, but this set may be pared back later, when the answer arrives.

Since the local user agent must be prepared to receive media upon applying the offer, this operation will cause local decoder resources to be allocated, based on the codecs indicated in the offer.

```
OffererJS->OffererUA: pc.setLocalDescription("offer", offer);
```

6.1.3. Handling ICE Callbacks

The initiator starts to receive callbacks on its onicecandidate handler. Candidates are provided to the IceCallback as they are allocated; when the last allocation completes or times out, this callback will be invoked with a null argument.

OffererUA->OffererJS: onIceCandidate(candidate);

6.1.4. Serializing the Offer and Candidates

At this point, the offerer is ready to send its offer to the callee using its preferred signaling protocol. Depending on the protocol, it can either send the initial session description first, and then "trickle" the ICE candidates as they are given to the application, or it can wait for all the ICE candidates to be collected, and then send the offer and list of candidates all at once.

6.2. Receiving the Session

Through the chosen signaling protocol, the recipient is notified of an incoming session request. It creates a PeerConnection, and sets up its own ICE callback. The ICE gathering process begins to gather candidates for a default number of streams.

AnswererJS->AnswererUA: var pc = new PeerConnection(config, null);
AnswererJS->AnswererUA: pc.onicecandidate = onIceCandidate;

6.2.1. Receiving the Offer

The recipient converts the received offer from its signaling protocol into SDP format, and supplies it to its PeerConnection, again marking it as an offer. As a remote description, the offer indicates what codecs the remote side wants to use for receiving, as well as what SRTP keys it will use for sending. The setting of the remote description causes callbacks to be issued, informing the application of what kinds of streams are present in the offer.

This step will also cause encoder resources to be allocated, based on the codecs specified in |offer|.

AnswererJS->AnswererUA: pc.setRemoteDescription("offer", offer);
AnswererUA->AnswererJS: onAddStream(stream);

6.2.2. Handling ICE Messages

If ICE candidates from the remote site were included in the offer, the ICE Agent will automatically start trying to use them. Otherwise, if ICE candidates are sent separately, they are passed into the PeerConnection when they arrive.

```
AnswererJS->AnswererUA: pc.addIceCandidate(candidate);
```

6.2.3. Generating the Answer

Once the recipient has decided to accept the session, it generates an answer session description. This process performs the appropriate intersection of codecs and other parameters to generate the correct answer. As with the offer, MediaConstraints can be provided to influence the answer that is generated, and/or the application can post-process the answer manually.

```
AnswererJS->AnswererUA: pc.createAnswer(offer, null);
```

6.2.4. Applying the Answer

The recipient then instructs the PeerConnection to use the answer as its local description for this session, i.e. what codecs it will use to receive media, etc. It also marks the description as an answer, which tells the UA that these parameters are final. This causes the PeerConnection to move to the ACTIVE state, and transmission of media by the answerer to start (assuming both sides have indicated this in their descriptions).

```
AnswererJS->AnswererUA: pc.setLocalDescription("answer", answer);
AnswererUA->OffererUA:  <media>
```

6.2.5. Serializing the Answer

As with the offer, the answer (with or without candidates) is now converted to the desired signaling format and sent to the initiator.

6.3. Completing the Session

6.3.1. Receiving the Answer

The initiator converts the answer from the signaling protocol and applies it as the remote description, marking it as an answer. This causes the PeerConnection to move to the ACTIVE state, and transmission of media by the offerer to start (assuming both sides have indicated this in their descriptions).

```
OffererJS->OffererUA:  pc.setRemoteDescription("answer", answer);
OffererUA->AnswererUA: <media>
```

6.4. Updates to the Session

Updates to the session are handled with a new offer/answer exchange. However, since media will already be flowing at this point, the new

offerer needs to support both its old session description as well as the new one it has offered, until the change is accepted by the remote side.

Note also that in an update scenario, the roles may be reversed, i.e. the update offerer can be different than the original offerer.

7. Security Considerations

TODO

8. IANA Considerations

This document requires no actions from IANA.

9. Acknowledgements

Harald Alvestrand, Dan Burnett, Neil Stratford, Eric Rescorla, Anant Narayanan, and Adam Bergkvist all provided valuable feedback on this proposal. Matthew Kaufman provided the observation that keeping state out of the browser allows a call to continue even if the page is reloaded. Richard Ejzak provided the specifics on session cloning.

10. References

10.1. Normative References

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

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

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

10.2. Informative References

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

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

[webrtc-api] Bergkvist, Burnett, Jennings, Narayanan, "WebRTC 1.0:

Real-time Communication Between Browsers", May 2011.

Available at http://dev.w3.org/2012/webrtc/editor/webrtc.html

Appendix A. JSEP Implementation Examples

A.1. Example API

The interface below shows a basic Javascript API that could be used to expose the functionality discussed in this document. This API is used for the examples in the following parts of this Appendix.

```
// actions, for setLocalDescription/setRemoteDescription
enum SessionDescriptionType { "offer", "pranswer", "answer" }

// constraints that can be supplied to the ctor or createXXXX
enum MediaConstraints {
    "offerConfig",    // controls the kind of offer created;
                      //   "default"    (normal offer)
                      //   "caps"       (all capabilities)
                      //   "new"        (brand new description)
                      //   "iceRestart" (new ICE creds)

    "iceTransports",  // controls ICE candidates; can be
                      //   "none"  (no candidates)
                      //   "relay" (only relay candidates)
                      //   "all"   (all available candidates)
}

[Constructor (int index, DOMString id, in DOMString candidateLine)]
interface IceCandidate {
    // the m= line index for this candidate
    readonly attribute int mLineIndex
    // the mid for the m= line for this candidate
    readonly attribute DOMString mLineId;
    // creates a SDP-ized form of this candidate
    stringifier DOMString ();
};

[Constructor (DOMString sdp)]
interface SessionDescription {
    // adds the specified candidate to the description
    void addCandidate(IceCandidate candidate);
    // serializes the description to SDP
    stringifier DOMString ();
};

[Constructor (DOMString configuration,
```

```
                optional MediaConstraints constraints)]
   interface PeerConnection {
       // creates a blob of SDP to be provided as an offer.
       SessionDescription createOffer (
           SessionDescriptionCallback successCb,
           optional ErrorCallback errorCb,
           optional MediaContraints constraints);
       // creates a blob of SDP to be provided as an answer.
       SessionDescription createAnswer (
           SessionDescription offer,
           SessionDescriptionCallback successCb,
           optional ErrorCallback errorCb,
           optional MediaContraints constraints);

       // sets the local session description
       void setLocalDescription (
           SessionDescriptionType action,
           SessionDescription desc);
       // sets the remote session description
       void setRemoteDescription (
           SessionDescriptionType action,
           SessionDescription desc)
       // returns the current local session description
       readonly attribute SessionDescription localDescription;
       // returns the current remote session description
       readonly attribute SessionDescription remoteDescription;

       // updates the constraints for ICE processing
       void updateIce (
         optional DOMString configuration,
         optional MediaConstraints constraints);
       // starts using a received remote ICE candidate
       void addIceCandidate (
         IceCandidate candidate);
       // notifies the application of a new local ICE candidate
       attribute Function?          onicecandidate;
   };
```

A.2. 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.2.1. Call using ROAP

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

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

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

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

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

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

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

A.2.2 Call using XMPP

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

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

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

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

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

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

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

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

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

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

A.2.3. 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.2.4. 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.2.5. Call using SIP

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

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

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

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

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

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

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

A.2.6. Handling early media (e.g. 1-800-FEDEX), using SIP

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

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

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

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

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

A.3. Full Example Application

The following example demonstrates a simple video calling application, using both trickle candidates and provisional answers to speed up call setup.

```
// Usage:
// Caller calls start(true)
// Callee calls start(false) to prepare the call/start connecting,
// and then accept() to start transmitting.

var signalingChannel = createSignalingChannel();
var pc = null;
var localStream = null;
signalingChannel.onmessage = handleMessage;
```

```
   // Set up the call, get access to local media,
   // and establish connectivity.
   function start(isCaller) {
     // Create a PeerConnection and hook up the IceCallback.
     pc = new webkitPeerConnection(null, null);
     pc.onicecandidate = function(evt) {
       sendMessage("candidate", evt.candidate);
     };

     // Get the local stream and show it in the local video element;
     // if we're the caller, ship off an offer once we get the stream.
     navigator.webkitGetUserMedia(
         {"audio": true, "video": true}, function (stream) {
       selfView.src = webkitURL.createObjectURL(stream);
       localStream = stream;
       if (isCaller) {
         pc.addStream(stream);
         pc.createOffer(function(sdp) {
             setLocalAndSendMessage("offer", sdp);
         });
     });

     // When the remote stream arrives, show it in the remote
     // video element.
     pc.onaddstream = function(evt) {
       remoteView.src = webkitURL.createObjectURL(evt.stream);
     };
   }

   // The callee has accepted the call, attach their media
   // and send a final answer.
   function accept() {
     // The addStream could also be done for the pranswer,
     // although that would delay the pranswer
     // (due to the need for user consent)
     pc.addStream(localStream);  // assumes we have the stream already
     pc.createAnswer(msg.sdp, function(sdp) {
       setLocalAndSendMessage("answer", sdp);
     });
   }

   // -- internal methods --

   // Apply SDP locally and send it to the remote side.
   function setLocalAndSendMessage(type, sdp) {
     pc.setLocalDescription(type, sdp);
     sendMessage(type, sdp);
   }
```

```
   // Send a signaling message to the remote side.
   function sendMessage(type, obj) {
     signalingChannel.send(
          JSON.stringify({ "type": type, "sdp": obj }));
   }

   // Handle incoming signaling messages.
   function handleMessage(str) {
     var msg = JSON.parse(str);
     switch (msg.type) {
       case "offer":
         // create the PeerConnection
         start(false);
         // feed the received offer into the PeerConnection
         pc.setRemoteDescription(msg.type, msg.sdp);
         // create provisional answer to allow ICE/DTLS to start
         pc.createAnswer(msg.sdp, function(sdp) {
           setDirection(sdp, "recvonly");
           setLocalAndSendMessage("pranswer", sdp);
         });
         break;
       case "pranswer":
       case "answer":
         pc.setRemoteDescription(msg.type, msg.sdp);
         break;
       case "candidate":
         pc.addIceCandidate(msg.sdp);
         break;
     }
   }
```

Appendix B. Change log

```
   01: Added diagrams for architecture and state machine.
       Added sections on forking and rehydration.
       Clarified meaning of "pranswer" and "answer".
       Reworked how ICE restarts and media directions are controlled.
       Added list of parameters that can be changed in a description.
       Updated suggested API and examples to match latest thinking.
       Suggested API and examples have been moved to an appendix.
   00: Migrated from draft-uberti-rtcweb-jsep-02.
```

Authors' Addresses

Justin Uberti
Google
5 Cambridge Center
Cambridge, MA 02142

Email: justin@uberti.name

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

Email: fluffy@cisco.com