RTCWeb Offer/Answer Protocol (ROAP)
draft-jennings-rtcweb-signaling-00

Abstract

This document describes an protocol used to negotiate media between browsers or other compatible devices. This protocol provides the state machinery needed to implement the offer/answer model (RFC 3264), and defines the semantics and necessary attributes of messages that must be exchanged. The protocol uses an abstract transport in that it does not actually define how these messages are exchanged. Rather, such exchanges are handled through web-based transports like HTTP or WebSockets. The protocol focuses solely on media negotiation and does not handle call control, call processing, or other functions.

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 April 16, 2012.

Copyright Notice

Table of Contents

1. Introduction

This specification defines a protocol that allows an RTCWeb browser to exchange information to control the set up of media to another browser or device. The scope of this protocol is limited to functionality required for the setup and negotiation of media and the associated transports, referred to as media control. The protocol defines the minimum set of messages and state machinery necessary to implement the offer/answer model as defined in **[RFC3264]**. The offer answer model specifies rules for the bilateral exchange of Session Description Protocol (SDP) messages **[RFC4566]** for creation of media streams.

The protocol specified here defines the state machines, semantic behaviors, and messages that are exchanged between instances of the state machines. However, it does not specify the actual on-the-wire transport of these messages. Rather, it assumes that the implementation of this protocol would occur within the browser itself, and then browser APIs would allow the application's JavaScript to request creation of messages and insert messages into the state machine. The actual transfer of these messages would be the responsibility of the web application, and would utilize protocols such as HTTP and WebSockets. To facilitate implementation within a browser, JSON notation is used to describe the messages **[RFC4627]**.

The protocol defined here covers media control, but does not provide any call control. Concepts like ringing of phones, user search, call forwarding, redirection, transfer, hold, and so on, are all the domain of call processing and are out of scope for this specification. It is assumed that the application running within the browser provides any call control based on the needs of the application, the scope of which is not a matter of standardization.

Despite that fact that it has an abstract transport, ROAP is still a protocol. This means it has state machines, and it has rules governing the behavior of those state machines which guarantee that system operates properly based on any set of inputs. It is assumed that this state machinery is implemented in the browser and thus immutable by the application, which can then guarantee proper behavior regardless of the operation of the resident JavaScript. This provides an important layer of protection.

The protocol is designed to operate between two entities (browsers for example), which exchange messages "directly" - meaning that a message output by one entity is meant to be directly processed by the other entity without further modification. In practice, this means that a web server can treat ROAP messages as opaque and just shuffle them between browser instances. This allows for simple implementations. However, more powerful applications can be built in which the web server or JavaScript can modify the messages in order to provide more complex features. As long as those modifications produce messages compliant to this specification, **SDP Offer/Answer** [RFC3264], **SDP** [RFC4566], **ICE** [RFC5245] and any other dependencies, the modifications are permissible.

This protocol is designed for two major use cases:

- Browser to browser
- Browser to SIP device via a SIP gateway

In the browser to SIP use case, the gateway obviously needs to be somewhat more sophisticated. However, because this design is a small subset of the design space covered by SIP **[RFC3261]**, it is intended to be simple to translate to and from/SIP via a signalling gateway. Moreover, many of the elements in messages have clear mappings to elements in SIP messages, thus allowing simple, stateless translation.

2. Requirements and Design Goals

There has been extensive debate about the best architecture for RTCWeb signaling. To a great extent this decision is dictated by the requirements that the signaling mechanism is intended to fit. The protocol in this document was designed to minimize the amount of implementation effort required outside the browser and RTC-Web signaling gateways. This implies the following requirements:

- It should be possible to develop a simple browser to browser voice and video service in a small amount of code. In particular, it MUST be possible to implement a functional service such that:
  - The web service maintains only transaction state, not call state;
  - In the browser to browser case, the web server can simply pass protocol messages between the browser agents without examining or modifying them;
  - The service operates without needing to examine the details of the browser capabilities (e.g., new codecs should be automatically accommodated without modifying either the service or the associated JS.

- It should be possible to implement a simple RTC-Web gateway that:
  - Connects to legacy SIP devices ranging from multiscreen video phones to PSTN gateways;
  - Has a deterministic mapping between RTC-Web messages and SIP messages;
  - Permits the mechanical translation of messages without knowledge of the details of all the browser capabilities;
  - Maintains only transaction state, not call state; and
  - Does not need to send or receive the media (unless also acting as a relay or a translator for codecs which are not jointly supported).

Finally it seems clear that SDP is too complicated to reinvent, so despite its manifest deficiencies we opt to take it as-is rather than trying to reinvent it.

3. Terminology

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

This draft uses the API and terminology described in **[webrtc-api]**.

TOC

4. Protocol Overview

We start with a simple example. Consider the case where browser A wishes to setup up a media session with browser B. At the high level, A needs to communicate the following information:

- This is a new media session and not an update to a different session.
- Here is my SDP offer, including media parameters and ICE candidates.

The OFFER message is used to carry this information. For example, A might send B:

```
{
  "messageType":"OFFER",
  "offererSessionId":"13456789ABCDEF",
  "seq": 1
  "sdp":"
v=0\n
o=- 2890844526 2890842807 IN IP4 192.0.2.1\n
s= \n
c=IN IP4 192.0.2.1\n
t=2873397496 2873404696\n
m=audio 49170 RTP/AVP 0"
}
```

The messageType field indicates that this is an OFFER and the offererSessionId indicates the media session that this OFFER is associated with. B can tell that this is for a new media session because it contains a offererSessionId that he has not seen before. The sdp field contains the offer itself, which is just an ordinary SDP offer rendered as a string.

If B elects to start a media session, B responds with an ANSWER message containing SDP, as shown below.

```
{
  "messageType":"ANSWER",
  "offererSessionId":"13456789ABCDEF",
  "answererSessionId":"abc1234356",
  "seq": 1,
  "sdp":"
v=0\n
o=- 2890844526 2890842807 IN IP4 192.0.2.3\n
s= \n
c=IN IP4 192.0.2.3\n
t=2873397496 2873404696\n
m=audio 49175 RTP/AVP 0"
}
```

The contents of this message are more or less the same as those in the OFFER, except that B also includes a answererSessionId to uniquely identify the session from B's perspective. The combination of offererSessionId and answererSessionId uniquely identifies this session.

Finally, in order to confirm that A has seen B's ANSWER, A responds with an OK message.

```
{
  "messageType":"OK",
  "offererSessionId":"13456789ABCDEF",
  "answererSessionId":"abc1234356",
  "seq": 1
}
```

Note that all of these messages contain a seq field which contains a transaction sequence number. The seq field makes it possible to correlate messages which belong to the same transaction, as well as to detect duplicates, which is described later in section **Section 5.1**.

The messageType value of "OFFER" will always contain an SDP offer, and an object with a messageType value of "ANSWER" will always contain an SDP answer. The complete list of message types is defined in **Section 5**. Only a small number of messages are permitted and much of the message set is devoted to error handling.

Once a session has been set up, additional rounds of offer/answer can be sent using the OFFER/ANSWER/OK sequence. Note that the seq attribute makes it easy to differentiate these additional rounds from the initial exchange and from each other.

TOC

5. Semantics & Syntax

TOC

5.1. Reliability Model

ROAP messages are typically carried over a reliable transport (likely HTTP via XMLHttpRequest or WebSockets), so the chance of message loss is low (though non-zero), provided that the signaling service is up. However, the common web reliability and scaleability model is based on the principle that transactions are idempotent and that requests can just be discarded and will be retried. A retry of a transaction might happened if a given host was down and the DNS round robin approach wanted to move to the next server, or if a server was

overloaded, or if there was a hiccup in the network. Web applications that want to work well need to deal with theses issues to get the advantages of the general web design pattern for scaleability and reliability.

To support this web model in this protocol, OFFER and ANSWER messages are retried by the client until they are acknowledged end to end with an ANSWER or OK. The combination of the sessionID and seq allow the browser to detect and discard duplicate requests.

5.2. Common Fields

5.2.1. Session IDs

Each call is identified by a pair of session identifiers:

offererSessionId
> The offerer's half of the session ID (supplied in the OFFER)

answererSessionId
> The answerer's half of the session ID (supplied in the response to an OFFER)

The session ID values MUST be generated so that they are globally unique. Thus, the combination of both sessionIds is itself globally unique. Session IDs never change for during an media session.

All messages MUST contain the "offererSessionId", and all messages other than OFFER or an error in response to an OFFER MUST contain both "offererSessionId" and "answererSessionId".

5.2.2. Seq

This is a sequence counter for the key requests that helps correlate responses to the correct request.

This is a 32-bit unsigned integer. On each new OFFER (from either browser) it is incremented by one. The Seq of an OK or ANSWER is set to the same Seq that was used in the OFFER which caused it. When a PeerConnection objects originates a new session by sending an OFFER type message, it starts the Seq at 1. Note: If browser A starts an OFFER/ANSWER/OK transaction with a seq of 1 to browser B, then later B initiates a second OFFER/ANSWER?/OK transaction, it will have a seq of 2.

5.2.3. More-coming

This is a boolean flag that can only appear in an ANSWER and, if set to true, indicates \that this answer is not the final answer that will be sent for the associated OFFER. If this flag is not present, it is assumed to be false.

A common situation where the flag may be set to true could be in a case where an Agent had received an OFFER and wished to immediately respond with an ANSWER that allowed ICE checking to start from both sides; but the Agent could not respond with a final ANSWER because the agent was still waiting for user authorization to determine which media should be sent. In this case, the Agent could send an ANSWER that had "more answer's coming" but that allowed ICE to start. Then later, when the user had authorized the media, the Agent could send an ANSWER with the more-coming flag set to false that indicated this was the final media selection.

This is a bit different that sending a final ANSWER with just the ICE right away then later sending an OFFER to update the media. Consider the where browser A requests video with B. When the A side that sent the initial OFFER gets an ANSWER that rejects the video, it may very well present an users interface that indicates that the there is no media. Five seconds later when browser B sends an OFFER requesting video, browser A may present an interface that ask if it is OK to do the video that was just rejected. This results in a crappy user an experience and in the extreme can result in both sides always rejecting the other sides OFFER of video, then waiting for the user to authorize video that results in a new OFFER that is always rejected.

It easier to be able to indicate that OFFER resulted in one valid ANSWER, but that the OFFER needs to be held open as other valid ANSWERS may replace the current one. This stops the other side from generating new a new OFFER while this is taking place. This is also needed to support a SIP gateway doing early media.

5.3. Media Setup

In order to initiate sending media between the browsers, the offerer sends an OFFER message. In order to accept the media, the answerer responds with an ANSWER message. A sample message flow for this is shown below:
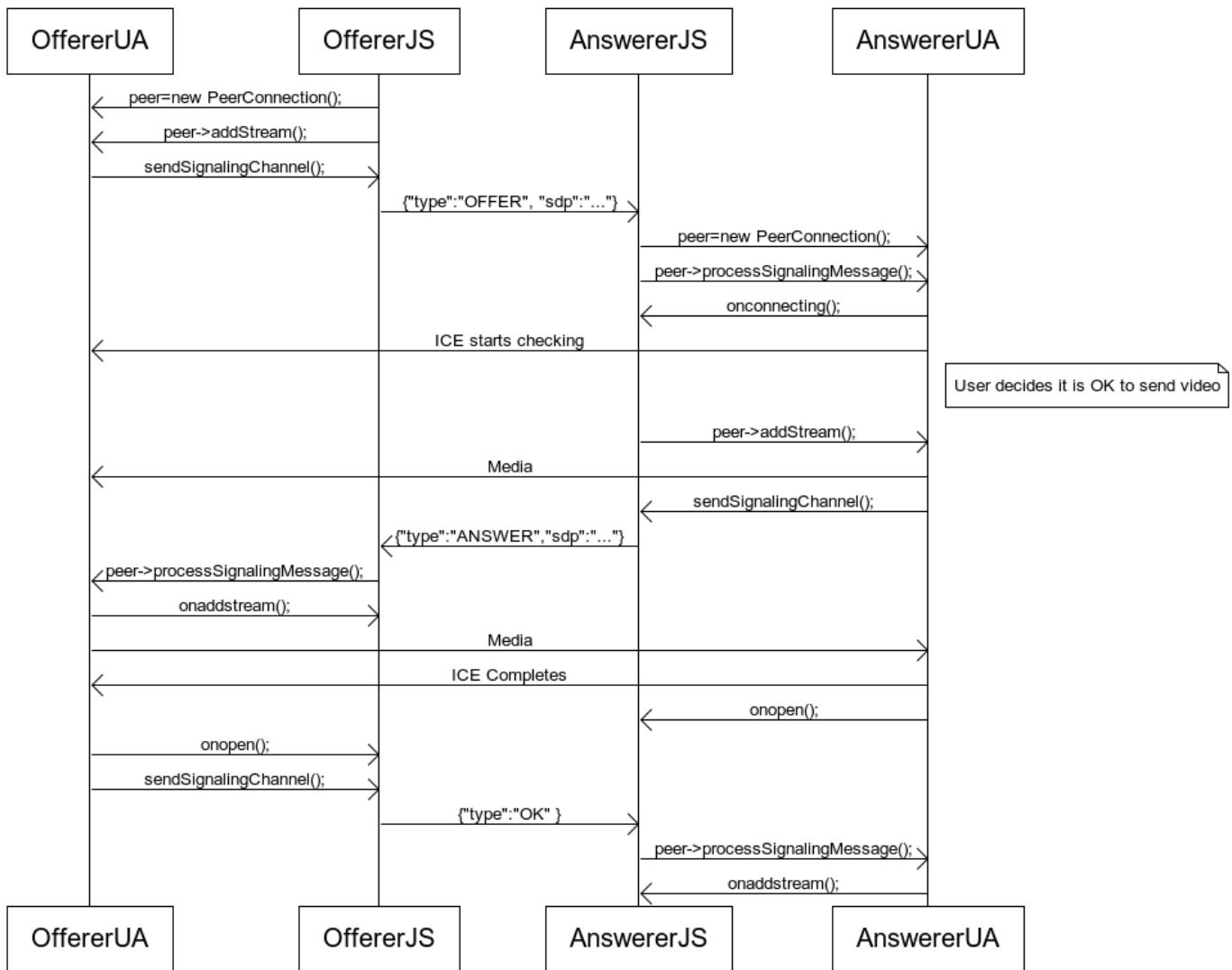
The above figure shows a simple message flow for negotiating media:

- The offerer sends an OFFER to initiate the call;
- At this point, ICE negotiation starts;
- Once the browser authorizes sending media to the far side, the answerer sends an ANSWER containing the media parameters; and finally,
- Once ICE is completed and an OK to the ANSWER is received, both sides know that media can flow.

The contents of each of these messages is detailed below.

5.3.1. OFFER Message

The first OFFER message with a given offererSessionId is used to indicate the desire to start a media session.

5.3.1.1. Offerer Behavior

In order to start a new media session, a offerer constructs a new OFFER message with a fresh offererSessionId. The answererSessionId field MUST be empty. Like all SDP offers, the message MUST contain an "sdp" field with the offerer's offer.

5.3.1.2. Answerer Behavior

A answerer can receive an OFFER in three cases:

- A new session (this is detected by seeing a new offererSessionId value);
- A retransmit of a new OFFER (known offererSessionId, empty answererSessionId); or
- A request to change media parameters (known offererSessionId, known answererSessionId, new seq value).

The first two situations are described in this section. The third case is described in **Section 5.4**. Any other condition represents an alien packet and SHOULD be rejected with Error: NOMATCH

If no media session exists with the given "offererSessionId" value, then this is a new media session. The answerer has three primary options:

- Reject the request, either silently with no response or with an Error: REFUSED message;
- Reply to the OFFER message with a final ANSWER message; or **Section 5.3.2**
- Send back a non final ANSWER message and then later respond with an final ANSWER.

In either of the latter two cases, the answerer performs the following steps:

1. Generate a "answererSessionId" value;
2. Create some local call state (i.e., a PeerConnection object) and bind it to the "offererSessionId"/"answererSessionId" pair. All future messages on this session MUST then be delivered to that PeerConnection object;
3. Start ICE handshaking with the offerer; and finally,
4. Respond with a message containing an SDP answer in the "sdp" field. This will contain the answerer's (potentially provisional) media information and the ICE parameters.

If an OFFER is received that has already been received and responded to and the media session still exists, then the answerer MUST respond with the same message as before. If the session has been terminated in the meantime, then an ERROR:NOMATCH message SHOULD be sent.

5.3.2. ANSWER

The ANSWER message is used by the receiver of an OFFER message to indicate that the offer has been accepted. The ANSWER message MUST contain the answererSessionId for this media session and an sdp parameter containing ICE candidates and the final media parameters for the session (although of course these can be adjusted by a new OFFER/ANSWER exchange. See **Section 5.4**)

5.3.3. OK

The OK message is used by the receiver of an ANSWER message to indicate that it has received the ANSWER message. It has no contents itself and is merely used to stop the retransmissions of the ANSWER.

5.3.4. ERROR

The ERROR message is used to indicate that there has been an error. The contents and semantics of this message are defined in **Section 5.5**.

5.4. Changing Media Parameters

Once a call has been set up, it is common to want to adjust the media parameters, e.g., to add video to an audio-only call. This is also done with the OFFER/ANSWER/OK sequence of messages, though the details are slightly different.

Either side may initiate a new OFFER/ANSWER exchange by sending an OFFER message. However, implementations MUST NOT attempt this for sessions which are still in active negotiation. Specifically, the offerer MUST NOT send a new OFFER until it has received the ANSWER, and the answerer MUST NOT send a new OFFER until it has received the OK indicating receipt of the ANSWER.

A new OFFER MUST contain a complete set of media parameters describing the proposed new media configuration as well as a full set of ICE parameters. The recipient of a new OFFER on a valid connection MUST respond with an appropriate ANSWER message. However that message MAY refuse to accept the proposed new configuration. If the session has been terminated in the meantime, then an ERROR:NOMATCH message SHOULD be sent.

5.4.1. Conflicting OFFERS (glare)

Note:
> The algorithm described here models what is used in SIP today. There is a backwards compatible proposal that may turn out to work better. If that evolves, it will probably be used to replace the algorithm described here.

Because a change of media parameters may be initiated by either side, there is a potential for the change requests to occur simultaneously (i.e., "glare"). When an agent which has sent an OFFER and not yet received an ANSWER receives an OFFER from the other side, it MUST respond with an ERROR: CONFLICT message.

An offerer which receives an Error: conflict message MUST either abandon the attempted capability change or generate a timer of T seconds, with T chosen as follows:

1. If the offerer is the offerer, T has a randomly chosen value between 2.1 and 4 seconds in units of 10 ms.
2. If the offerer is the answerer, T has a randomly chosen value of between 0 and 2 seconds in units of 10 ms.

When the timer fires, the offerer SHOULD increment the Seq and attempt a new OFFER once more, if it still desires that session modification to take place. The new OFFER might be the same as the original offer (other than the seq) or it might be different.

[FIGURE: Glare]

The following figure assumes the previous message flow has happened and media is flowing.

OffererUA   OffererJS   AnswererJS   AnswererUA

"Hi, Let's do video"

"Sounds great"

peer->addStream(new MediaStream());

sendSignalingChannel();

peer->addStream(new MediaStream());

sendSignalingChannel();

peer->processSignalingMessage();

peer->processSignalingMessage();

sendSignalingChannel();

sendSignalingChannel();

peer->processSignalingMessage();

peer->processSignalingMessage();

wait 1.1 seconds

sendSignalingChannel();

{"type":"OFFER", "sdp":"..."}

peer->processSignalingMessage();

sendSignalingChannel();

{"type":"ANSWER", "sdp":"..."}

peer->processSignalingMessage();

One way Video

sendSignalingChannel();

{"type":"OK"}

peer->processSignalingMessage();

onaddstream();

wait 2.7 seconds

sendSignalingChannel();

{"type":"OFFER", "sdp":"..."}

peer->processSignalingMessage();

sendSignalingChannel();

{"type":"ANSWER", "sdp":"..."}

peer->processSignalingMessage();

Both way Video

sendSignalingChannel();

{"type":"OK"}

peer->processSignalingMessage();

onaddstream();

OffererUA   OffererJS   AnswererJS   AnswererUA

5.4.2. Premature OFFER

It is an error, though technically possible, for an agent to generate a second OFFER while it already has an unanswered OFFER pending. An

It is an error, though technically possible, for an agent to generate a second OFFER while it already has an unanswered OFFER pending. An agent which receives such an offer MUST respond with an ERROR: FAILED message containing a "RetryAfter" attribute generated as a random value from 0 to 10 seconds.

5.5. Errors

Errors are indicated by the messageType "ERROR". All errors MUST contain an "errorType" field indicating the type of error which occurred and echo the "seq" value (if any) and the session id values of the message which generated the error. The following sections describe each error type.

5.5.1. NOMATCH

An implementation which receives a message with either an unknown offererSessionId (for an OFFER) or an unknown offererSessionId/answererSessionId pair SHOULD respond with a NOMATCH error.

5.5.2. TIMEOUT

The TIMEOUT error is used to indicate that the corresponding message required some processing which timed out. For instance, an agent which is a SIP gateway translates ROAP signaling messages into SIP messages. If those SIP messages time out, the gateway would generate a TIMEOUT error.

5.5.3. REFUSED

An agent which has received an initial OFFER MAY indicate its refusal of the media session by sending a REFUSED error. Note that this error is not required; an agent MAY simply drop the OFFER with no acknowledgement at all. However, agents which do not wish to accept subsequent OFFERS SHOULD [OPEN ISSUE: MUST?] send a REFUSED in order to avoid timeouts and confusion on the offerer side.

5.5.4. CONFLICT

The CONFLICT error is used to indicate that an agent has received an OFFER while it has its own OFFER outstanding. The offerer's behavior in response to this error is defined in **Section 5.4.1**.

5.5.5. FAILED

FAILED is a catch-all error indicating that something went wrong while processing a message. A FAILED error MAY contain a "retryAfter" field, which indicates the time (in seconds) after which the message MAY be retried (though retries are OPTIONAL).

6. Security Considerations

TBD

7. Companion APIs

Note:
> This section may need to move to the requirements draft**[I-D.ietf-rtcweb-use-cases-and-requirements]** but for now it is convenient to put it here just to help see how all the pieces fit together.

The offer / answer concepts in this draft are not enough to meet all the use cases of RTCWeb. They need to be combined with some additional functionality that the browser exposes to the JavaScript applications. This additional functionality loosely falls into three categories: capabilities, hints, and stats. The capabilities allow the JS application to find out what video codecs and capabilities a given browser supports before initiating a media session. The hints provide a way for the JS application to provide useful information to the browser about how the media will be used so that the browser can negotiate appropriate codecs and modes. Stats provides statistics about what the current media sessions. The capabilities, hints, and stats do not need to be communicated between the two browsers, so they are not specified in this draft. However, this drafts assumes the existence of API so that these three can be used to build complete systems. Some of the assumptions about these APIs are described in the following sections.

7.1. Capabilities

The APIs need to provide a way to find out the capabilities as defined in section 9 of RFC 3264. This allows the JS to find out the codecs that the browser supports.

7.2. Hints

When creating a new PeerConnection in a browser, the application needs to be able to provide optional hints to the browser about

When creating a new PeerConnection in a browser, the application needs to be able to provide optional hints to the browser about preferences for the media to be negotiated. These include:

1. Whether the session has audio, video, or both;
2. Whether the audio is spoken voice or music;
3. Preferred video resolution and frame rate (perhaps these just come from the MediaTrack objects);
4. Whether the video should prefer temporal or spatial fidelity;
5. <add more here>

The JS applications should also be able to update and change these hints mid-session. Some types of hint changes may simply impact the parameter on various codecs and require no signalling to the other end of the media stream. Other types of hint changes may cause a new offer answer exchange.

7.3. Stats

Several parts of the media session create statistics that are important to some applications. APIs should provide the JS applications with information on the following statistics:

1. Total IP data rate for the session;
2. ICE statistics including current candidates, active pairs, RTT;
3. RTP statistics including codecs selected, parameters, and bit rates;
4. RTCP statistics including packet loss rate; and
5. SRTP statistics.

---

8. Relationship with SIP & Jingle

The SIP **[RFC3261]** specifies an application protocol that provides a complete solution for setting up and managing communications on the Internet. It combines both "call processing" functions - identity and name spaces, call routing, user search, call features, authentication, and so on - as well as media processing through its transport of SDP and support for the offer/answer model.

In a web context, application processing can be done through proprietary logic implemented in Javascript/HTML, along with proprietary logic implemented in the web server, and proprietary messaging transported through HTTP and WebSockets. One of the advantages of the web is to allow a rich set of applications to be built without changing the browser. Although application processing and be done in JavaScript and the web servers, we do require raw media control in the browser. ROAP basically extracts the offer/answer media control processing used in SIP, and puts it into an protocol that can operate independently of SIP itself.

The information contained in ROAP messages corresponds closely to the offer/answer information carried by complete solutions such as SIP and Jingle, so it is straightforward to build gateways to and from ROAP. These gateways need only translate the signaling, while allowing end-to-end media without the need for media relays (except, of course, for NAT traversal.) In the case of SIP, which uses SDP directly, such gateways would translate between SIP and ROAP, while transporting SDP end-to-end. In the case of Jingle **[XEP-0166]**, it would also be necessary to translate between SDP and the Jingle offer/answer format; **[XEP-0167]** describes such a mapping.

---

9. IANA Considerations

This document requires no actions from IANA.

---

10. Acknowledgments

Many thanks for comment, ideas, and text from Eric Rescorla, Harald Alvestrand, Magnus Westerlund, Ted Hardie, and Stefan Hakansson.

---

11. Open Issues

How to negotiate support for enhancements to this JSON message. (consider supported / required)

Common way to indicate destination in offer going to a signalling gateway.

Need to generate proper ASCII art version of message flows.

---

12. References

12.1. Normative References

[RFC4627]   Crockford, D., "**The application/json Media Type for JavaScript Object Notation (JSON)**," RFC 4627, July 2006 (**TXT**).
[RFC3264]   Rosenberg, J. and H. Schulzrinne, "**An Offer/Answer Model with Session Description Protocol (SDP)**," RFC 3264, June 2002 (**TXT**).
[RFC2119]   Bradner, S., "**Key words for use in RFCs to Indicate Requirement Levels**," BCP 14, RFC 2119, March 1997 (**TXT**, **HTML**, **XML**).
[RFC4566]   Handley, M., Jacobson, V., and C. Perkins, "**SDP: Session Description Protocol**," RFC 4566, July 2006 (**TXT**).

12.2. Informative References

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

	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 (**TXT**).
[XEP-0166]	**Ludwig, S.**, **Beda, J.**, **Saint-Andre, P.**, **McQueen, R.**, **Egan, S.**, and **J. Hildebrand**, "**Jingle**," XSF XEP 0166, December 2009.
[XEP-0167]	**Ludwig, S.**, **Saint-Andre, P.**, **Egan, S.**, **McQueen, R.**, and **D. Cionoiu**, "**Jingle RTP Sessions**," XSF XEP 0167, December 2008.
[RFC5245]	Rosenberg, J., "**Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols**," RFC 5245, April 2010 (**TXT**).
[webrtc-api]	Bergkvist, Burnett, Jennings, Narayanan, "WebRTC 1.0: Real-time Communication Between Browsers," October 2011.
	Available at http://dev.w3.org/2011/webrtc/editor/webrtc.html
[I-D.ietf-rtcweb-use-cases-and-requirements]	Holmberg, C., Hakansson, S., and G. Eriksson, "**Web Real-Time Communication Use-cases and Requirements**," draft-ietf-rtcweb-use-cases-and-requirements-06 (work in progress), October 2011 (**TXT**).

Authors' Addresses

Cullen Jennings
Cisco
170 West Tasman Drive
San Jose, CA 95134
USA
Phone: +1 408 421-9990
Email: **fluffy@cisco.com**

Jonathan Rosenberg
jdrosen.net
Email: **jdrosen@jdrosen.net**
URI: **http://www.jdrosen.net**

	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 (**TXT**).
[XEP-0166]	**Ludwig, S.**, **Beda, J.**, **Saint-Andre, P.**, **McQueen, R.**, **Egan, S.**, and **J. Hildebrand**, "**Jingle**," XSF XEP 0166, December 2009.
[XEP-0167]	**Ludwig, S.**, **Saint-Andre, P.**, **Egan, S.**, **McQueen, R.**, and **D. Cionoiu**, "**Jingle RTP Sessions**," XSF XEP 0167, December 2008.