Network Working Group	C. Jennings
Internet-Draft	Cisco
Intended status: Standards Track	J.R. Rosenberg
Expires: May 03, 2012	jdrosen.net
	J. Uberti
	Google
	R. Jesup
	Mozilla
	October 31, 2011

RTCWeb Offer/Answer Protocol (ROAP)
draft-jennings-rtcweb-signaling-01

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.
The IETF has been notified of intellectual property rights claimed in regard to some or all of the specification contained in this document. For more information consult the online list of claimed rights.

Status of this Memo

Copyright Notice

Table of Contents

*Authors' Addresses

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, messages are encoded in JSON [RFC4627]. This protocol, with appropriate selected transports, could also be implemented by a signalling gateway that converts ROAP to SIP or Jingle.
This protocol is designed to be closely aligned with the PeerConnection API defined in the RTCWeb API[webrtc-api] specification. It is important to note that while ROAP does not require what has been referred to as a low level API for media manipulation, ROAP does not prevent having a such an API as well and both styles of API could coexist and be used where appropriate.
The protocol defined here 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 for 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.
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, interoperability is still possible.
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:

*It's possible to build a web service that 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;

*is only required to maintains transaction state, not call state (note is fine if an implementation want to maintain 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].

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 A's 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.
In building web systems it is often useful for a request to contain some state that is passed back in future messages. This system includes two types of state: session state and request state. If a browser receives a message that contains state in a setSessionState attribute, any future messages it sends that have the same offererSessionId MUST include this state in a sessionState attribute. Similarly if a request contains an setResponseState attribute, that state MUST be included in any response to that request in a responseState attribute.
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.
At the point that one side which to end the session, it simply sends a SHUTDOWN message which is responded to with an OK response. A SHUTDOWN can be sent regardless of it any response has been received to the initial OFFER. The key purpose of the SHUTDOWN messages is to allow the other side to know they can clean up any state associated with the session.

5. Semantics & Syntax

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. Because only the application knows what its internal reliability characteristics are, the JS application (and whatever associated servers it uses) are ultimately responsible for ensuring end-to-end delivery; the browser simply assumes that messages which are provided to the JS will be delivered eventually.
However, in order to maintain OFFER/ANSWER transaction state, the SDP state machine does need to understand when the far end has received an ANSWER if it caused an error or not. To support this model, OFFER and ANSWER messages are acknowledged end to end with an ANSWER or OK however any retransmission need to be handled by the JS or whatever is providing the transport of the ROAP messages. The combination of the sessionID and seq allow the browser to detect and discard duplicate requests and to detect glare.

NOTE: The split of the reliability model between the JS and browser is something where implementations are playing around with and trying to get some experience with what works best. This is an area that is highly likely to change as understanding of the implications evolves.

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 the duration of 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. Session Tokens

While session IDs serve to uniquely identify a session, it may be useful to allow one or another sides to offload state onto the other side (for instance to enable a stateless gateway). The "setSessionToken" and "sessionToken" fields are used for this purpose. When an implementation receives a message with a "setSessionToken" field, it MUST associate the field value with the session. For all future messages in the session MUST send the associated value in the "sessionToken" field (unless the session token is reset by another "setSessionToken" value). If no session token has yet been received, the "sessionToken" field MUST be omitted.

5.2.4. Response Tokens

In addition to tokens which persist for the life of a session, it is also possible to have tokens which are only valid for the lifetime of a given request/response pair. The "setResponseToken" and "responseToken" fields are used for this purpose.
When an implementation responds to a message from the other side (e.g., supplies an answer to an offer, or replies to an answer with an OK), it MUST copy into the "responseToken" field any value found in a "setResponseToken" field in the message being responded to. If no "setResponseToken" field is present, then the "responseToken" field MUST be omitted.

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:

```
participant OffererUA
participant OffererJS
participant AnswererJS
participant AnswererUA
OffererJS->OffererUA: peer=new PeerConnection();

OffererJS->OffererUA: peer->addStream();
OffererUA->OffererJS: sendSignalingChannel();
OffererJS->AnswererJS: {"type":"OFFER", "sdp":"..."}
AnswererJS->AnswererUA: peer=new PeerConnection();
AnswererJS->AnswererUA: peer->processSignalingMessage();
AnswererUA->AnswererJS: onconnecting();

AnswererUA->OffererUA: ICE starts checking

note right of AnswererUA: User decides it is OK to send video
AnswererJS->AnswererUA: peer->addStream();
AnswererUA->OffererUA: Media

AnswererUA->AnswererJS: sendSignalingChannel();
AnswererJS->OffererJS: {"type":"ANSWER","sdp":"..."}
OffererJS->OffererUA: peer->processSignalingMessage();
OffererUA->OffererJS: onaddstream();
OffererUA->AnswererUA: Media

AnswererUA->OffererUA: ICE Completes
AnswererUA->AnswererJS: onopen();
OffererUA->OffererJS: onopen();

OffererUA->OffererJS: sendSignalingChannel();
OffererJS->AnswererJS: {"type":"OK" }
AnswererJS->AnswererUA: peer->processSignalingMessage();
AnswererUA->AnswererJS: onaddstream();
```

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. It MUST also contain the tieBreaker field, containing a 32 bit random integer used for glare resolution as described in Section 5.4.1.

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 with moreComing=true) 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). In addition, ANSWERs MAY contain the moreComing flag, as described below.

5.3.2.1. moreComing Flag

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.
One motivating use case for moreComing is where an Agent wishes to respond immediately to an OFFER in order to start ICE checking before the user has provided authorization to send media. The Agent cannot send an ANSWER containing media information but can send ICE candidate. In this case, the Agent could send an ANSWER that had moreComing=true but that allowed ICE to start. Then later, when the user had authorized the media, the Agent could send an ANSWER with the moreComing flag=false that indicated this was the final media selection.
To see why simply having multiple independent offers (as opposed to multiple answers for a single offer), consider the case 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 a UI indication that there is no media. Five seconds later when browser B sends an OFFER requesting video, browser A may present a UI element that asks is OK to do the video that was just rejected. This results in a bad user experience and in the extreme can result in both sides always rejecting the other side's 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 which would 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.

5.3.3. 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 5.6.

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)

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"). This document defines a glare handling procedure that results in immediate resolution of the glare condition allowing one OFFER message to continue to be processed while the other is terminated. It is defined in such a way that it can interwork with SIP's glare handling mechanism. However SIP's timer based mechanism aren't suitable for the ROAP as strict requirements on ROAP message transport between end-points are not possible and thus easily could result in an repeated glare situation.

To achieve immediate resolution each OFFER message includes a 32 unsigned integer value, the tie breaker, that is randomly generated for each new OFFER message an end-point issues. Whenever a end-point receives an OFFER message that has the same sequence number as an outstanding OFFER the end-point itself sent, a glare condition has arisen. In a glare condition the end-point compares the received OFFER's tiebreaker value with the tiebreaker value of the tiebreaker in the OFFER outstanding. The OFFER with the greatest numerical value wins and that OFFER is allowed to continue being processed. IF the received OFFER lost the tie breaking an Error:CONFLICT message is sent. If it is the outstanding OFFER that lost, the end-point can expect an Error:CONFLICT message to be eventually received. However, that OFFER can immediately be considered as terminated.
Some special considerations has been made in this glare handling for interworking well with SIP glare handling as currently specified. Thus it has the notion of a gateway that converts the ROAP message into SIP message. This process is discussed in more detail below after the basic rules are defined normatively.
A regular end-point SHALL generate a random 32-bit unsigned numerical value for each OFFER message. In the case the random value becomes 0 or 4,294,967,295 a new random value SHALL be generated until it is neither values. The values 0 and 4,294,967,295 MAY be assigned to ROAP messages generated by gateways to ensure efficient glare handling towards other systems.
An ROAP message end-point that has an outstanding OFFER, i.e. an OFFER where it has not yet received an ANSWER SHALL upon receiving an OFFER perform the following processing:

1 Check if the incomming OFFER has a answererSessionId, if not it is an initial offer. If the outstanding OFFER also is an intial OFFER there is an Error. If the outstanding OFFER is not an initial OFFER and the outstanding OFFER do have answererSessionId equal to the offererSessionId in the received message then the sequence numbers are checked. In case the incomming OFFER's sequence number is equal to the sequence number of the outstanding OFFER there is glare. If the sequence number is not the same and the sequence number of the incomming is larger than the outstanding OFFER's sequence number, then this message is out of order with an ANSWER to the out-standing message. If the sequence number of the incomming is lower than the outstanding, then this is a old request.

2 In case of glare, compare the tie-breaker values for each OFFER. The tie-breaker value that is greater than the other wins. The OFFER with the winning value is processed as if there was no glare. The OFFER with the losing value is terminated, see 3A or 3B. In case the tie-breaker values are equal the double-glare case in 3C is invoked.

3A The OFFER being terminated is the received one: The end-point SHALL send a Error:CONFLICT response message.

3B
 The OFFER being terminated is this end-points outstanding OFFER: The end-point knows the OFFER will be terminated and can expect an Error:CONFLICT response. The end-point can assume this termination and MAY issue a new OFFER as soon as possible after having concluded the transactions for the winning OFFER.

3C The two tie-breaker values where equal, in this case both OFFERs are terminated and a Error:DOUBLCONFLICT message is sent. Both of the Offerer SHOULD re-attempt their offers by generating new OFFER messages, these messages SHALL have new tie-breaker values and incremented sequence number. Also gateways SHOULD generate random values, as one reason for this double conflict is that two gateways have become interconnected and both selects either 0 or 4,294,967,295.

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

```
participant OffererUA
participant OffererJS
participant AnswererJS
participant AnswererUA

note left of OffererJS: "Hi, Let's do video"
note right of AnswererJS: "Sounds great"
OffererJS->OffererUA: peer->addStream( new MediaStream() );
OffererUA->OffererJS: sendSignalingChannel();
AnswererJS->AnswererUA: peer->addStream( new MediaStream() );
AnswererUA->AnswererJS: sendSignalingChannel();
OffererJS->AnswererJS: {"type":"OFFER", tiebreaker="123", "sdp":"..."}
AnswererJS->OffererJS: {"type":"OFFER", tiebreaker="456", "sdp":"..."}
AnswererJS->AnswererUA: peer->processSignalingMessage();
OffererJS->OffererUA: peer->processSignalingMessage();


OffererUA->OffererJS: sendSignalingChannel();
AnswererUA->AnswererJS: sendSignalingChannel();
OffererJS->AnswererJS: {"type":"ERROR",error="conflict","sdp":"..."}
AnswererJS->OffererJS: {"type":"ANSWER", "sdp":"..."}
AnswererJS->AnswererUA: peer->processSignalingMessage();
OffererJS->OffererUA:  peer->processSignalingMessage();

OffererUA->OffererJS: sendSignalingChannel();
OffererJS->AnswererJS: {"type":"OK"}
AnswererJS->AnswererUA: peer->processSignalingMessage();
AnswererUA->AnswererJS: onaddstream();


AnswererUA->AnswererJS: sendSignalingChannel();
AnswererJS->OffererJS: {"type":"OFFER", tiebreaker="789", "sdp":"..."}
OffererJS->OffererUA: peer->processSignalingMessage();
OffererUA->OffererJS: sendSignalingChannel();
OffererJS->AnswererJS: {"type":"ANSWER", "sdp":"..."}
AnswererJS->AnswererUA: peer->processSignalingMessage();
AnswererUA->OffererUA: Both way Video
AnswererUA->AnswererJS: sendSignalingChannel();
AnswererJS->OffererJS: {"type":"OK"}
OffererJS->OffererUA: peer->processSignalingMessage();
OffererUA->OffererJS: onaddstream();
```

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 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. Notification of Media Termination

The SHUTDOWN message is used to indicate the termination of an existing session. Either side may initiate a SHUTDOWN at any time during the session, including while the initial OFFER is outstanding (i.e., before an ANSWER has been sent/received.)

TODO - FIX NAMES

```
participant OffererUA
participant OffererJS
participant AnswererJS
participant AnswererUA

OffererJS->OffererUA: peer->close();
OffererUA->OffererJS: sendSignalingChannel();
OffererJS->AnswererJS: { "type":"SHUTDOWN" }
AnswererJS->AnswererUA: peer->processSignalingMessage();
AnswererUA->AnswererJS: onclose();

AnswererUA->AnswererJS: sendSignalingChannel();
AnswererJS->OffererJS: {"type":"OK"}
OffererJS->OffererUA: peer->processSignalingMessage();
OffererUA->OffererJS: onclose();
```

Upon receipt of a SHUTDOWN which corresponds to an existing session, an agent MUST immediately terminate the session and send an OK message. Subsequent messages directed to this session MUST result in an Error:NOMATCH message. Similarly, on receipt of the OK, the agent which sent the SHUTDOWN MUST terminate the session and SHOULD respond to future messages with Error:NOMATCH.

5.6. 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.

5.6.1. 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.

5.6.2. 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.

5.6.3. 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.

5.6.4. 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.

5.6.5. DOUBLECONFLICT

The DOUBLECONFLICT error is used to indicate the tiebreaker values in CONFLICT were the same. See Section 5.4.1.

5.6.6. 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 PeerConenction 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

The text for the glare resoltuion section was provided by Magnus Westerlund. 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.
[RFC3264]	Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, June 2002.
[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
[RFC4566]	Handley, M., Jacobson, V. and C. Perkins, "SDP: Session Description Protocol", RFC 4566, July 2006.

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 Protocol", RFC 3261, June 2002.
[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.
[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", Internet-Draft draft-ietf-rtcweb-use-cases-and-requirements-06, October 2011.

Authors' Addresses

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

Jonathan Rosenberg Rosenberg jdrosen.net EMail: jdrosen@jdrosen.net URI: http://www.jdrosen.net

Justin Uberti Uberti Google, Inc.

Randell Jesup Jesup Mozilla EMail: randell-ietf@jesup.org