Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 19, 2013

E. Ivov
Jitsi
E. Marocco
Telecom Italia
P. Thatcher
Google

June 17, 2013

No Plan: Economical Use of the Offer/Answer Model in WebRTC Sessions with Multiple Media Sources
draft-ivov-rtcweb-noplan-01

Abstract

This document describes a model for the lightweight use of SDP Offer/Answer in WebRTC. The goal is to minimize reliance on Offer/Answer exchanges in a WebRTC session and provide applications with the tools necessary to implement the signalling that they may need in a way that best fits their custom requirements and topologies. This simplifies signalling of multiple media sources or providing RTP Synchronisation source (SSRC) identification in multi-party sessions. Another important goal of this model is to remove from clients topological constraints such as the requirement to know in advance all SSRC identifiers that they could potentially introduce in a particular session.

The model described here is similar to the one employed by the data channel JavaScript APIs in WebRTC, where methods are supported on PeerConnection without being reflected in SDP.

This document does not question the use of SDP and the Offer/Answer model or the value they have in terms of interoperability with legacy or other non-WebRTC devices.

Status of This Memo

Copyright Notice

Table of Contents

1. Background

In its early stages the RTCWEB working group chose to use the Session

Description Protocol (SDP) and the Offer/Answer model [RFC3264] when establishing and negotiating sessions. This choice was also accompanied by the decision not to mandate a specific signalling protocol so that, once interoperability has been achieved, web applications can choose the semantics that best fit their

requirements. In some scenarios however, such as those involving the use of multiple media sources, these choices have left open the issue of exactly which operations should be handled by SDP Offer/Answer and which of them should be left to application-specific signalling.

At the time of writing of this document, the RTCWEB working group is considering two approaches to addressing the issue, that are often referred to as Plan A [PlanA] and Plan B [PlanB]. Both of them describe semantics that require Offer/Answer exchanges in a number of situations where this could be avoided, particularly when adding or removing media sources to a session. This requirement applies equally to cases where a client adds the stream of a newly activated web cam, a simulcast flow or upon the arrival or departure of a conference participant.

Plan A handles such notifications with the addition or removal of independent m= lines [PlanA], while Plan B relies on the use of multiplexed m= lines but still depends on the Offer/Answer exchanges for the addition or removal of media stream identifiers [MSID].

By taking the Offer/Answer approach, both Plan A and Plan B take away from the application the opportunity to handle such events in a way that is most fitting for the use case, which, among other things, also goes against the working group's decision to not to define a specific signalling protocol. (It could be argued that it is therefore only natural how proponents of each plan, having different use cases in mind, are remarkably far from reaching consensus).

Reliance on preliminary announcement of SSRC identifiers is another issue. While this could be perceived as relatively straightforward in one-to-one sessions or even conference calls within controlled environments, it can be a problem in the following cases:

- interoperability with legacy/non-WebRTC endpoints
- use within non-controlled and potentially federated conference

environments where new RTP streams may appear relatively often. In such cases the signalling required to describe all of them through Offer/Answer may represent substantial overhead while none or only a part of it (e.g. the description of a main, active speaker stream) may be required by the application.

By increasing the number of Offer/Answer exchanges Both Plan A and Plan B also increase the risk of encountering glare situations (i.e. cases where both parties attempt to modify a session at the same time). While glare is also possible with basic Offer/Answer and resolution of such situations must be implemented anyway, the need to frequently resort to such code may either negatively impact user

experience (e.g. when "back off" resolution is used) or require substantial modifications in the Offer/Answer model and/or further venturing into the land of signalling protocols [ROACH-GLARELESS-ADD].

2. Introduction

The goal of this document is to provide directions for use of the SDP Offer/Answer model in a way that satisfies the following requirements:

- the addition and removal of media sources (e.g. conference participants, multiple web cams or "slides") must be possible without the need of Offer/Answer exchanges;
- the addition or removal of simulcast or layered streams must be possible without the need for Offer/Answer exchanges beyond the initial declaration of such capabilities for either direction.
- call establishment must not require preliminary announcement or even knowledge of all potentially participating media sources;
- application specific signalling should be used to cover most semantics following call establishment, such as adding, removing or identifying SSRCs;
- straightforward interoperability with widely deployed legacy endpoints with rudimentary support for Offer/Answer. This includes devices that allow for one audio and potentially one

video m= line and that expect to only ever be required to render a single RTP stream at a time for any of them. (Note that this does NOT include devices that expect to see multiple "m=video" lines for different SSRCs as they can hardly be viewed as "widely deployed legacy").

To achieve the above requirements this specification expects that browsers and WebRTC endpoints in general will only use SDP Offer/Answer to establish transport channels and initialize an RTP stack and codec/processing chains. This also includes any renegotiation that requires the re-initialisation of these chains. For example, adding VP8 to a session that was setup with only H.264, would obviously still require an Offer/Answer exchange.

All other session control and signalling are to be left to applications.

The actual Offer/Answer semantics presented here do not differ fundamentally from those proposed by Plan A and Plan B. The main

differentiation point of this approach is the fact that the exact protocol mechanism is left to WebRTC applications. Such applications or lightweight signalling gateways can then implement either Plan A, or Plan B, or an entirely different signalling protocol, depending on what best matches their use cases and topology.

3. Reliance on Offer/Answer

The model presented in this specification relies on use of SDP and Offer/Answer in quite the same way as many of the pre-WebRTC (and most of the legacy) endpoints do: negotiating formats, establishing transport channels and exchanging, in a declarative way, media and transport parameters that are then used for the initialization of the corresponding stacks.

The following is an example presenting what this specification views as a typical offer sent by a WebRTC endpoint:

```
v=0
o=- 0 0 IN IP4 198.51.100.33
s=
```

```
t=0 0

a=group:BUNDLE audio video                 // declaring BUNDLE Support
c=IN IP4 198.51.100.33
a=ice-ufrag:Qq8o/jZwknkmXpIh               // initializing ICE
a=ice-pwd:gTMACiJcZv1xdPrjfbTHL5qo
a=ice-options:trickle
a=fingerprint:sha-1                        // DTLS-SRTP keying
      a4:b1:97:ab:c7:12:9b:02:12:b8:47:45:df:d8:3a:97:54:08:3f:16

m=audio 5000 RTP/SAVPF 96 0 8
a=mid:audio
a=rtcp-mux

a=rtpmap:96 opus/48000/2                   // PT mappings
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000

a=extmap:1 urn:ietf:params:rtp-hdrext:csrc-audio-level  //5825 header
a=extmap:2 urn:ietf:params:rtp-hdrext:ssrc-audio-level  //extensions

[ICE Candidates]

m=video 5002 RTP/SAVPF 97 98
a=mid:video
a=rtcp-mux
```



```
a=rtpmap:97 VP8/90000     // PT mappings and resolutions capabilities
a=imageattr:97 \
  send [x=[480:16:800],y=[320:16:640],par=[1.2-1.3],q=0.6] \
       [x=[176:8:208],y=[144:8:176],par=[1.2-1.3]] \
  recv *
a=rtpmap:98 H264/90000
a=imageattr:98 send [x=800,y=640,sar=1.1,q=0.6] [x=480,y=320] \
               recv [x=330,y=250]

a=extmap:3 urn:ietf:params:rtp-hdrext:fec-source-ssrc   //5825 header
a=extmap:4 urn:ietf:params:rtp-hdrext:rtx-source-ssrc   //extensions

a=max-send-ssrc:{*:1}                      // declaring maximum
a=max-recv-ssrc:{*:4}                      // number of SSRCs
```

```
[ICE Candidates]
```

The answer to the offer above would have roughly the same structure and content. The most important aspects here are:

- Preserves interoperability with most kinds of legacy or non-WebRTC endpoints.
- Allows the negotiation of most parameters that concern the media/RTP stack (typically the browser).
- Only a single Offer/Answer exchange is required for session establishment and, in most cases, for the entire duraftion of a session.
- Leaves complete freedom to applications as to the way that they are going to signal any other information such as SSRC identification information or the addition or removal of RTP streams.

3.1. Interoperability with Legacy

Interoperating with the "widely deployed legacy endpoints" is one of the main reasons for the RTCWEB working group to choose the SDP Offer/Answer model as basis for media negotiation. It is hence important to clarify the compatibility claims that this specification makes.

A "widely deployed legacy endpoint" is considered to have the following characteristics:

- Likely to use the SIP protocol.
- Capability to gracefully handle one audio and potentially one video m= line in an SDP Offer.
- Capability to render one SSRC per m=line at any given moment but multiple, consecutive SSRCs over a period of time. This would be the case with transferred session replacements for example. While

the capability to handle multiple SSRCs simultaneously is not uncommon it cannot be relied upon and should first be confirmed by signalling.

- Possibly have features such as ICE, BUNDLE, RTCP-MUX, etc. Just as likely not to.

- Very unlikely to announce in SDP the SSRCs that they intend to use for a given session.

- Exact set of features and capabilities: Guaranteed to be wildly and widely diverse.

While it is relatively simple for RTCWEB to accommodate some of the above, it is obviously impossible to design a model that could simply be labeled as "compatible with legacy". It is reasonable to assume that use cases involving use of such endpoints will be designed for a relatively specific set of devices and applications. The role of the WebRTC framework is to hence provide a least-common-denominator model that can then be extended by applications.

It is just as important not to make choices or assumptions that will render interoperability for some applications or topologies difficult or even impossible.

This is exactly what the use of Offer/Answer discussed here strives to achieve. Audio/Video offers originating from WebRTC endpoints will always have a maximum of one audio and one video m= line. It will be up to applications to determine exactly how many streams they can afford to send once such a session has been established. The exact mechanism to do this is outside the scope of this document (or WebRTC in general).

Note that it is still possible for WebRTC endpoints to indicate support for a maximum number of incoming or outgoing streams for reasons such as processing constraints. Use of the "max-send-ssrc" and "max-recv-ssrc" attributes [MAX-SSRC] could be one way of doing this, although that mechanism would need to be extended to provide ways of distinguishing between independent flows and complementary ones such as layered FEC and RTX. Even with this in mind it is still

important, not to rely on the presence of that indication in incoming

descriptions as well as to provide applications with a way of retrieving such capabilities from the WebRTC stack (e.g. the browser).

Determining whether a peer has the ability to seamlessly switch from one SSRC to another is also left to application specific signalling. It is worth noting that protocols such as SIP for example, often accompany SSRC replacements with extra signalling (re-INVITEs with a "replaces" header) that can easily be reused by applications or mapped to something that they deem more convenient.

For the sake of interoperability this specification strongly advises against the use of multiple m= lines for a single media type. Not only would such use be meaningless to a large number of legacy endpoints but it is also likely to be mishandled by many of them and to cause unexpected behaviour.

Finally, it is also worth pointing out that there is a significant number of feature rich non-WebRTC applications and devices that have relatively advanced, modern sets of capabilities. Such endpoints hardly fit the "legacy" qualification. Yet, as is often the case with novel and/or proprietary applications, they too have adopted diverse signalling mechanisms and the requirements described in this section fully apply when it comes to interoperating with them.

4. Additional Session Control and Signalling

- Adding and removing RTP streams to an existing session.
- Accepting and refusing some of them.
- Identifying SSRCs and obtaining additional metadata for them (e.g. the user corresponding to a specific SSRC).

All of the above semantics are best handled and hence should be left to applications. There are numerous existing or emerging solutions, some of them developed by the IETF, that already cover this. This includes CLUE channels [CLUE], the SIP Event Package For Conference State [RFC4575] and its XMPP variant [COIN] as well as the protocols defined within the Centralised Conferencing IETF working group [XCON] . Additional mechanisms, undoubtedly many based on JSON, are very likely to emerge in the future as WebRTC applications address varying use cases, scenarios and topologies.

The most important part of this specification is hence to prevent certain assumptions or topologies from being imposed on applications. One example of this is the need to know and include in the Offer/

Answer exchange, all the SSRCs that can show up in a session. This can be particularly problematic for scenarios that involve non-WebRTC endpoints.

Large scale conference calls, potentially federated through RTP translator-like bridges, would be another problematic scenario. Being able to always pre-announce SSRCs in such situations could of course be made to work but it would come at a price. It would either require a very high number of Offer/Answer updates that propagate the information through the entire topology, or use of tricks such as pre-allocating a range of "fake" SSRCs, announcing them to participants and then overwriting the actual SSRCs with them. Depending on the scenario both options could prove inappropriate or inefficient while some applications may not even need such information. Others could be retrieving it through simplistic means such as access to a centralized resource (e.g. an URL pointing to a JSON description of the conference).

5. Demultiplexing and Identifying Streams (Use of Bundle)

This document assumes use of BUNDLE in WebRTC endpoints. This implies that all RTP streams are likely to end up being received on the same port. A demuxing mechanism is therefore necessary in order for these packets to then be fed into the appropriate processing chain (i.e. matched to an m= line).

> Note: it is important to distinguish between the demultiplexing and the identification of incoming flows. Throughout this specification the former is used to refer to the process of choosing selecting a depacketizing/decoding/processing chain to feed incoming packets to. Such decisions depend solely on the format that is used to encode the content of incoming packets.
>
> The above is not to be confused with the process of making rendering decision about a processed flow. Such decisions include showing a "current speaker" flow at a specific location, window or video tag, while choosing a different one for a second, "slides" flow. Another example would be the possibility to attach "Alice", "Bob" and "Carol" labels on top of the appropriate UI components. This specification leaves such rendering choices entirely to application-specific signalling as described in Section 4.

This specification uses demuxing based on RTP payload types. When creating offers and answers WebRTC applications MUST therefore allocate RTP payload types only once per bundle group. In cases where rtcp-mux is in use this would mean a maximum of 96 payload

types per bundle [RFC5761]. It has been pointed out that some legacy devices may have unpredictable behaviour with payload types that are

outside the 96-127 range reserved by [RFC3551] for dynamic use. Some applications or implementations may therefore choose not to use values outside this range. Whatever the reason, offerers that find they need more than the available payload type numbers, will simply need to either use a second bundle group or not use BUNDLE at all (which in the case of a single audio and a single video m= line amounts to roughly the same thing). This would also imply building a dynamic table, mapping SSRCs to PTs and m= lines, in order to then also allow for RTCP demuxing.

While not desirable, the implications of such a decision would be relatively limited. Use of trickle ICE [TRICKLE-ICE] is going to lessen the impact on call establishment latency. Also, the fact that this would only occur in a limited number of cases makes it unlikely to have a significant effect on port consumption.

An additional requirement that has been expressed toward demuxing is the ability to assign incoming packets with the same payload type to different processing chains depending on their SSRCs. A possible example for this is a scenario where two video streams are being rendered on different video screens that each have their own decoding hardware.

While the above may appear as a demuxing and a decoding related problem it is really mostly a rendering policy specific to an application. As such it should be handled by app. specific signalling that could involve custom-formatted, per-SSRC information that accompanies SDP offers and answers.

6. Simulcasting, FEC, Layering and RTX (Open Issue)

From a WebRTC perspective, repair flows such as layering, FEC, RTX and to some extent simulcasting, present an interesting challenge, which is why they are considered an open issue by this specification.

On the one hand they are transport utilities that need to be understood, supported and used by browsers in a way that is mostly transparent to applications. On the other, some applications may need to be made aware of them and given the option to control their

use. This could be necessary in cases where their use needs to be signalled to non-WebRTC endpoints in an application specific way. Another example is the possibility for an application to choose to disable some or all repair flows because it has been made aware by application-specific signalling that they are temporarily not being used/rendered by the remote end (e.g. because it is only displaying a thumbnail or because a corresponding video tag is not currently visible).

One way of handling such flows would be to advertise them in the way suggested by [RFC5956] and to then control them through application specific signalling. This options has the merit of already existing but it also implies the pre-announcement and propagation of SSRCs and the bloated signalling that this incurs. Also, relying solely on Offer/Answer here would expose an offerer to the typical race condition of repair SSRCs arriving before the answer and the processing ambiguity that this would imply.

Another approach could be a combination of RTCP and RTP header extensions [RFC5285] in a way similar to the one employed by the Rapid Synchronisation of RTP Flows [RFC6051]. While such a mechanism is not currently defined by the IETF, specifying it could be relatively straightforward:

Every packet belonging to a repair flow could carry an RTP header extension [RFC5285] that points to the source stream (or source layer in case of layered mechanisms).

Again, these are just some possibilities. Different mechanisms may and probably will require different extensions or signalling ([SRCNAME] will likely be an option for some). In some cases, where layering information is provided by the codec, an extensions is not going to be necessary at all.

In cases where FEC or simulcast relations are not immediately needed by the recipient, this information could also be delayed until the reception of the first RTCP packet.

7. WebRTC API Requirements

One of the main characteristics of this specification is the use of

SDP for transport channel setup and media stack initialisation only. In order for applications to be able to cover everything else it is important that WebRTC APIs actually allow for it. Given the initial directions taken by early implementations and specification work, this is currently almost but not entirely possible.

The following is a list of requirements that the WebRTC APIs would need to satisfy in order for this specification to be usable. (Note: some of the items are already possible and are only included for the sake of completeness.)

1. Expose the SSRCs of all local MediaStreamTrack-s that the application attaches to a PeerConnection.

2. Expose the SSRCs of all remote MediaStreamTrack-s that are received on a PeerConnection

3. Expose to applications all locally generated repair flows that exist for a source (e.g. FEC and RTX flows that will be generated for a webcam) their types relations and SSRCs.

4. Expose information about the maximum number of incoming streams that can be decoded and rendered.

5. Applications should be able to pause and resume (disable and enable) any MediaStreamTrack. This should also include the possibility to do so for specific repair flows.

6. Information about how certain MediaStreamTrack-s relate to each other (e.g. a given audio flow is related to a specific video flow) may be exchanged by applications after media has started arriving. At that point the corresponding MediaStreamTrack-s may have been announced to the application within independent MediaStream-s. It should therefore be possible for applications to join such tracks within a single MediaStream.

The following section Section 7.1 provides suggestions for addressing the above requirements.

7.1. Suggested WebRTC API Using TrackSendParams

This document proposes that the following methods and dictionaries be

added to the WebRTC API. The changes follow the model of createDataChannel, which has a JS method on PeerConnection that makes it possible to add data channels without going through SDP. Furthermore, just like createDataChannel allows 2 ways to handle neogitation (the "I know what I'm doing; Here's what I want to send; Let me signal everything" mode and the "please take care of it for me; send an OPEN message" mode), this also has 2 ways to handle negotiation (the "I know what I'm doing; Here's what I want to send; Let me signal everything" mode and the "please take care of it for me; send SDP back and forth" mode).

Following the success of createDataChannel, this allows simple applications to Just Work and more advanced applications to easily control what they need to. In particular, it's possible to use this API to implement either Plan A or Plan B.

```
// The following two method are added to RTCPeerConnection
partial interface RTCPeerConnection {
 // Create a stream that is used to send a source stream.
 // The MediaSendStream.description can be used for signalling.
 // No media is sent until addStream(MediaSendStream) is called.
 LocalMediaStream createLocalStream(MediaStream sourceStream);
```



```
 // Create a stream that is used to receive media from the remote side,
 // given the parameters signalled from MedaiSendStream.description.
 MediaStream createRemoteStream(MediaStreamDescription description);
}

interface LocalMediaStream implements MediaStream {
  // This can be changed at any time, but especially before calling
  // PeerConnection.addStream
  attribute MediaStreamDescription description;
}

// Represents the parameters used to either send or receive a stream
// over a PeerConnection.
dictionary MediaStreamDescription {
  MediaStreamTrackDescription[] tracks;
}
```

```
  // Represents the parameters used to either send or receive a track over
  // a PeerConnection.  A track has many "flows", which can be grouped
  // together.
  dictionary MediaStreamTrackDescription {
    // Same as the MediaStreamTrack.id
    DOMString id;

    // Same as the MediaStreamTrack.kind
    DOMString kind;

    // A track can have many "flows", such as for Simulcast, FEC, etc.
    // And they can be grouped in arbitrary ways.
    MediaFlowDescription[] flows;
    MediaFlowGroup[] flowGroups;
  }

  // Represents the parameters used to either send or receive a "flow"
  // over a PeerConnection.  A "flow" is a media that arrives with a
  // single, unique SSRC.  One to many flows together make up the media
  // for a track.  For example, there may be Simulcast, FEC, and RTX
  // flows.
  dictionay MediaFlowDescription {
    // The "flow id" must be unique to the track, but need not be unique
    // outside of the track (two tracks could both have a flow with the
    // same flow ID).
    DOMString id;

    // Each flow can go over its own transport.  If the JS sets this to a
```



```
    // transportId that doesn't have a transport setup already, the
    // browser will use SDP negotiation to setup a transport to back that
    // transportId.  If This is set to an MID in the SDP, then that MID's
    // transport is used.
    DOMString transportId;

    // The SSRC used to send the flow.
    unsigned int ssrc;

    // When used as receive parameters, this indicates the possible list
    // of codecs that might come in for this flow.  For exmample, a given
```

```
    // receive flow could be setup to receive any of OPUS, ISAC, or PCMU.
    // When used as send parameters, this indicates that the first codec
    // should be used, but the browser can use send other codecs if it
    // needs to because of either bandwidth or CPU constraints.
    MediaCodecDescription[] codecs;
  }


  dictionary MediaFlowGroup {
    DOMString type;  // "SIM" for Simulcast, "FEC" for FEC, etc
    DOMString[] flowids;
  }

  dictionary MediaCodecDescription {
    unsigned byte payloadType;
    DOMString name;
    unsigned int? clockRate;
    unsigned int? bitRate;
    // A grab bag of other fmtp that will need to be further defined.
    MediaCodecParam[] params;
  }

  dictionary MediaCodecParam {
    DOMString key;
    DOMString value;
  }
  }
```

Some additional notes:

- When LocalMediaStreams are added using addStream, onnegotiatedneeded is not called, and those streams are never reflected in future SDP exchanges. Indeed, it would be impossible to put them in the SDP without first resolving if that would be Plan A SDP or Plan B SDP.

- Just like piles of attributes would need to be defined for Plan A and for Plan B, similar attributes would need to be defined here (Luckily, much work has already been done figuring out what those parameters are :).

API Pros:

- Either Plan A or Plan B or could be implemented in Javascript using this API
- It exposes all the same functionality to the Javascript as SDP, but in a much nicer format that is much easier to work with.
- Any other signalling mechanism, such as Jingle or CLUE could be implemented using this API.
- There is almost no risk of signalling glare.
- Debugging errors with misconfigured descriptions should be much easier with this than with large SDP blobs.

API Cons:

- Now there are two slightly different ways to add streams: by creating a LocalMediaStream first, and not. This is, however, analogous to setting "negotiated: true" in createDataChannel. On way is "Just Work", and the other is more advanced control.
- All the options in MediaCodecDescription are a bit complicated. Really, this is only necessary because Plan A requires being able to specify codec parameters per SSRC, and set each flow on different transports. If we did not have this requirement, we could simplify.

7.1.1. Example 2

Following is an example of how these API additions would be used:

```
// Imagine I have MyApp, handles creating a PeerConnection,
// signalling, and rendering streams.  This is how the new API could be
// used.
var peerConnection = MyApp.createPeerConnection();

// On sender side:
var stream = MyApp.getMediaStream();
var localStream = peerConnection.createSendStream(stream);
sendStream.description = MyApp.modifyStream(localStream.description)
```

```
MyApp.signalAddStream(localStream.description, function(response)) {
  if (!response.rejected) {
    // Media will not be sent.
    peerConnection.addStream(localStream);
  }
}

// On receiver side:
MyApp.onAddStreamSignalled = function(streamDescription) {
  var stream = peerConnection.createReceiveStream(streamDescription);
  MyApp.renderStream(stream);
}

// In this exchange, the MediaStreamDescription signalled from the
// sender to the receiver may have looked something like this:

{
  tracks: [
  {
    id: "audio1",
    kind: "audio",
    flows: [
    {
      id: "main",
      transportId: "transport1",
      ssrc: 1111,
      codecs: [
      {
        payloadType: 111,
        name: "opus",
        // ... more codec details
      },
      {
        payloadType: 112,
        name: "pcmu",
        // ... more codec details
      }]
   }]
 },
 {
    id: "video1",
    kind: "video",
    flows: [
    {
      id: "sim0",
      transportId: "transport2",
      ssrc: 2222,
```

```
         codecs: [
```



```
         {
           payloadType: 122,
           name: "vp8"
           // ... more codec details
         }]
       },
       {
         id: "sim1",
         transportId: "transport2",
         ssrc: 2223,
         codecs: [
         {
           payloadType: 122,
           name: "vp8",
           // ... more codec details
         }]
       },
       {
         id: "sim2",
         transportId: "transport2",
         ssrc: 2224,
         codecs: [
         {
           payloadType: 122,
           name: "vp8",
           // ... more codec details
         }]
       },

       {
         id: "sim0fec",
         transportId: "transport2",
         ssrc: 2225,
         codecs: [
         {
           payloadType: 122,
           name: "vp8",
           // ...
         }]
       }],
```

```
      flowGroups: [
      {
        semantics: "SIM",
        ssrcs: [2222, 2223, 2224]
      },
      {
        semantics: "FEC",
        ssrcs: [2222, 2225]
```



```
      }]
    }]
  }
```

8. IANA Considerations

None.

9. Informative References

[CLUE] Duckworth, M., Pepperell, A., and S. Wenger, "Framework for Telepresence Multi-Streams", reference.I-D.ietf-clue-framework (work in progress), May 2013, <reference.I-D.ietf-clue-framework>.

[COIN] Ivov, E. and E. Marocco, "XEP-0298: Delivering Conference Information to Jingle Participants (Coin)", XSF XEP 0298, June 2011, <reference.I-D.ietf-coin-framework>.

[MAX-SSRC] Westerlund, M., Burman, B., and F. Jansson, "Multiple Synchronization sources (SSRC) in RTP Session Signaling ", reference.I-D.westerlund-avtcore-max-ssrc (work in progress), July 2012, <reference.I-D.westerlund-avtcore-max-ssrc>.

[MSID] Alvestrand, H., "Cross Session Stream Identification in the Session Description Protocol", reference.I-D.ietf-mmusic-msid (work in progress), February 2013, <reference.I-D.ietf-mmusic-msid>.

[PlanA] Roach, A. and M. Thomson, "Using SDP with Large Numbers of Media Flows", reference.I-D.roach-rtcweb-plan-a (work in progress), May 2013, <reference.I-D.roach-rtcweb-plan-a>.

[PlanB] Uberti, J., "Plan B: a proposal for signaling multiple media sources in WebRTC.", reference.I-D.uberti-rtcweb-plan (work in progress), May 2013, <reference.I-D.uberti-rtcweb-plan>.

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

[RFC3551] Schulzrinne, H. and S. Casner, "RTP Profile for Audio and Video Conferences with Minimal Control", STD 65, RFC 3551, July 2003.

[RFC4575] Rosenberg, J., Schulzrinne, H., and O. Levin, "A Session Initiation Protocol (SIP) Event Package for Conference State", RFC 4575, August 2006.

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

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

[RFC5956] Begen, A., "Forward Error Correction Grouping Semantics in the Session Description Protocol", RFC 5956, September 2010.

[RFC6015] Begen, A., "RTP Payload Format for 1-D Interleaved Parity Forward Error Correction (FEC)", RFC 6015, October 2010.

[RFC6051] Perkins, C. and T. Schierl, "Rapid Synchronisation of RTP Flows", RFC 6051, November 2010.

[ROACH-GLARELESS-ADD] Roach, A., "An Approach for Adding RTCWEB Media Streams

without Glare", reference.I-D.roach-rtcweb-glareless-add (work in progress), May 2013, <reference.I-D.roach-rtcweb-glareless-add>.

[SRCNAME] Westerlund, M., Burman, B., and P. Sandgren, "RTCP SDES Item SRCNAME to Label Individual Sources ", reference.I-D.westerlund-avtext-rtcp-sdes-srcname (work in progress), October 2012, <reference.I-D.westerlund-avtext-rtcp-sdes-srcname>.

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

[XCON] , "Centralized Conferencing (XCON) Status Pages", , <http://tools.ietf.org/wg/xcon/>.

Appendix A. Acknowledgements

Many thanks to Bernard Aboba and Mary Barnes, for reviewing this document and providing numerous comments and substantial input.

Authors' Addresses

Emil Ivov
Jitsi
Strasbourg 67000
France

Phone: +33-177-624-330
Email: emcho@jitsi.org

Enrico Marocco
Telecom Italia
Via G. Reiss Romoli, 274
Turin 10148
Italy

Email: enrico.marocco@telecomitalia.it

Peter Thatcher
Google
747 6th St S
Kirkland, WA 98033
USA

Phone: +1 857 288 8888
Email: pthatcher@google.com