This doc specifies an API, and the semantics thereof, for manipulating point-to-point connections that use ICE / STUN / TURN to set up audio and video flows using RTP through a Javascript interface that can be implemented in browsers.

It is expressed using WebIDL syntax, which should suffice to completely specify a Javascript API. Other specifications can specify concrete APIs that expose this functionality in other languages.

Overview

The goal of this API is to provide a high-level interface to allow developers to create media sessions, such as used in phone calls, games or multiparty video conferences.

Goals

- Support for voice and video sessions.
- Support for data transfer.
- Support for handling multiple voice and video streams, possibly multiplexing them onto a lower number of transports
- Support for encryption of media.
- Safe for untrusted content, with user permission.
- High performance; suitable for high-throughput applications such as HD video conferencing
- Does not depend on any specific signaling service or protocol.
- Able to exchange media with existing RTP/(S)AVP(F) endpoints that are ICE-aware.

Non-Goals

- Complete interoperation with legacy (non-ICE-aware) endpoints.
- Direct support for SIP or XMPP

History and previous names

This interface was at various times known as ConnectionPeer, being evolved from the ConnectionPeer of the HTML5 proposal (as of March 2011 named PeerConnection).
The current name, RtcSession, is chosen to reflect that this is intended for real-time connections at a higher level than a transport (“byte transmission”) interface.

References

Sockets API: http://dev.w3.org/html5/websockets/

The channel spec is here:
http://www.whatwg.org/specs/web-apps/current-work/#channel-messaging

The current HTML5 specification for video conferencing is here:
http://www.whatwg.org/specs/web-apps/current-work/#video-conferencing-and-peer-to-peer-communication
The section entitled “Peer to peer connections” (currently section 9.4) from that reference are intended to be obsoleted by this document.

A proposed interface "Session" by Justin Uberti contributed some concepts:
https://docs.google.com/a/google.com/document/d/16fQmZxV-Re8esLDstcIuVL3RDxj1aDY8UkJtPoRymv0/edit?hl=en#

A proposed “transport” interface, which this can possibly be layered on top of:
https://docs.google.com/a/google.com/document/d/1IpBIkkyH4UzHVUrSfUcfO-nfURk6WMYBPzca-d3PXbc/edit?hl=en#

The RtcSession interface

This specification replaces the one in the HTML5 specification as of Feb 14, 2011.

A RtcSession is a higher level object that encompasses all information about the media and data flow between two peers, and knows how to use the signalling path to handle changes in the devices.
It may encompass a number of RTP sessions. If mapped into SDP, its state at any given time can be represented by a single SDP session.

It is an interface to underlying subsystems of the browser; it makes no sense to think of it as an object with independent existence. In particular, the data flows that flow through a RtcSession, and are configured via the Stream objects described below, are not directly accessible to JavaScript; they are internal functions inside the browser.

It is created via a factory object, and initialized via a formatted string.

To illustrate the use of the API, this HTML creates a very simple videophone initiator:

```
<video id=”display”>
<audio id=”speaker”>
<device id=”camera” type=”video”>
<device id=”microphone” type=”audio”>
<script>
  configstring = “{“stun_service”: { “host”: “stun.example.com” }}”
  var factory = new RtcSessionFactory(configstring);
  var connection = factory.CreateSession();
  connection.onOutgoingNegotiationItem = function(session, item) {
      // Not specified: How to transmit the negotiation item
  }
  connection.onOutgoingNegotiationBlob = sendToServer;
  connection.addOutgoingMedia(“camera”, camera.data)
  connection.addOutgoingMedia(“microphone”, microphone.data)
  display.src = connection.expectIncomingMedia(videoconfig).url
  speaker.src = connection.expectIncomingMedia(audioconfig).url
  // At this point, we have the information needed to perform
  // the negotiation.
  connection.Connect();

  connection.onConnect = StartSendingMedia();
  StartSendingMedia() {
     alert("Your camera and microphone will now be turned on");
     connection.OutgoingMedia("camera").unmute();
          connection.OutgoingMedia("microphone").unmute();
  }
  // Function called by whatever function receives negotiation data
  function incomingNegotiationHandler(item) {
     connection.IncomingNegotiationItem(item)
  }
</script>
```

The API for a RtcSession, the object that defines a single group of media connections going to one other participant, iis defined below. It uses the concepts of "StreamSource" and "StreamSink" that are described further on in this document.

```
   [Constructor(in DOMString config)]

   interface RtcSessionFactory {
     RtcSession CreateSession();
  };


   interface RtcSession {
    // Initializes a StreamSource of a requested
    // type/configuration.
    StreamSource expectIncomingMedia(in DOMString config);
     // Alternative handler: Set up a callback instead of getting
    // the object. The callback will be called when a compatible
    // incoming media stream is requested from the other end.
    void expectIncomingMedia(in DOMString media_config,
                                    in Function media_handler);
    void addOutgoingMedia(in DOMString media_config,
                               in StreamSource stream);
    // Called when all media info has been entered.
    void connect();
    // Access to resources, indexed by ID
     StreamSink OutgoingMedia(in DOMString id);
    StreamSource IncomingMedia(in DOMString id);
    // Events
    // Negotiation finished. Media can be sent.
    attribute Function onconnect;
     attribute Function onmediachange;  // media starts or stops
     attribute Function onerror;
     // f.ex. connection broken and can't be reestablished

     // Connection negotiation.
     // The handler will
     // be called when a RtcSession reuests an outgoing
     // connection. Details TBD.
     attribute Function onOutgoingNegotiationItem;
     void IncomingNegotiationItem(DOMString item);
   // Functions for querying status are TBD.
   // More functions for detailed manipulation can be added
   /// at-will.
  }
```

Streams

A Stream is an interface that controls data flows. It supports:

- Identification
- Connecting a source (StreamSource object) to one or more sinks (StreamSink object).

The id is guaranteed to be unique across all items on the same class in an app context (there is no guarantee that it’s globally unuque, or that all StreamSources have ids distinct from all StreamSinks).

The usual usage of a stream is that one creates streams using factory functions of other objects. Some of these return objects that implement the StreamSink interface, some return objects that implement the StreamSource interface, some may return objects that implement both.

A StreamSource and StreamSink can be connected:

```
ChannelA.add_data_recipient(ChannelB)
```

Conceptually, one can imagine that this connection is implemented in terms of an “onmessage” event handler in ChannelA that calls a function in ChannelB to pass the data:

```
ChannelA.onmessage = function(event) {
    ChannelB.send(event.data)
}
```

but in practical implementation, the data will not be accessible to outside inspection.

It is possible to connect multiple StreamSources to one StreamSink (mixing). If the sink does not support the requested mixing, the add_data_recipient call will throw an <IncompatibleDestination> exception.

It is possible to connect a StreamSource to multiple StreamSinks (split, for instance used in self-view provisioning).

When a StreamSink is connected to a StreamSource, the add_data_recipient call may thrown an <IncompatibleDestination> exception if the types are incompatible (codec mismatch, or connecting a video stream to an audio-only device).

```
interface StreamSource {
   void Mute();
   void Unmute();
   // Define where data will be sent.
   void add_data_recipient(StreamSink recipient);
   readonly attribute DOMString id;
   // Where data are currently being sent.
   // It is not certain that these need to be exposed.
   attribute StreamSink[] data_recipients;
   // Callback when media stops being available
   attribute Function onerror;
}

interface StreamSink {
   readonly attribute DOMString id;
   // Conceptually, but not in practice:
   // void send(ByteArray data);
   attribute StreamSource[] data_sources;
   // Callback when it’s not possible to send data
   attribute Function onerror;
}
```

There are multiple possible destinations and sources - for instance, one may desire to display video using a <video> tag, a <canvas> tag or a WebGL interface. Rather than adapting to each specific form, we define a constructor for StreamSource and StreamSink that gives a StreamSource or a StreamSink for the DOM object that is going to be used.

function StreamSink(DOMObject something_we_can_stream_to);
function StreamSource(DOMObject something_we_can_stream_from);

These return an object with the right handlers.
If the implementation doesn’t support streaming to that particular type of device, the <IncompatibleDestination> exception is thrown.

Configuration strings

The configuration strings are serialized JSON objects.

RtcSessionFactory configuration string

The RtcSessionFactory initialization string looks like this:

```
{
 “stun_service”: { “host”: “stun.example.com”,
                   “service”: “stun”,
                   “protocol”: “udp”
```

```
                    },
 “turn_service”: { “host”: “turn.example.com” }
}
```

The protocol used for negotiation is mediated through the onconnectionrequest handler.
If the onconnectionrequest handler is instantiated, it is OK to omit the connectionmediator from the initialization string; if neither is given, “connect()” will throw an IllegalConfiguration exception.

The STUN server may either be an IP address:port literal, or be a domain name. If it is a domain name, the procedure in section 9 of RFC 5389 (SRV record lookup, with fallback to port 3478 (STUN) or 5349 (STUN over TLS)) is used to establish the IP address and port to use for STUN and TURN.
If “service” and “protocol” are omitted, they are assumed to be “stun” and “udp” for stun_service, and “turn” and “udp” for turn_service.
For TURN, the procedure is defined in RFC 5766 section 6.1. The procedure of RFC 5928 (using S-NAPTR applications) is not used.

Media configuration string

The media configuration string gives the type of media and any parameters required to refine it further. It is used as part of the input to construct a media negotiation string.

Example:

```
{
 “type”: “video”,
 “label”: “my-own-label”,
 “width”: “640”,
 “height”: “360”,
 “max-bitrate”: 1024000
}
```

“type” MUST be present. All other attributes are optional.
If “label” is present, it MUST be unique within the set of streams of this class. If it is absent, the implementation will generate an unique string.
The “type” is one of “video” or “audio”. The implementation may support other types. (TODO: Add “data” once there is an agreed proposal for how to transport data)
The “label” attribute conforms to the syntax of RFC 4574 section 4 “Label” attributes (ASCII with some syntax-sensitive characters disallowed).

For video, the attribute “size” gives the width x height in pixels of the largest display area that makes sense to the caller; it is used by the video engine to select a suitable video stream resolution, but it gives no guarantee that the resulting video stream will have exactly that resolution.

Media negotiation string

The media negotiation string is passed across the negotiation interface. It contains the information required to negotiate media.

While the SDP format is universally understood to have multiple flaws that mean we should not emulate or require it, it is also relatively common to use SDP in an offer/answer mode to communicate the information needed for setup - which means that it is at least able to represent the information needed. The fields below are picked to make it obvious how they are mapped to SDP fields; there is no assumption that all SDP fields make sense in this format.

Example:

```
{
// session level parameters go here.
"media": [
 {
 // media-level parameters go here - one array entry per element.
 // MIME type, parameters:
 label: "my-own-label",
 rtpmap: [
    "97": {codec: "video/vp8"}
 ]
 attributes: [ // these correspond to a:xxx lines
   ice-pwd: "asd88fgpdd777uzjYhagZg",
   ice-ufrag: 8hhY
   candidate: [
     {component: 1 foundation:1 generation:1 proto:UDP
      priority:2130706431
      ip:10.0.1.1 port:8998 type:host}
     {component: 1 foundation:2 proto:UDP
      priority:1694498815
      ip:192.0.2.3 port:45664
      type:srflx raddr:10.0.1.1 rport:8998}
   ]
 ]
 }
]  // end of "media"
}
```

Connection establishment event flow

To initiate a call, an application will create an RtcSession object and initialize it to know what sources and sinks to request, and then call the "connect" method. The object will then do internal processing to emit one or more calls to the "onOutgoingNegotiatonItem" callback; the media negotiation string will be sufficient to construct (if required) an SDP "offer" for use in a SIP exchange.

The responding peer, if it is of the same type, will construct an RtcSession object and call its IncomingNegotiationItem() function with the passed information. If the negotiation ins successful, it will call its onOutgoingNegotiationItem callback, which is assumed to pass the information to the initiating peer.

The initiating peer's application will then call its IncomingNegotiationItem function; if the answer is acceptable to the initiator, "onconnect" is signalled.

The session description strings sent to a RtcSession need to contain all the information needed to successfully negotiate a multimedia connection.

Appendix: SDP description of a codec setup

Copied from RFC 4317, SDP offer/answer examples.

```
  [Offer,]

    v=0
    o=alice 2890844526 2890844526 IN IP4 host.atlanta.example.com
    s=
    c=IN IP4 host.atlanta.example.com
    t=0 0
    m=audio 49170 RTP/AVP 0 8 97
    a=rtpmap:0 PCMU/8000
    a=rtpmap:8 PCMA/8000
    a=rtpmap:97 iLBC/8000
    m=video 51372 RTP/AVP 31 32
    a=rtpmap:31 H261/90000
    a=rtpmap:32 MPV/90000

 [Answer]

    v=0
    o=bob 2808844564 2808844564 IN IP4 host.biloxi.example.com
    s=
    c=IN IP4 host.biloxi.example.com
    t=0 0
    m=audio 49172 RTP/AVP 0 8
    a=rtpmap:0 PCMU/8000
    a=rtpmap:8 PCMA/8000
    m=video 0 RTP/AVP 31
    a=rtpmap:31 H261/90000
```

Appendix: ICE info in session control protocols

The session description strings need to contain all the information needed to successfully set up a bidirectional datagram transport. This section reproduces readily available examples of how this information is represented in SDP and XMPP.
This is included to make sure the expressive power of the ICE info in media negotiation strings is sufficient.

Example using SDP

Example from RFC 5245 (note that these assume ICE is in use):

a=ice-pwd:asd88fgpdd777uzjYhagZg
a=ice-ufrag:8hhY
a=candidate:1 1 UDP 2130706431 10.0.1.1 8998 typ host
a=candidate:2 1 UDP 1694498815 192.0.2.3 45664 typ srflx raddr 10.0.1.1 rport 8998

Example using XMPP

Example from XEP-176:

```
 <transport xmlns='urn:xmpp:jingle:transports:ice-udp:1'
            pwd='asd88fgpdd777uzjYhagZg'
            ufrag='8hhy'>
   <candidate component='1'
              foundation='1'
              generation='0'
              id='el0747fg11'
              ip='10.0.1.1'
              network='1'
              port='8998'
              priority='2130706431'
              protocol='udp'
              type='host'/>
   <candidate component='1'
              foundation='2'
              generation='0'
              id='y3s2b30v3r'
              ip='192.0.2.3'
              network='1'
              port='45664'
              priority='1694498815'
              protocol='udp'
              rel-addr='10.0.1.1'
              rel-port='8998'
              type='srflx'/>
  </transport>
```