Network Working Group	C. Jennings
Internet-Draft	Cisco
Intended status: Standards Track	March 08, 2011
Expires: September 09, 2011	

Architecture and API Requirements for RTC Web
draft-jennings-rtcweb-api-00

Abstract

Internet browsers and other software applications are enabling support for real time interactive voice and video. This draft outlines a set of IETF protocols that can be used for this purpose and describes the overall architecture. It also identifies the requirements for an application programming interface to control these protocols.

Status of this Memo

Copyright Notice

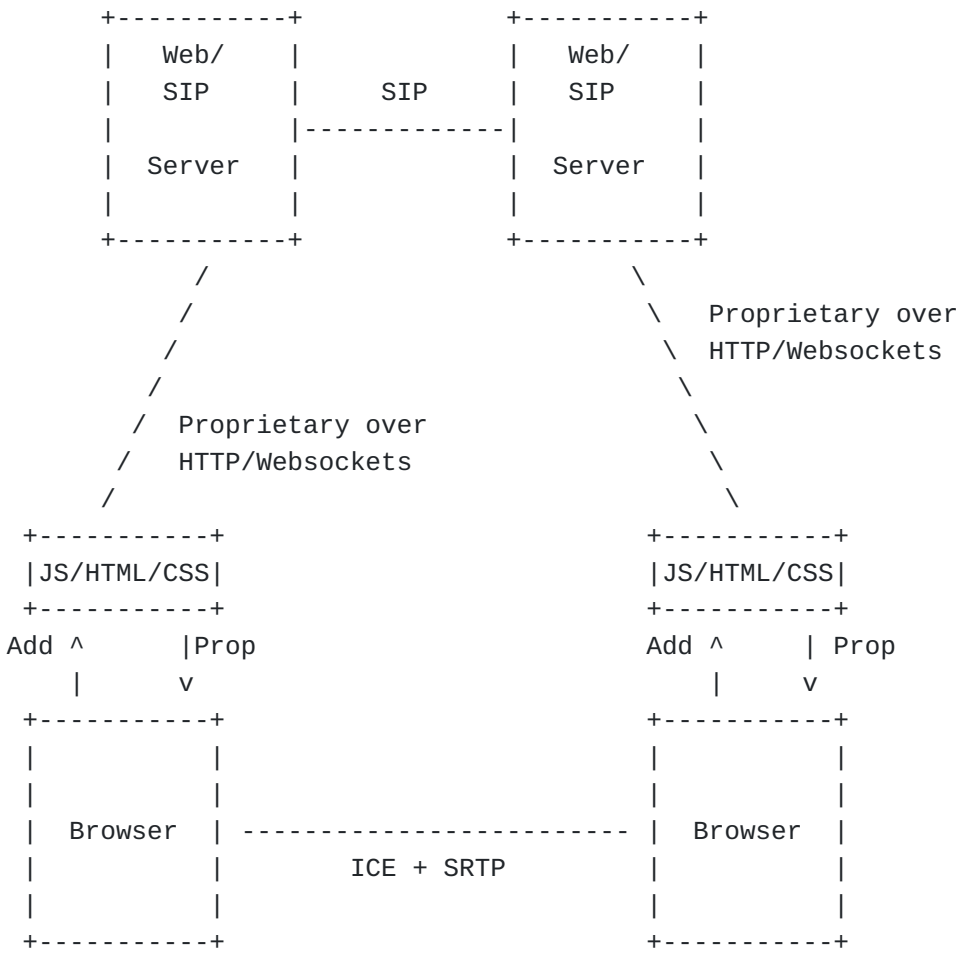
Table of Contents

1. Overview

This draft describes two models of how this would work, which are referred to as the advertisement proposal (AdProp) model and the offer answer (OffAns) model. Both of these models are useful in various situations, and they involve very similar code development efforts. This draft proposes an API and protocol set standardization that supports both models.

1.1. Advertisement Proposal Model

The AdProp model standardizes a way to send media between two browsers and standardizes an API in the browser, such that browser-based applications can find out the media capabilities of the browser and can tell the browser what media streams to send and receive. We use the term "browser app" to refer to a program that is running in the browser and using HTML, CSS, and JavaScript to control the browser. It is assumed that the browser app could communicate with the web server using existing approaches, and that the web server communicates with a SIP server as a way of federating to other websites or connecting to legacy VoIP systems. There are many different ways this model could be used, but the diagram below covers a fairly complex case that most other cases end up being a subset of. More use cases are discussed in section XXX.

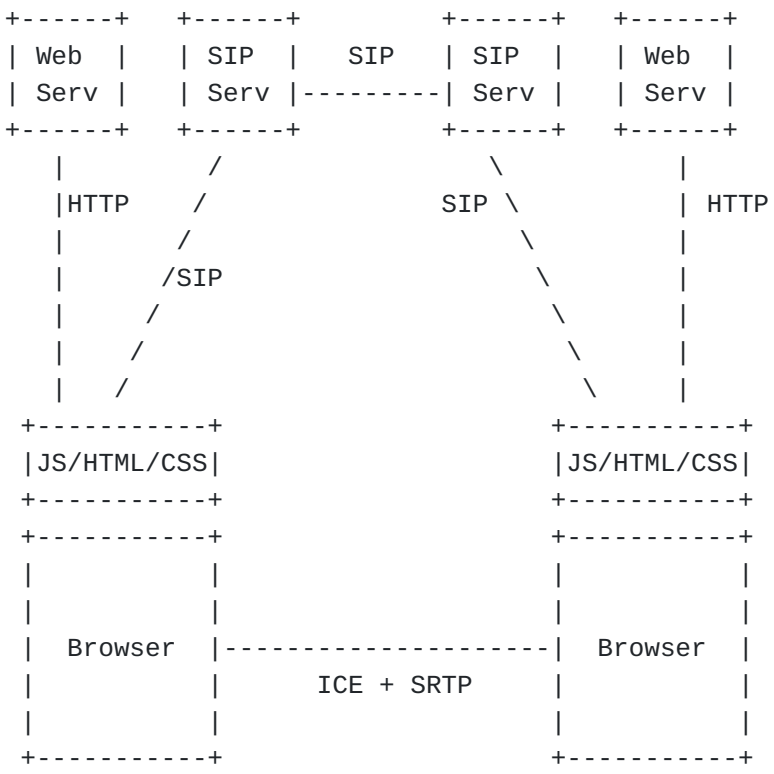


The API for this model has two distinct phases. First there is a Connection API that allows the browser app to use ICE to form a connection to the other browsers. This API assumes that the browser applications will be able to exchange ICE candidates lists by some out-of-band means -- most likely involving passing them up to the web servers over HTTP. The second stage is referred to as the AVT API. This API allows the browser apps to discover which codecs and capabilities the browser supports. It then allows the browser app to control which media streams the browser will send and receive. The browser describes its range of capabilities in an advertisement object. The browser app requests that a particular set of media streams be set up in a proposal to the browser. This is done as an atomic request which is either accepted or not. Partial acceptance has proven to be very difficult to deal with in the implementation of existing systems. The general overview and advantage of the AdProp model is discussed in draft-peterson-sipcore-advprop *[I-D.peterson-sipcore-advprop]*.
The model above shows SIP as the protocol between the two web servers, but the API proposed would also work using Jingle or H.323 as the federation signaling protocol. It would also be possible to implement the processing of SIP messages in the JavaScript in the browser application and then somehow tunnel the SIP messaging between the clients. XMPP over websockets has been proposed for this. The

architecture and API in this draft would support all of these possibilities.

1.2. Offer Answer Model

The OffAns model standardizes a way to send media between the browsers, but it also selects an existing signaling protocol to negotiate and set up the media. The browser app would indicate to the browser that it wished to form a communication session with another entity, and then the browser would take care of the rest. A typical model for this is show below.



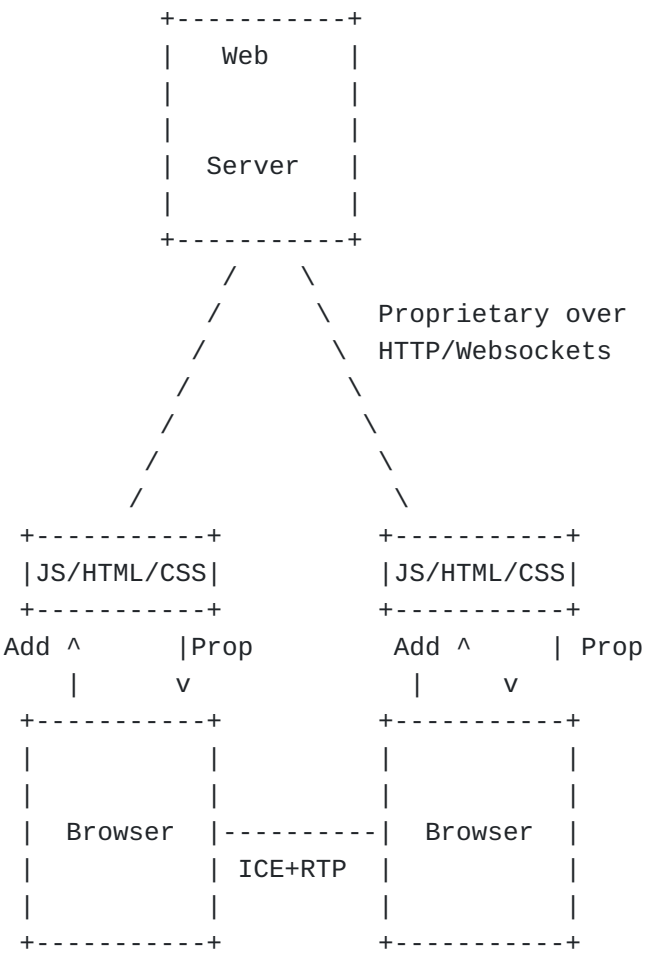
The major goal for this API is to be extremely simple to use in enabling a website for voice and video. On an iPhone today, one can simply put a tel URL on the web page and the iPhone can call it. That is a simple approach that web developers like and use. Since standards are involved, this proposal will have to be more complex. The API defines an HTML session element that can be used like a source element inside of an audio or video element. It also provides a JavaScript API to control the session and replace the user interface.

1.3. Use Cases

1.3.1. Facebook

Consider the case of a social networking site that allows IM between users and wants to also allow voice and video between them, but does

not need to federate with others. The case could easily use the AdProp model. Assuming that it was only supported on browsers meeting a certain minimum functionality and it always uses the same capabilities, there is no need to even negotiate or share the advertisements between the two browsers. The browser app simply sets up the connection to the far end, and then uses a proposal for the media steam that is always the same.

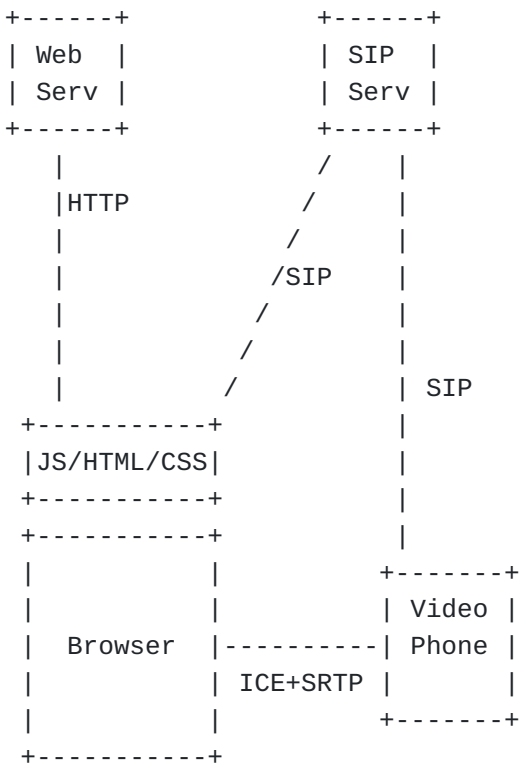


1.3.2. Webex

TBD

1.3.3. Amazon

Consider the case of a website that supports searching and displays advertisements related to the search. In this case clicking on the advertisement could directly connect the user with a sales agent at the company associated with the advertisement.



In this sort of case the people operating the web server do not need to deploy anything special to display the advertisement, and the company associated with the advertisement can use its existing call center, assuming it meets the legacy VoIP requirements outlined in section XXX. The security issue of a browser sending a SIP packet to a device that does not meet the same origin policy is discussed in the section XXX, but the brief preview of the solution is that the SIP messages can use CORS REF much like a HTTP does.

2. Terminology

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

3. Requirements

The section defines the set of protocols and selected subset profiles of these protocols that a browser would need to implement, and forms the requirements for the API to control these protocols. At a high level we split this into connection management, transports for real time media such as audio and video, transports for non media data, codecs support, and signaling protocols.
All of the data plane sessions are set up using ICE [REF] or ICE-Lite for security reasons, discussed in section XXX. Devices that could be deployed behind NATs, such as a web browser, are REQUIRED to support ICE while other devices that always deploy on public addresses can do

ICE-Lite. The only mode of ICE REQUIRED is aggressive. Real time media is transported over RTP REF or SRTP [REF]. Support for multicast RTP is OPTIONAL. To support ICE, implementation needs to be able to do STUN REF and TURN REF. In addition, there is a strong interest to define a TURN-like protocol that looks like HTTP to intermediaries, so that media can be tunneled over HTTP. Support RTCPMUX REF is REQUIRED. RTP keep alive is done using RTCP as described in REF. The API needs to allow the DSCP REF for each RTP or media stream to be set. The API needs to allow the browser app to observer and control the SSRC values in the RTP.
Open Issue: There is a desire to be able to pass non media type data directly between browsers. For example, an application such as Second Life or gaming application may wish to pass small chunks of data such as player position with stringent real time requirements. There are several proposals for how to do this. The session would be set up using ICE, just as with RTP. One proposal is just to use a thin shim on top of UDP or DTLS to demux the packets from other packets such as RTP on the same connection. Another proposal is DTLS over DCCP over UDP with some appropriate congestion control scheme chosen for DCCP. Another proposal is to define a data codec to carry the data in RTP.
The mandatory to implement audio codecs are: PCMA, PCMU, telephone-event, and opus [REF]. The API needs to support the following OPTIONAL codecs: G729, G722, G7221, G723, AMR, AMR-WB, iLBC, L16 and opus. PCMU and PMCA codecs are REQUIRED to support 1 channel with a rate of 8000 and a ptime of 20. The mandatory to implement video codecs are: <to be chosen by working group - leading candidates for consideration are H.264-AVC and VP8>. The minimum profile and resolutions supported by the mandatory to implement video codecs are TBD. The API needs to support the following OPTIONAL codecs: H263-2000, H264, H264-SVC, raw and VP8.
The signaling protocol selected here is SIP though very little overall architecture would change if the WG decided to use Jingle REF instead of SIP. The browser needs to implement the subset of SIP REF3261,3263,3264 and is required to support registration, invite, ack, cancel, bye, and update. Support for the following features is OPTIONAL: INVITES without an offer, re-invite, forking, S/MIME and sips. Support for the following is REQUIRED: sip over TLS, outbound proxy, 3xx redirects, early media, multipart mine REF 5621, update, identity 4916 & 4471, rport REF 3581, SIP keep alive as described in 5626.
Open Issue: define a TURN like protocol to tunnel RTP over HTTP
Open Issues: define a RTP mux protocol to multiplex RTP on top of a single UDP port. Would likely use SSRC as the demux code point.
Open Issue: Mandatory to implement video codec(s) and minimum profile.
Open Issue: Mandatory to implement audio codecs.

4. Connection API

It is expected this section will be removed from this draft and moved to a W3C draft but it is provided for reference at this point. The

straw man API are many things including adequate error handling. The API would likely end up using exceptions for many things.

4.1. Session API

The session element can be used anywhere in HTML that the source element could be used. Fundamentally, this is an alternative way of setting up a source for an audio or video element.
Categories: None
Contexts in which this element can be used: same as source element
Content model: Empty
Content attributes:

src: URL to destination to create session with.

aor: Address of Record that identifies this user.

credential: Password or credential for the specified AOR.

proxy: URL for outbound proxy.

DOM interface:

```
interface Session : HTMLElement {

  attribute double volume; // control speaker volume
  attribute boolean mute; // control microphone
  attribute boolean sendVideo; // control camera

  attribute DOMObject videoPane;
  attribute DOMString aorUrl;
  attribute DOMString credential;
  attribute DOMString outboundProxyURL;
  readonly attribute DOMString remoteName;
  readonly attribute boolean secure;

  readonly attribute DOMString registrationState;
  // noRegistrar, registering, registered, registrationFailed
  attribute Function onRegisterStateChange;

  void open( in DOMString url ); // tel or SIP URL
  void close();
  void accept( boolean accept );

  readonly attribute DOMString sessionState;
  // noSession, openingSession, acceptingSession,
  // inSession, closingSession
  attribute Function onSessionStateChange;

  boolean sendKeyPress( in DOMString key ); // send DTMF or KPML
  attribute Function onReceiveKeyPress;
};
```

If the session will be able to display video, the DOM object for a video tag must be provided in the videoPane parameter of the constructor. If an aorUrl is provided, the session will attempt to register for incoming calls at the server using the provided credentials. If an outbound proxy is provided, all signaling for this session will use that proxy. The progress of the registration can be tracked with the onRegisterStateChange callback. The registrationState attribute will be a string with one of the following values: noRegistrar, registering, registered, or registrationFailed.
Open Issue: need to decide how to handle credentials and if they will be in the JavaScript. Similar issues for TURN server credentials.
To make a call, the open session method is called and the session state will change to "opening session".
Events:
Exceptions:

4.1.1. Session Example Incoming

The following HTML snippet would display a video pane with a user interface such that when the user clicked, it would create an audio video session by making a SIP call to "sales@example.com".

```
<video width='320' height='240' >
  <session src="sip:sales@example.com" >
</video>
```

4.1.2. Session Example Outgoing

The following HTML snippet would register to receive calls to the address "sip:fluffy@example.com". Furthermore it would use an outbound SIP proxy at sip.example.com.

```
<video width='320' height='240' >
  <session  aor="sip:fluffy@example.com"
            credential="password"
            proxy="sip:sip.example.com" >
</video>
```

4.2. Connection API

```
[NoInterfaceObject]
interface IceCandidate {
 DOMString foundation;
 unsigned short component-id; // always 1 ?
 DOMString transport; // udp
 unsigned long priority;
 DOMString type; // host, srflx, prflx, relay
 DOMString addressFamily; // v4 v6
 DOMString connectionAddress; // v4 or v6 ip address
 unsigned short port;
};

[NoInterfaceObject]
interface IceCandidateList {
 IceCandidate  candiate[];
 DOMString icePassword;
 DOMString iceUFragment;
};

[NoInterfaceObject]
interface RelayServer {
  DOMString type; // stun turn
};

[NoInterfaceObject]
interface StunServer : RelayServer {
  DOMString ip;
};

[NoInterfaceObject]
interface TurnServer : RelayServer {
  DOMString ip;
  DOMString username;
  DOMString password;
};

[Constructor(in optional RelayServer relayServers[])]
interface Connection {
  attribute int keepAlivetime; // default 30 seconds

  attribute RelayServer relayServers[]

  readonly attribute IceCandidateList candidateList;

  readonly attribute IceCandidate connectionNearEnd;
  readonly attribute IceCandidate connectionFarEnd;

  void open( IceCandidateList addressList );

  readonly attribute DOMString state;
```

```
  // creating,ready,connecting,open,closed

  attribute Function onready;
  attribute Function onopen;

  void send(in DOMString data);
  attribute Function onmessage; // implements MessageEvent interface
  attribute Function onerror;

  void close();
  attribute Function onclose;
};
```

The general usage for a browser that had a stun server at 192.0.2.1 would be to create a connection, wait for ICE to gather candidates and the state to change to ready, then send the ICE candidates list to the far side as shown in the following code.
Open Issue: Need to add more into this so that an application can understand what is going on and get information to provide status and debug problems as well as statistics. Also may need parameters to change the algorithm.

```
 myConn = new Connection( [ {type:"stun",ip:"192.0.2.1"} ] );
 myConn.onready = function() {
   myCandidates = myConn.candidateList;
   // send myCandidates to far side
 }
```

Open issue: add text around setter calling function if in that state when set.
Later when the far side has sent its candidate list to this side, the browser app calls open to start opening the connection to the other side. Once the connection is open, the browser app can start sending and receiving data.

```
 myConn.open( farSideCandidateList );
 myConn.onOpen = function() {
   // can start sending data for far side
   myConn.send( "Hello" );
 }
 myConn.onmessage = function(e) {
   alert "Received data:" + e.data;
 }
```

4.3. Audio Video API

Note this section is far from complete and is more just a sketch to get the flavor of the interface.

```
interface Advertisement {
  CodecAd codecs[];
  boolean rtcpMux; // default true
  boolean rtpMux; // default true
  boolean srtp; // default true
  DOMString protocols[];
  // RTP/AVP, RTP/AVPF, UDP/TLS/RTP/SAVP, UDP/TLS/RTP/SAPF
  srtpSuites[]; // AES_CM_128_HMAC_SHA1_32
};

interface CodecAd{
  string mediaType;
  int clockRate;
  float minBandwidth; // kbps
  float maxBandwidth; // kbps
  boolean canReceive;
  boolean canSend;
  boolean supportDscp;
};

interface TelEventDataCodecAd {
  int supportCodes[]; // defaults to 0-11 if not present
};

interface AudioCodecAd : CodecAd {
 int maxPacketTime; // ms
};

interface IlbcAudioCodecAd : AudioCodecAd {
 int modeList [];
}

interface G729AudioCodecAd : AudioCodecAd {
  boolean vadSupported;
};

interface G711uAudioCodecAd : AudioCodecAd {
  // G.711 PCMU must be 1 channel at rate of 8000
};

interface G711aAudioCodecAd : AudioCodecAd {
  // G.711 PCMA must be 1 channel at rate of 8000
};

interface L16AudioCodecAd : AudioCodecAd {
  int rates[];
  int channels[];
  DOMString emphasis[];
  DOMString channel-order[];
};
```

```
interface AMRAudioCodecAd : AudioCodecAd {
 DOMString modeSet;
 // bunch more needed here
};

interface VideoCodecAd : CodecAd {
 float maxFramerate; // fps
 int clockRate;
 int minXsize; int maxXsize;
 int minYsize; int maxYsixe;
 float minPar; float maxPar; float parList[];
 float minSar; float maxSar; float sarList[];
};

interface VP8CodecAd : VideoCodecAd {
   int versions[];
};

interface H264CodecAd : VideoCodecAd {
  unsigned short profile-levls[];
  unsigned short max-recv-level;
  int max-mbps;
  int max-smbps;
  int max-fs;
  int max-cpb;
  int max-dpb;
  int max-br;
  boolean redundant-pic-cap;
  DOMString sprop-parameter-sets;
  DOMString sprop-level-parameter-sets;
  boolean use-level-src-parameter-sets;
  boolean in-band-parameter-sets;
  boolean level-asymmetry-allowed;
  int packetization-modes[];
  int sprop-interleaving-depth;
  int sprop-deint-buf-req;
  long deint-buf-cap;
  int sprop-init-buf-time;
 // int sprop-init-buf-time;
  long max-rcmd-nalu-size;
  int sar-understood;
  int sar-supported;
};


interface Proposal {
  StreamProp streams[];
};
```

```
interface StreamProp {
  string mediaType;
  int clockRate;
  float minBandwidth; // kbps
  float maxBandwidth; // kbps
  boolean canReceive; // default true
  boolean canSend; // default true

  DOMString fingerprint; // RFC4572
  int pTime;
  DOMString protocol;
  // RTP/AVP, RTP/AVPF, UDP/TLS/RTP/SAVP, UDP/TLS/RTP/SAPF
  long ssrc;
  int dscp;
  DOMString srtpSuites;
  int srtpKdr;
  boolean srtpUnencryptedRtcp;
  boolean srtpUnauthenticated;
  DOMString srtpFecOrder; //FEC_SRTP, "SRTP_FEC"
  int srtpLifetime; // log base 2 of max packets with one kety
  DOMString srtpKeys[];
  int srtpMki[]; // MKI corresponding to srtpKeys at same index
};

interface VideoProp : StreamProp {
  int sizex;
  int sizey;
  float sar;
  float frameRate;
};

interface AudioProp : StreamProp {
  int pTime; // ms
};

interface Stats {
   StreamStats steam[];
};

interface StreamStats {
  // TODO RTCP stats
};

interface AVT {
  attribute Connection connection;
  readonly attribute Advertisement advertisement;
  readonly attribute Advertisement advertisementNoVideo;
```

```
  attribute DOMObject camera;
  attribute DOMObject mic;
  attribute HTMLVideoElement videoPane;

  readonly attribute Stats stats;

  readonly attribute Proposal proposal;
  boolean setProposal( Proposal newProp );
};
```

Using this interface is fairly simple. First an AVT object is loaded and bound to an existing Connection object. It is also bound to cameras, microphones, and speakers, Then the current advertisement can be retried.
Open Issue: The SRTP keying should not be per stream.

```
var myAvt = org.w3c.device.load("device", "AVT", "1");

myAvt.connection = myConn; // the ICE formed connection
myAvt.camera = org.w3c.device.load("device", "camera", "1");
myAvt.mic = org.w3c.device.load("device", "mic", "1");
myAVT.videoPane = document.getElementById("myVideo");

mdAdv = myAvt.advertisement;
```

Open Issues: What's the best way to get an AVT object? How to get the other devices and wire them up to the AVT object?
Assume that the browser supports VP8 video at 720P and G.711. The myAvt object might look like:

```
{
   "codecs" :  [
      {
          "mediaType" : "PCMU",
          "clockRate": "8000",
          "maxPacketTime" : "60"
      },
      {
          "mediaType" : "PCMA",
          "clockRate": "8000",
          "maxPacketTime" : "60"
      },
      {
            "mediaType" : "VP8",
            "clockRate" : "90000",
            "maxXsize" : ""1440,
            "maxYsize" : "720",
            "parList" :  [ "1.0" ],
            "versions" : [ "1" ]
       } ],
   "protocols" : ["RTP/AVP", "RTP/AVPF" ]
};
```

Then, based on some knowledge about what the far end browser supports, the system would decide that it wants to use PCMU with VP8 at a QCIF resolution and 15fps. After forming a connection to the far end and waiting for the connection object to be in the ready state, it would construct the following proposal object and then send that proposal to the AVT systems as shown in the code below. Assuming the proposal is acceptable, the setProposal returns true and (returns false if it is not).

```
var proposal = {
  "streams" : [
      {
        "mediaType" : "VP8",
        "clockRate: : "90000",
        "protcol" : "RTP/AVP",
        "sizex" : "176",
        "sizey" : "144",
        "sar" : "1.0",
        "frameRate" : "15",
        "version" : "1"
      },
      {
        "mediaType" : "PCMU",
        "clockRate: : "8000",
        "pTime" : "20",
        "protcol" : "RTP/AVP"
      } ]
};


if ( myAvt.setProposal( proposal ) ) {
   // it worked
}
```

5. IANA Considerations

This document does not require any action of IANA.

6. Security Considerations

6.1. Attack Model

This architecture involves all the normal security consideration and attack models of HTTP, SIP and RTP but introduces yet another key issue. The assumption is that a user may browse to the attacker's website. The other assumption is that the browser is behind a firewall, and inside that firewall there are devices that would not have appropriate security models for the internet. For example, there could be SIP gateways that if sent an invite to call a 1-900 number would do so with no authentication or authorization. Whatever HTML/CS/Javascript is downloaded must not be able to send arbitrary packets to hosts behind the firewall or send SIP or RTP to devices that do not consent to communicate with the browser.

6.2. Media Security

The browser MUST enforce the constraint that no RTP or other media is sent to a given destination unless that destination completes an ICE connectivity check and proves it knows the secret generated by the

browser. The browser must keep a list of locations it has attempted to contact with ICE in the previous 30 seconds and not contact any locations that have previously failed.

6.3. Signaling Security

The browser stops unwanted SIP signaling by using CORS REF. The same CORS headers used for HTTP will be added to the SIP signaling. Before the browser sends SIP signaling, it will preflight the SIP messaging using a SIP OPTIONS message. This is done the same ways CORS can preflight check an HTTP request.

7. Legacy VoIP Interoperability

There is no way to meet all the security requirements and maintain comparability with all legacy VoIP equipment. This draft tries to minimize the impedance mismatch. The requirements here would allow interoperability with legacy VoIP equipment as long as that equipment either directly supported, or was fronted by an SBC that supported, the following: SIP CORS extension, ICE or ICE-Lite, codecs from the mandatory to implement set, supported SIP invites containing an offer, and supported DTMF over RTP with telephone events.
A substantial fraction of VoIP equipment does all of this except for the CORS extensions. The item most commonly lacking is ICE-Lite but that is becoming increasingly prevalent, particularly on devices designed to sit on the edge of a domain and connect to remote UAs that may be behind NATs. For an edge device that was willing to receive SIP call from others, implementing the CORS is pretty trivial. When the UAS receives a SIP options request with an Origin header, it checks whether the header field value is on the white list, and if it is then the UAS copies the value to the Access-Control-Allow-Origin header field value in the response. For many situations the white list would be everything, while for others it would be just the list of websites that are expected to originate calls to this SIP device.

8. Acknowledgement

Thanks to Joe Hildebrand, Matt Miller, Matthew Kaufman, Eric Rescorla and Lyndsay Campbell for their review, comments and contributed ideas.

9. References

9.1. Normative References

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

9.2. Informative References

[I-D.peterson-sipcore-advprop]	Peterson, J and C Jennings, "The Advertisement/Proposal Model of Session Description", Internet-Draft draft-peterson-sipcore-advprop-00, February 2010.

Author's Address

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