A Survey of Browser Real-Time Communication Options

Eric Rescorla
ekr@skype.net

Cullen Jennings
fluffy@cisco.com

September 20, 2010

1 Introduction

The RTC-Web project is developing protocol and API definitions to allow *real-time communications (RTC)* systems inside the browser environment. For example, it would be possible for a Web site to host a softphone as a JavaScript application which would run in an enhanced browser with no further downloads. We assume that in general we wish to avoid media gatewaying.

This document is neither a requirements document nor a specific proposal. Rather, it is intended to map out the potential design space in order to inform requirements setting and protocol design.

2 Operational Scenarios

We first consider a number of operational scenarios. This is not to say that we need to service each of these scenarios, but it is instructive to explore the technical mechanisms necessary for each.

2.1 Browser-to-Browser; Same Site

The simplest scenario is browser-to-browser communication when two browsers made by the same vendor are visiting the same site.[1] Since we wish to avoid media gatewaying this means that we need the browsers to be able to directly communicate with each other. Because permission to communicate needs to be established in the browser (Section 3.3) this means we need a standardized mechanism for performing those checks. However, we do not technically need standardized signaling, media transport, or encryption, since all this can be implemented in site-specific JavaScript.[2] As a practical matter, it's attractive to have standardized formats for at least media and security, but at some level this is an optimization.

2.2 Browser-to-Browser; Different Sites

We next consider the case of two browser users who are different sites and wish to make a phone call between each other. This case requires quite a bit more standardization than that described in

[1] We are assuming that the browsers are from different vendors, since otherwise there is no point in any kind of standards activity.

[2] There are of course performance issues here, since it means that the media and encryption need to be done in JavaScript, but since the same routines are run over and over again, the browser can simply compile them if given the right hints.

Section 2.1. In particular, since the web sites will presumably not be running the same JavaScript, it is necessary to standardize at minimum: (1) the media transport (2) media security, and (3) signaling. Clearly the media transport and the security must be standardized in the browser because they will be communicating directly. However, the signaling may either be standardized at the browser or a non-standard protocol may be used between the browser and the sites and then gatewayed to a standard protocol between the sites.

2.3 Browser-to-PSTN

Another likely use case is a browser user calling to an ordinary PSTN phone. There are two plausible designs here:

- The browser talks to some media gateway operated by the site.
- The browser talks directly to a standard PSTN gateway (whether operated by the site or not).

Obviously the second design is more desirable from the perspective of the site operator, since it requires significantly less resources on their part; the first design requires them to transit all the media, which is far more expensive. On the other hand, it requires far more effort on the browser side. (Not on the Web app, but on the browser itself). In particular, the client must be able to speak a media protocol that is consistent with the PSTN gateway (presumably RTP). There are also security issues here, see Section 3.3.

2.4 Browser-to-Unmodified SIP Phone

As well as calling an ordinary PSTN phone, a browser might wish to call an ordinary SIP phone. As in that case, this can be achieved using a gateway, but another possibility is to have the browser component speak standard protocols (SIP, RTP, ICE, etc.) and communicate directly with the SIP phone. There are also varying degrees of gatewaying here. For instance, the gateway could speak some non-standard protocol to the client and produce SIP but have RTP go directly, or could tunnel SIP from the browser.

The predominant way to connect a VoIP phone to a PSTN gateway or enterprise phone system is SIP. Even systems that use signaling protocols other than SIP, such as Google's Jingle, gateway to SIP to connect to the PSTN or enterprise systems. Today the bulk of PSTN gateways are accessible via SIP so this use case is very important for any service that wants to connect to the PSTN or other VoIP systems.

3 Other Constraints

This section describes some non use-case constraints.

3.1 No Additional Extensions

Obviously, this technology constitutes a browser extension, but it must be possible to implement whatever use cases are eventually chosen solely using JavaScript without additional browser extensions. I.e., whatever we introduce must be complete.

3.2 Browsing Safety

A basic guarantee of the browsing experience is that it is safe to visit arbitrary—even malicious—web sites without risk to the user. Obviously, this guarantee is often violated due to browser errors, but anything we design must not make the problem worse absent implementation errors by the browser vendors. This must mean that it isn't possible for a malicious Web site to subvert the user's computer. We are particularly concerned about situations where the user's expectations about input device (camera, microphone) access are violated and a malicious site is able to a bug the user or capture the media from a call initiated by another site.

3.3 Communications Permission Checks

As a special case of browsing safety, it must not be possible for any extensions we introduce to allow an arbitrary web site to send arbitrary traffic to sites which haven't consented. This is a natural extension of the *Same Origin Policy* (SOP). There are a number of mechanisms intended to allow clients to do this in the web environment (WebSockets and *Cross Origin Resource Sharing* (CORS)). We either need to use these mechanisms or build an analog for our system.

4 Overall Designs

We now consider a variety of potential designs, in rough order of increasing complexity in the browser (and generally decreasing complexity for the Web programmer). Note that these are not the only possible combinations—I am not trying to tile the space but rather describe the major likely approaches.

4.1 Raw Data Transport

The absolute minimal design is to just provide raw data transport with explicit support for connectivity tests with no ICE.[3]:

APIs

```
// Get my candidate addresses
Address[] gather_candidates(
    Connection conn,                    // The connection to gather candidates for
);

// Send a STUN connectivity test
void send_connectivity_test(
    Connection conn,                    // The connection object
    Address far_address,                // The address to send to
    String  far_username,               // The STUN username to use
    String  far_password                // The STUN password
);

// Called when a connectivity request is received
void on_connectivity_test(
    Connection conn,                     // The connection object
    Address far_address,                 // The address the ping came from
    String far_username                  // The username the peer authenticated
);
```

[3]All APIs are provided in Java-like pseudocode, since JavaScript function signatures are not that evocative.

```
// Called when a connectivity request succeeeds (STUN response received)
void on_connectivity_success(
     Connection conn,                  // The connection object
     Address far_address,              // The address the response came from
     Address reflexive_address,        // What the other side saw as our addr
     String username                   // The username used
);
```

These APIs are sufficient to construct an ICE-like implementation in the Web application (i.e., in JS). As each connectivity test succeeds, the implementation adds the relevant address pair to the access control list (i.e., the list of destinations to which packets may be sent.) Until an address is on that list, no other packets may be sent

One the channel is established, then traffic can be sent. In this minimal design, we would simply send bytes over the wire and let the other side figure it out. For instance:

```
// Send a datagram
void send(
    Connection conn,                   // The connection object
    Address far_address,               // The address to send to
    byte[]  data                       // The data to send
);

void on_recv(
    Connection conn,                   // The connection object
    Address far_address,               // The address the data came from
    byte[]  data                       // The data to send
);
```

Note that this is really raw authorized I/O. For instance, it doesn't even explicitly support relays and doesn't have any media capabilities. However, both of those can in principle be supported by this sort of mechanism, since relays can act as generic data receivers and we can implement (in principle) the media stack in JavaScript (clearly this is actually insane, but more later.)[4]

Standardization and Interoperability

In order to implement this minimal a system we would need to standardize the following:

- The connectivity checking protocol. (This could most likely be STUN).
- The APIs themselves. This includes not only the APIs above but also APIs for directly manipulating the media devices.

Once this was done, it would in principle be possible to build a simple softphone in the browser. Note that said softphone would only really support the simplest cases in an interoperable fashion, namely browser-to-browser with a single site and any other endpoints that that site chose to gateway to. It would not be possible to talk between two browsers on different sites without either gatewaying or some side-agreement about the protocols.

Comments

This isn't really a viable approach—I am presenting it solely for completeness. In particular without standardizing the media transport it's very hard to see how this API is useful, both in terms of plausibly writing an app and in terms of performance.

[4] We probably also need a construct to talk about which interface things go out.

4.2 Flow-Oriented Data Transport (Built-In ICE)

We can add to the design described in Section 4.1 by taking the concept of flow more seriously. The basic idea is to make a Connection more of a first-class object and move connection maintenance out of JavaScript and into the browser. The connection would do NAT traversal, relay management, etc., and generally guarantee the provision of a clear channel.

APIs

The general API would look something like this:

```
// Set up a connection
Connection establish_channel(
    String  far_username,               // The STUN username to use
    String  far_password                // The STUN password
    Address[] far_addresses             // The candidate addresses from the other side
);

// Called when a connection is ready
void on_established(
    Connection conn                     // The connection that was established
);
```

In this API model, when the application wanted to make a call it would call `gather_candidates()` which would do whatever STUN checks were required to determine server-reflexive addresses, set up relays, etc., and then return all the addresses to advertise. He would then transmit them to the peer in some (non-interoperable) way, and upon receipt of the response, call `establish_channel()` which would do whatever ICE-type stuff was required and call `on_established()` when the checks were finished. Note that it is possible to have `on_established()` called multiple times as candidates start to succeed. For instance, you might have `on_established()` also take a list of valid candidates and then have the caller indicate which one he wanted.

Once `on_connection()` is called, you would be able to send and receive data:

```
// Send a datagram
void send(
    Connection conn,                    // The connection object
    byte[]  data                        // The data to send
);

void on_recv(
    Connection conn,                    // The connection object
    byte[]  data                        // The data to send
);
```

Note that the APIs here are simpler than in the previous section in that they can omit the `Address` field. The concept here is that the API provides the concept of a `Connection` as an abstraction, so you just read and write to it and the implementation manages whatever network behavior is required to make it work.

Standardization and Interoperability

The standardization and interoperability profiles of this design are roughly the same as those of the raw design in Section 4.1. This design does not necessarily standardizing any additional protocols. In practice, it means actually committing to ICE, whereas in principle the design described in Section 4.1 could work without ICE, though I doubt that would actually work well.

Comments

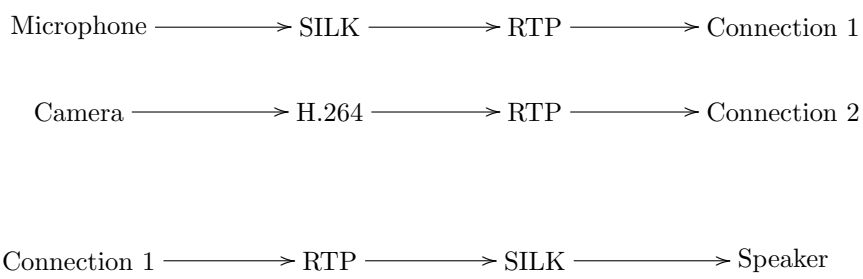
This design is largely implausible for the same reasons as Section 4.1: it is simply too much effort to make the media work well.

4.3 A/V Transport

Probably the minimal design that will actually work is to add standardized APIs not only for manipulating the media devices but for encoding and decoding data between them and the wire protocol.

APIs

The general idea here is that you will be able to "pipe-fit" devices and filters (e.g., codecs) to the network. For instance, you might want to assemble the following set of media pipelines:



In this diagram, the top two pipelines indicate the media we are generating and the bottom one indicates the media we are receiving. Thus, this call is sending audio and video but only receiving audio.

To enable this idea, we build new APIs on top of the design described in Section 4.2. These APIs are necessarily rather sketchy, but here's a sketch:

```
// Find devices
Device[] get_devices();

// Construct a new incoming pipeline
Pipeline new_recv_pipeline(
    String name,                    // An arbitrary text name for the pipeline
    Connection source,              // Where the traffic is coming from
    Filter[] filters,               // The filters that this traffic goes through
    Device sink                     // Where to sink the traffic into (i.e., play it)
);

// Construct a new outgoing pipeline
Pipeline new_send_pipeline(
    String name,                    // An arbitrary text name for the pipeline
    Device source,                  // Where the traffic is coming from (e.g., microphone)
    Filter[] filters,               // The filters that this traffic goes through
    Connection sink                 // Where to sink the traffic into (i.e., peer)
);
```

Any `Device`, `Filter`, or `Connection` interface would export a series of APIs that allowed you to manipulate the object. For instance, you might have an API which let you set bitrates, quality, etc. It's worth noting that Because JavaScript objects are duck typed, there's no reason that filter A needs to support exactly the same API points as filter B, as long as there is some guarantee

about generic API points where appropriate. For instance, you might have a published spec that says that codec A supports VBR but codec B does not, and so codec A has a `set_bit_rate()` call. This works fine in JavaScript.

All three of these object types of course are wrappers around a lot of native functionality. E.g., when we assemble a pipeline of two filters, though each one is JavaScript accessible, that doesn't mean that data between them flows through JS APIs.[5]

Note that this general design in principle supports multiplexing and traffic "forking"/"tee", simply by having a two pipelines share a component. In practice, the above API gets messy if you do that, but it's easy to imagine APIs which do not.

Standardization and Interoperability

This design require significantly more standardization effort than the previous designs. First, the API (whether declarative or procedural) must be standardized. Second these imply a common set of wire protocols for the media to flow over. Presumably this means RTP, but of course it could be anything. The major benefit of this level of standardization is that it is possible to write a simple web application with almost no knowledge of the media stack that will still run in any browser, whereas the previous designs all required quite a lot of media processing in the web application.

Note that assuming a standard media profile, it will be possible for two web applications developed by completely different people and operated by different sites to communicate directly without gatewaying provided that *the signaling is gatewayed.*

Comments

This is probably the first design we have introduced that will actually mostly work even for the single-site case.[6] Note that this design could work with or without ICE. I.e., one could have standardized media flows but over a transport that was established with NAT traversal in JS.

4.4 A/V Transport + Media Negotiation

The previous designs do not provide any interoperability for signaling at all. Adding a full call control stack obviously adds real complexity (though see Section 4.6). An intermediate approach is to avoid standardizing signaling but to standardize media negotiation only. The two natural design points here are:

- Do something very simple (e.g., `http://tools.ietf.org/html/draft-peterson-sipcore-advprop-00`).
- Do SDP.

The advantages to each of these is obvious: SDP is a lot of work but it's what everyone uses now so it provides easy gatewaying when necessary. SDP is too complicated and something simple is much easier to wrap your head around.

[5] With that said, one nice feature of this general design is that in principle you could instantiate new objects directly in JS—e.g., for testing purposes—as long as the JS runtime was smart enough about whether it was connecting to a primitive or JS code. This might put too much stress on the runtime.

[6] Good thing I spent all that time discussing other cases.

APIs

In either case, there is no overriding need for new APIs. However, if we want people to use SDP, as a practical matter we will need to provide APIs for SDP construction and parsing. I do not propose to suggest such APIs here, however.

Standardization and Interoperability

Clearly, this design would require standardizing (or adopting) a media parameters negotiation/signaling protocol. The payoff is that it significantly lowers the interoperability barrier. Two sites which wish to allow cross-site calling (Section 2.2) need only arrange for call control signaling (and simple signaling is easy) because they can just transport/tunnel the media descriptions. Clearly, if SDP is adopted this also implies that it's possible for sites to gateway to SIP without doing any media parameter translation.

Comments

What's attractive about this design is obviously that it makes interoperability easier, though not a slam dunk.

4.5 Inline Signaling

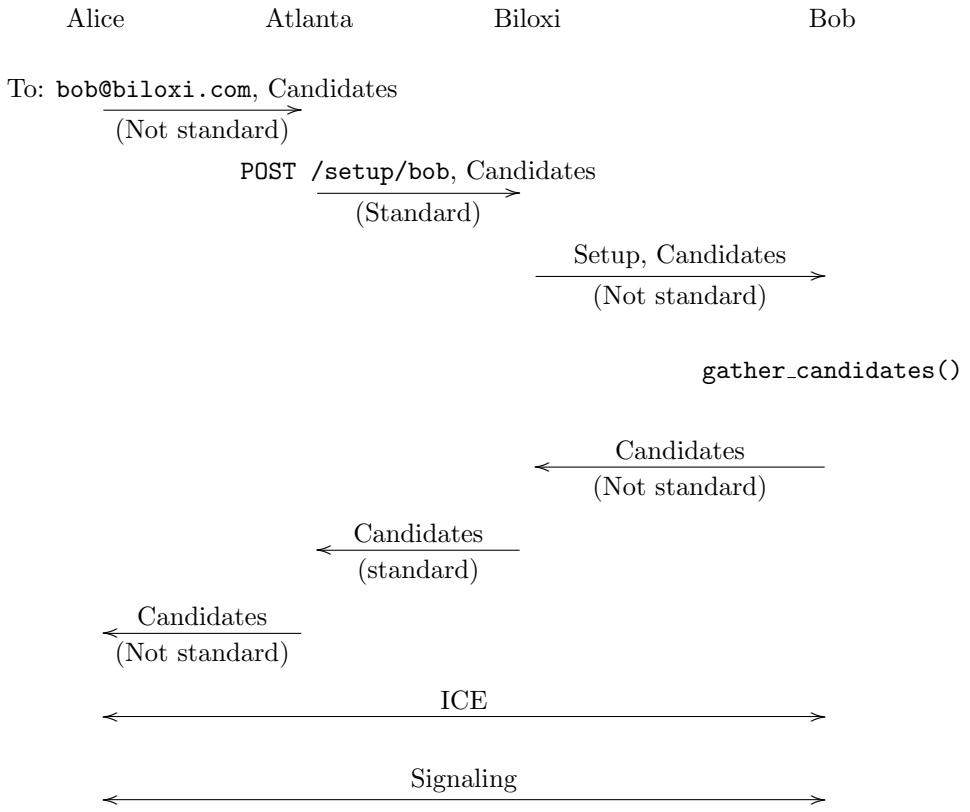
We describe two designs which are intended to provide fully non-gatewayed interoperability. The first, described in this section, is intended only to provide interoperability between new clients. The second, described in Section 4.6, is designed to provide interoperability with as many user agents as possible. The design in this section is inspired by that in RELOAD [7]. In this design:

- The communicating peers set up a direct channel.
- They send signaling traffic over that channel.
- The signaling traffic is used to set up a media channel.

In RELOAD the signaling protocol is SIP, but that doesn't add a huge amount of interoperability here because in general we want to send SIP through proxies (Section 4.6); here we consider some simpler protocol, likely one we newly designed.

As with RELOAD we have a bootstrapping problem: how do we set up the direct channel between two people on different sites? This requires the construction of some small amount of protocol machinery (presumably Web services APIs). For instance, each site could serve a standard API that allowed the exchange of ICE messages.

[7] `http://tools.ietf.org/html/draft-ietf-p2psip-base-10`



We can also standardize the Web APIs that are used to pass the candidates to and from each site, in which case it is possible to write a completely portable JS calling application.

APIs

A broad variety of APIs are possible here, depending on how ornate we want to get and how fine control we want to provide. However, the more functionality we put in the browser the easier it is for Web site operators to develop functionality for themselves.

If we do not standardize the protocols to and from the local site, then we will want to provide a standard calling API that is invoked by the JS app. For instance:

```
// Called to initiate a call
void initiate_call(
    Connection connection,              // The connection to call on
    String  far_username,               // The STUN username to use
    String  far_password                // The STUN password
    Address[] far_addresses             // The candidate addresses from the other side
    [...]                               // Requested media parameters
);

// Called when a call is established
void on_established(
    Connection conn                     // The connection that was established
```

```
    [...]                               // Some description of established media
);
```

In the limit if we standardize the entire ICE passing functionality, then one could imagine providing a truly simple API:

```
void initiate_call(
    String callee_address,              // The email-style callee address
    [...]                               // Requested media parameters
);
```

Indeed, one could imagine simply providing a URI which used whatever the default media parameters are, e.g., `rtc:ekr@example.com`.

Obviously, it would still be possible to provide lower-level APIs as well.

Standardization and Interoperability

The minimum standards profile for this design would be the inband signaling protocol. As discussed above, this means that some custom JS would be required for each site to manage the discovery of ICE candidates (both to and from the site and between sites). The next natural step is inter-site candidate exchange and then finally from browser to site. If only the first two of these are standardized, then interoperability becomes possible between any two browser clients, but the browser app itself will not be portable. If all three are standardized, then the browser apps become portable.

None of these provide interoperability with any SIP endpoint or PSTN gateway. That must be provided via a signaling gateway, though perhaps without a need for media gatewaying.

Comments

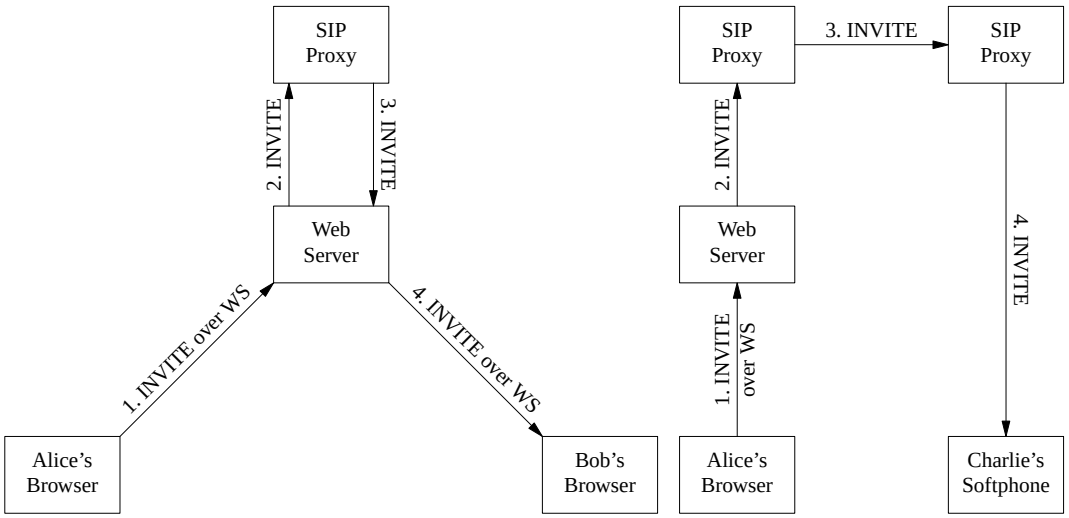
How good a point in the design space this is seems to largely depend on how hard it is to design the inline signaling protocol. If it's simple (e.g., we can pull something off the shelf) then this looks attractive. If it's complicated, then just using SIP starts to look attractive.

4.6 SIP in Browser

The final (and most complicated) case we consider is to embed a complete SIP softphone in the browser. This is the most complicated design but also the most functional, in at least two ways:

- It allows interoperability with any SIP endpoint and PSTN gateway, provided that they are willing to implement ICE, and potentially even if they do not.(Section 5.1).
- It is at least in principle compatible with "standard" `sip:` and `tel:` URIs for click-to-call.

The general design concept here would be have the site act as a captive SIP registrar/proxy with the client talking to it over WebSockets. In order to implement this design, the site would just stand up a standard SIP gateway and configure their WS server to proxy to it. Figure 1a shows the basic case where we have two browsers talking to the same site. Note that we just hairpin the SIP for call control. Figure 1b shows the case where Alice talks to Charlie's softphone. Note that the code on the web site is identical.



(a) SIP-based browser-to-browser call; same site (b) SIP-based browser-to-softphone call

Figure 1: SIP over WebSockets architecture

APIs

The natural APIs here just mirror typical SIP stack APIs, ported to JS and designed for greater ease of use. I won't go into these in any detail, but I think it's pretty obvious the style of thing you would want.

As with Section 4.5, it is possible to simply have an implementation with no APIs at all, i.e., just embed a SIP URI in HTML for click-to-call.

Standardization and Interoperability

As discussed above, this design requires the most standardization (in the limit a complete SIP API if you want portable apps!) but also offers the possibility of maximal interoperability. This is an improvement over Section 4.5 in that we can at least in theory have just URIs which allow direct interop with existing devices.

5 Semi-Orthogonal Issues

In this section we address a number of issues that are largely though not completely orthogonal to the high-level designs discussed above. (I.e., they are relevant with some but not all designs).

5.1 Interaction with RTP-Only Endpoints

Many endpoints do not do any form of connectivity check and just expect to receive raw RTP. Unfortunately, as discussed in Section 3.3, this is a security issue. There are a number of potential approaches for dealing with such endpoints.

Ignore Them

The simplest design is simply to abandon support for any such endpoint, requiring all endpoints to perform some connectivity check. This is clearly the most secure solution but also the least interoperable. The situation isn't as bad as it sounds because it's possible to stand up some kind of proxy, such as an SBC, which would respond to connectivity checks[8] but it's generally not great.

Whitelists

A step up is to have a whitelist of hosts which are willing to accept raw RTP. This whitelist could be embedded in browsers as shipped (and periodically updated).

Obviously, a whitelist design isn't practical for ordinary endpoints, but it is at least potentially useful for large RTP destinations (e.g., PSTN gateways) who are unwilling to change but who are readily identifiable. However, even then it would be very challenging: There are a large number of SBCs that would be need to be added, the list would be constantly changing and there is no financial incentive for the browser vendors to operate the complex service of adding and removing IPs to the white list. Finally, it is not clear that there is a good way for the browser vendor to verify claims that a given IP is owned by the requester and thus should be added to the white list.

RTCP

One possibility is to use RTCP as an implicit connectivity check:

1. The RTP transport can only be used with audio and specific codec, such as G.711 or ILBC.
2. The RTP transport can only be sent over UDP, to ports in the dynamic port range (no well-known ports)
3. If, after 10 seconds, no RTCP feedback is received that mirrors sequence numbers appropriately, media is terminated.

This is at least in some cases a reasonable defense against voice hammer attacks, and—if we do not permit any user control of the audio pipeline—against cross-protocol attacks.[9]. However, it suffers from two main drawbacks.

First, many endpoints (in particular many Cisco phones) do not support RTCP. Obviously, this mechanism does not work properly with any such endpoints, and will produce a really annoying user experience, since the user will just see a fast call drop after 10 or so seconds of audio. In order to assess the interoperability of this approach we would need to know the actual level of support for RTCP in the type of endpoints we care about interop with and moreover whether they send RTCP within a reasonable time window (which is not required by the standard).

Second, it's not clear if it's secure. An attacker who can inject plausible RTCP packets can simply fake this handshake. There are two defenses against RTCP forgery:

- Address filtering preventing the attacker from generating packets with the right IP address.
- Randomness in the SSRC and ISN.

[8] This idea suggested by Matthew Kaufman.

[9] If the user can inject audio packets of his choice, then he can embed traffic in the G.711 payload

There are a number of networks without adequate ingress/egress filtering, so the first defense is not enough. Together the SSRC and ISN have 48 bits of entropy, which seems like enough but it's not clear to me whether the (alleged) recipient of the traffic needs to know the SSRC. If he does, then we're down to 16 bits of entropy in which case there probably isn't enough entropy, since the sequence number can be anywhere within a permissible range.

Even if we ignore these issues: this design (and any design which allows transmission of uncomfirmed RTP), still poses some DoS threat. Even a low-bandwidth RTP-only stream provides something like 50 pps (8-16 kbps) from a single site. If a popular site such as Slashdot or Twitter embedded code to generate such stream, this could easily produce enough traffic to significantly degrade all but the highest bandwidth victim sites. It is of course possible for an attacker to mount this kind of attack via HTTP (e.g., with embedded IMG URLs) but the scale of amplification here is probably higher by about an order of magnitude.

6 Overall Thoughts

This paper has surveyed a number of potential designs. In this section we ask the question of whether there are some locally optimal points in the design space.

ICE vs. Just Connectivity Checks As discussed in Section 3.3, it is imperative that implementations do connectivity checks before sending media. Given the requirement for interoperation with existing equipment and the state of current NAT traversal technology, it is equally clear that those checks must be based on STUN. While it is possible to simply standardize STUN, previous attempts to do just this have met with failure, which is what lead the IETF to develop ICE. While it is true that ICE is complicated, and that there are settings where something simpler would be effective, attempts to develop a simpler generic solution have generally not been successful. Moreover, there already exist profiles of ICE (ICE-lite) specifically tuned for these simpler special cases, and these are not significantly more complicated to implement than just bare connectivity checks. Thus, it seems best to simply standardize on ICE.

This still leaves the question of whether ICE ought to be native in the browser or implemented separately in JavaScript. In principle, either is possible, but if ICE is to be the standard and thus every application will then need it, it seems natural to simply embed it in the browser with a standard API.

Extent of the Media Stack As argued in Section 4.3, the minimum acceptable media support seems to be a configurable media pipeline with support for a variety of codecs and explicit RTP support. For performance reasons this needs to be implemented in tht browser. While it may be desirable in the future to enable pipeline components to be implemented in JavaScript or some other downloadable language, this seems like a generic extension to the browser processing model—e.g., to support other performance-intensive browser applications—and is thus orthogonal to the arrangement of the media stack.

Signaling? The extent of signaling support remains the most difficult question. There seem to be three natural anchor points:

- No signaling at all

- Standardize media negotiation only
- Standardize on SIP

These are listed both in increasing order of capability and in increasing order of "power" exposed to the web application programmer. It is not really possible to distinguish between these purely on a technical level–the decision to large extent depends on the desired feature set we wish to expose to those programmers.