Architectural Framework for Browser-Based Real-Time Communications

Jonathan Rosenberg
Matthew Kaufman
Magnus Hiie
Francois Audet

Skype

September 21, 2010

Introduction

Real-time communications (RTC) remains one of the few – if only – classes of desktop applications that is not yet possible using the native capabilities of the web browser. These applications run natively on the desktop, or are powered by plugins. The functionality provided by these desktop clients is rich and complex – ranging from user interface, to real-time notifications, to call signaling and call processing, to instant messaging and presence, and of course – the real-time media stack itself, including codecs, transport, firewall and NAT traversal, security, and so on.

Given the breadth of functionality in today's desktop RTC clients, careful consideration needs to be paid to how that functionality manifests in the browser. What functionality lives within the browser itself? What functionality lives on top of it – either in client-side Javascript or within servers? What protocols are spoken by the browser itself? What protocols can be implemented within the Javascript? What protocols need to be standardized, and which do not?

Pictorially, the question is what protocols, APIs, and functionality reside within the box marked "Browser RTC Functionality" in Figure 1. Indeed, the central question is what functionality resides in that box, as the functionality will ultimately dictate the protocols that interface to it, and the APIs which control it.

The Media Component Model

It is our position that the functionality that manifests within the box be a *media component model*. In this model, the browser implements the necessary functionality to perform the real-time processing of media, starting from capture/render, through encapsulation in real-time transport protocols sent over the Internet. This functionality must be built into the browser, rather than within Javascript, due to its tight timing requirements and complexity. Furthermore, the functionality manifest as a set of loosely coupled components, each of which performs some aspect of the real-time processing. Each component has APIs which allow that component to be configured (with sensible defaults where appropriate), along with APIs that allow applications to gather information and statistics about the performance of that module.

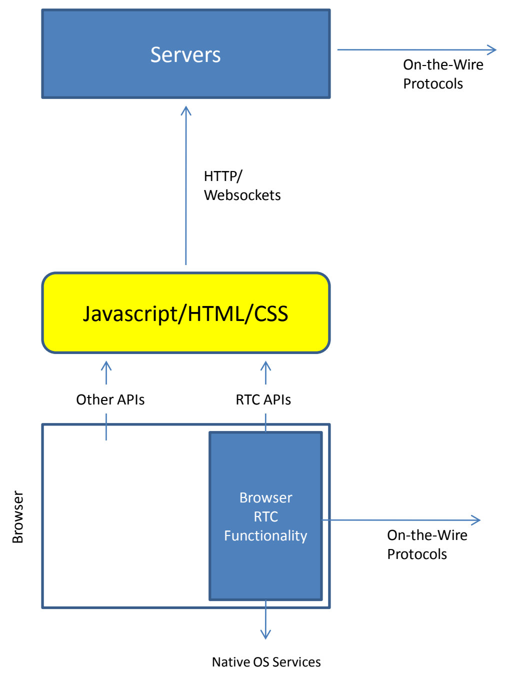

Figure 1: Browser RTC Model

The modules would include the codec itself, the acoustic echo canceller (AEC), the jitter buffer, audio and video pre-processing modules, and network transport components (including encryption and integrity protection of media) which speak specific transport protocols (such as the Real-Time Transport Protocol (RTP)).

The media component model is purposefully minimalistic. It opts for maximizing the functionality that lives outside of the browser itself – within Javascript or servers. In particular, only functionality which is real-time – which cannot be done using Javascript or server functionality – resides within the browser itself. As explained in [***Benefits of the Media Component Model***], this facilitates innovation, differentiation, and development velocity – all of the key characteristics that have made the web what it is.

As an example, a codec component implementing SILK [SILK ID] might be represented by a Javascript object with properties that mirror the configuration settings of the codec itself – the sample rate (one of narrowband, mediumband, wideband or super-wideband), the packet rate (number of frames per packet), the bitrate (which can vary between 6 and 40kbps), a slider that adjusts the packet loss resilience, a Boolean which indicates whether inband FEC should be used, and another Boolean which indicates whether to apply silence suppression. Of course, all of these parameters might have reasonable defaults so that non-expert programmers can just make it work. However, an advanced programmer could force a mode or change a setting as needed. After all, the SILK codec itself makes these parameters tunable exactly because there is no one right value; the correct setting depends on the application scenario and needs of the developer.

The Role of Signaling

It is our view that signaling is accomplished using a combination of existing client-server web protocols (HTTP, COMET, and websockets) and standards-based server-to-server protocols, such as SIP. A view of the “browser RTC Trapezoid” is shown in Figure 2.

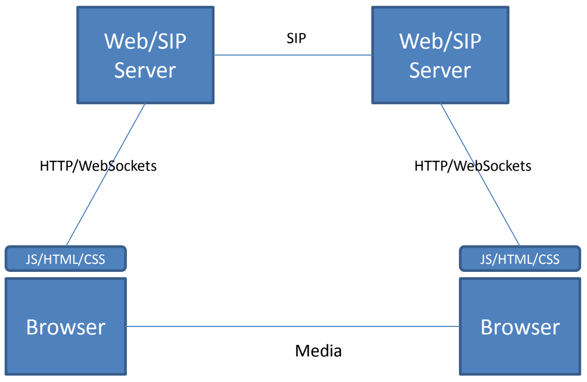

Figure 2: Browser RTC Trapezoid

In this example, a call is placed between two different providers. They use a SIP-based interface to federate between them. However, each of their respective browser-based clients signals to its server using proprietary application protocols built ontop of HTTP and Websockets. For example, provider A might offer simple calling services, and have a very simple web services interface for placing calls:

http://calling.providerA.com/call?target=joe@providerB.com&myIP=1.2.3.4:4476

Which takes only the called party and local IP/port as arguments. Provider A’s server infrastructure – some combination of web and SIP servers built in any way it likes – uses the identity of the target, along with previously-known information on the capabilities of the caller’s browser learned through a web-services registration, to generate a SIP INVITE. This arrives at provider B’s server infrastructure, which alerts its browser-based client of the incoming call. Provider B might be an enterprise service provider, and offer much richer features and signaling. Provider B uses a websocket interface to the browser, providing it the identity of the caller, the list of available codecs, and so on. B's service provider offers web-services based APIs for answering the call, declining it, sending to voicemail, redirecting to another number, parking it, and so on.

APIs within the browser allow each side to instruct the browsers to send media, including selection of media types and codecs.

In this model, there is no SIP in the browser. It *is our view that SIP has no place within the browser*.

SIP is an application protocol – providing call setup, registration, codec negotiation, chat and presence, amongst other features. For each and every new feature that is desired to run between a SIP client and a SIP server, a new standard must be defined and then implemented. The feature set is indeed vast, considering the wealth of potential endpoints, ranging from simple consumer "voice only" clients, to richer videophones, to voice and video multiparty conferencing (including content sharing), to low-end enterprise phones, to high end executive admin phones, to contact centers endpoints, and beyond. Each of those requires more and more SIP extensions in order to function. As an example, the BLISS working group in IETF was formed [BLISS charter] to tackle some basic business phone features – including line sharing, park, call queuing, and automated call handling. Each of these individual features requires one or more specifications, and needs to be designed to meet the needs of all of the participants in the process.

There are two important consequences of this. First, the requirement of standardization acts as a huge deterrent to innovation. Indeed, in many ways, it is anathema to the very notion of how the web is supposed to work. In the web model, the provider can define arbitrary content to render to users, craft arbitrary UI, and define arbitrary messaging from the browser back to the server, all without standardization or change to the web browser. Google does not need to wait for the browsers to implement IMAP in order to provide mail service. Facebook does not need the browser to have XMPP or SIP to enable presence and instant messaging. Why is call processing any different? Why should Skype or any other real-time communications provider be constrained by standardized application protocols? Each provider should be able to design and innovate what it needs, and not be constrained by the functionalities of the application protocols burned into the browser.

The second consequence is that interoperability will suffer dramatically. Interoperability between SIP clients and SIP servers is relatively poor; working only for basic call setup, teardown, and basic features. Important concepts like configuration remain poorly standardized and almost never interoperate. The web has certainly had interoperability problems, but nothing like those seen between SIP phones and servers, where in many cases features simply do not and cannot work. Interoperability is improved when there are fewer standards and not more. Instead of adding SIP and its myriad extensions to the browser (the current SIP hitchhikers guide [RFC5411] includes 140 references, most of which are SIP extensions) , application providers can use the tools that are already there – HTTP and websockets, and then define whatever signaling functions they desire ontop of that, without interoperability consequences.

SIP remains important as a glue between service providers, and between server infrastructure within provider networks. However, in a web context, there is simply no need for SIP support in the browser.

The Role of Media Transport

Unlike signaling, media transport does need to be in the browser, for two important reasons:

1. It operates in real-time and does not fit well with the programming model of Javascript
2. It needs to flow between endpoints directly – over UDP - in order to achieve low latency, and therefore requires standardization in order to interoperate with other providers or endpoints

The second point is important. Unlike most other web protocols, real-time media needs to be sent from the browser client to recipients other than the origin server or domain from which the web content came from. This is essential for ensuring low latency operations – one of the key metrics of quality in Voice over IP systems. In some cases, the recipient will be another browser endpoint from the same provider. However, it could be a desktop client or mobile client from the same provider, or as shown in Figure 2, it could be a browser endpoint or desktop endpoint from another service provider. In all cases, a direct connection – indeed a direct UDP connection - is important whenever possible.

From a security perspective however, the browser cannot just have an API that tells it to send arbitrary UDP datagrams or even standardized-format voice (or worse - video) media packets to an arbitary IP address. The former introduces the opportunity for malicious JavaScript to craft packets that mimick other application protocols and send them to arbitrary endpoints (for example, an enterprise SNMP server). The latter would introduce a substantial opportunity for denial-of-service attacks. Malicious Javascript could tell the browser to “spam” an unwitting recipient with high bandwidth video. In the voice literature, this is referred to as the voice hammer attack [ICE]. In existing voice systems, this attack is possible but not likely due to the closed nature of most of the software and systems. In a web environment, where all it takes is one line of malicious Javascript, the attack becomes almost a certainty.

To avoid this attack, a simple handshake can be utilized. The browser should support a simple STUN-based [RFC5389] connection handshake. The exchange of the STUN transaction ID prior to transmission of media prevents the attack.

Benefits of the Media Component Model

There are several important benefits of the media component model proposed here.

Enabling Innovation

One of the reasons why the Web has been successful as a user interface platform is the short turn-around time to deploy new versions of web-based services. Often, these new versions are experiments that vary small details which are important to make the service successful. It is the fine granularity of user interface elements in HTML and related technologies that allow this experimentation with details.

As there is no agreed-upon configuration of real-time audio/video communication technologies that always delivers the best result, we think that it is essential to give the application developers the same benefit of short turn-around time and ability to experiment with details. Therefore, the real-time communication primitives offered by user agents to web applications/services should be fine-grained enough to allow for enhanced configurations and possibly new scenarios. Also, these interfaces to the primitives should allow gathering real-world data in enough detail on how the primitives are operating, to enable the feedback loop of deploy-measure-reconfigure-redeploy.

One of the areas where perhaps the most innovation can be expected is signaling - one only needs to look at the plethora of standards around SIP. Proposing user-agent vendors to implement all these standards is a sure way to make the common denominator across user agents marginal. Instead, the browser already has a programmability model (JavaScript) that can handle all these use cases, and more, provided the programming environment has access to the underlying media components as we propose here. Drawing again parallels from user interface development, there is an undecided problem of what should be executed by the user agent, and what by the web servers (e.g. validation). Similar gray boundary between the client and the server exists in the field of real-time communications. Therefore we propose to leave standardization of signaling out of scope for this activity, and let the web service providers define signaling as they see fit.

The Importance of Flexibility

There are obviously tradeoffs between built-in functionality and programmability. It is often tempting to provide the web page author with a simple and relatively inflexible way of expressing their intent so as to minimize the page author's effort and accelerate adoption. As an example, the "<blink>" tag was adopted much more rapidly than it would have been if blinking text could have only been implemented by writing a JavaScript timer task to manipulate the DOM style objects.

On the other hand, such built-in functionality comes at two important costs. First, each browser implementation must implement the functionality, and the more which is moved from JavaScript to built-in functionality the more code must be present for that implemention. Second, and more important, the page author is now restricted to the subset of functionality which is provided by these browser implementations.

The "<video>" tag as it currently stands is an excellent example. While it does make it possible for a page author to embed video playback within a page without relying on external plug-ins (and without knowing much more than the URL of the video they wish to play), it also leaves the implementation of advanced functionality - such as adaptive multi-bitrate streaming - in the hands of the browser developer, not the page author. Unless all vendors agree on a standard for transmission of such videos (including things like the file format for manifest information), this advanced functionality will be not available across the browser landscape. Most importantly, the logic – the actual decisions about when to switch rates and why – becomes buried deep inside the browser, hard or potentially impossible to adjust for various circumstances.

An alternative approach for adaptive multi-bitrate video streaming was recently adopted by the Flash Player. The video object simply has an API for receiving bits to be played back. The script engine (and thus the script author, usually through the use of a pre-existing library) becomes responsible for determining which bits to download and which bits to pass to the video object. This enables adaptive multi-bitrate HTTP streaming video, but it also enables any number of other uses, many of which were not even contemplated by the providers of that API. It also means that upgrades to this logic come in the form of new script libraries, and not in the form of an upgrade to the Flash Player itself.

We advocate a similar approach here whenever it is possible. With the exception of the passing of real-time data to and from the media components (which we believe must communicate directly in order to meet real-time latency constraints) we advocate placing all of the logic outside of the browser itself and instead into the hands of the page author through JavaScript APIs. These APIs may be more complex to use for some cases, but they minimize the implementation effort on the part of the browser vendor and can provide functionality that has not yet been contemplated.

An example of this might be the peer-to-peer NAT traversal problem. Rather than having an API for "browser, please use ICE [RFC5245] to open a connection to another peer" we would instead have APIs like "browser, please send an ICE-compatible STUN [RFC5389] probe to the following candidate address". This allows the actual logic, the sequencing, the choice of what to implement at the client and what to offload to the server, to be in the hands of the JavaScript developer. We expect that libraries to implement common functionality (such as ICE, which could be built ontop of this) will become readily and freely available, and so in short order the extra work required for a page author to work with these lower level APIs becomes insignificant.

Interoperability with Existing VoIP Gear

In order for Browser-based Real-Time Communication to be successful, it is essential to ensure a good level of interoperability with existing VoIP gear. This means that a strong baseline for interoperability of end-to-end media needs to be defined.

The amount of VoIP gear currently deployed is very substantial for both VoIP Service Providers and Enterprise IP Telephony. In both cases, media is transported on RTP/RTCP [RFC3550] using codecs such as G.711 and G.729. Signaling for call control uses SIP [RFC3261], H.323, H.248/Megaco, and a wide range of proprietary protocols. Inter-domain, the signaling protocol is mostly SIP.

Interoperability at the signaling level can be handled by gateways, and is outside the scope of this paper. Media interoperability however needs to be addressed. It is not acceptable to rely on servers to convert media from one transport (and codec) to another because it introduces significant challenges. First, it requires a large number of servers to do the actual transcoding, which increases cost. Second, it affects the routing of media by adding an additional leg to the transport, which increases end-to-end delay, and therefore decreases voice quality. If there is codec translation, it decreases voice quality even further. And finally, it can potentially complicate end-to-end security.

Interoperability means working with reality, and not just standards. As such, it is important that browsers support basic RTP transport for voice and support the G.711 codec. Furthermore, they should interoperate with network-based session border controllers, which are the most commonly deployed technique for NAT traversal in existing networks. They should also support security, based on SRTP [RFC3711].

Proposed Standardization Process

In order to realize the vision of browser-based media, standards are needed to define the APIs, protocols, and layers of abstraction within the browser. This standards activity covers the areas of expertise and control of both IETF (which has expertise in real-time along with control over the standards it uses) and W3C (which has expertise in browsers and control over HTML and Javascript APIs).

As such, it is our proposal that the standards activity progress through a joint partnership between IETF and W3C, each of them handling different aspects of the problem. IETF should charter work to define the layer of abstraction and recommended information and controls that should be available to applications, and on-the-wire protocols that should be used. W3C would take this as input, and produce the actual specification for extensions to HTML and Javascript as needed.