# Real time communications in a web browser

Support for interactive real time communications, e.g. audio and video, in a web browser currently requires development and installation of plug-ins that implement the required functionality on behalf of the browser. While such plug-ins already exist for several browsers, they have limitations:

- They are specific to the browser and operating system and thus not universally available especially in environments such as mobile devices
- They are often bound to a specific service or application, i.e. can't be used by any developer wishing to utilize real time communications capabilities.
- As ad hoc extensions, they may lack support for instance for hardware acceleration, which is important for real time media. Extra processing causes extra power consumption, which is a major issue for portable devices.
- Installing them is always an extra step for the user.

The goal should be that browsers can natively support the capabilities that real time communications require, and that any developer can integrate real time communications in his/her web application with as little effort as possible. This would proliferate real time communications in the web for real, even in the long tail.

What should be done is to agree on and standardize the minimum necessary set of capabilities that every browser would need to support. Keeping the set at minimum would ease the effort of getting the browsers eventually compliant with it, and would also leave room for innovation. On the other hand, the minimum set should really contain everything that would allow building interoperability across browsers run by different operation systems. The minimum set should ensure that the basic services work, but could leave certain scalability and efficiency related optimizations as optional as long as forward-compatibility can be achieved.

The capabilities that "real time communications compliant" browsers would need to have range from standard protocols to standard APIs. There are already many widely used protocols for real time communications, mainly developed by the IETF and the XSF, that are also widely supported in different platforms. Reusing them instead of reinventing the wheel for the browser environment is important. The same applies for the APIs, they should utilize the work already done in W3C as far as possible.

## Types of applications to be supported

Before setting the requirements it is useful to look at what kind of applications are expected to use real time communications capabilities. In most cases the interactive voice and video communication would probably be an application internal feature, meaning that calls could be made only between the users of the same application. For instance a social networking site or a discussion forum could include a video call between the currently logged in users. In that case issues such as addressing would be also internal to the application.

There could also be applications that provide general calling capability to and from VoIP networks and even PSTN. For instance a VoIP provider could setup this type of web "portal" to

the service they are already offering. In these cases support for URIs, mapping to protocols such as SIP and XMPP, and inter-domain federation issues like global address space for the user identities would be important. Even some of the browser-to-browser calls would be inter-domain.

It would be useful to make both of these application types possible.

## Required capabilities in the browsers

The capabilities that browsers should have can be grouped in the following categories:

- Real-time transport and related NAT traversal

  The best approach would be if the two communicating endpoints (browsers) werebe able to establish an optimal path for the real time streams given the potential NATs and firewalls between them. If possible, the media streams should be sent directly end-to-end between the endpoints. For this it would be best to utilize well-tested and well supported protocols that many real time apps already use. These include RTP (and optionally RTCP) over UDP for carrying the media, and STUN/TURN/ICE for NAT traversal and connectivity checks. Potentially a stripped-down version of ICE might need to be defined for this purpose. In the worst case scenario some kind of HTTP tunneling for the media might be needed, but it is open if that would need to be standardized or not. The tricky part is that the use of ICE is not just something that the media transport does, but also poses requirements on the session setup signaling.

  For security, Secure RTP with “media path” key exchange would seem like the best fit.

- Connection maintenance between the browser and the server for asynchronous communication from the server towards the browser

  Real time communications applications need it since they communicate asynchronously with their server. Today most web applications resort to some form of HTTP long polling to mimic persistent two-way connectivity with the server. It is expected that in the future browsers will support the WebSocket protocol for this purpose – at least as long as it works through the legacy intermediaries. For real-time communications WebSocket over TLS seems like an ideal long term solution.

- Codecs

  Codecs for both audio/voice and video are needed. It would be useful to agree on a single mandatory-to-implement codec for each media (and perhaps “quality level”, such as a single narrowband and a single wideband voice codec), with the necessary profiling, to guarantee interoperability between different browsers. However, that may be difficult as there are so many to choose from. The emergence of widely accepted “royalty free” codecs may however change the situation. In any case, codecs should be negotiable, and such capability is a requirement for the session setup “signaling” protocol.

- Session setup “signaling”

This is clearly the most open part. One approach would be to leave it application specific. The application could do the session setup in the way it best sees fit, just utilizing the transport, codec and device capabilities. This would leave open how to implement certain end-to-end (browser-to-browser) functions such as codec and ICE/media endpoint candidate negotiation. These could still interoperate as long as both users are using the same application.

The Next level would be to define the session setup and ICE handshaking messages which the application level is assumed to transfer transparently between the peers, the messages could be presented (at application level) as base64 encoded strings, but have an internal well defined and extendable structure (e.g. XML based on XMPP/Jingle).

In order to make real time communication feature integration really simple for web application developers, the browsers should support a common session setup protocol. XMPP/Jingle over WebSocket over TLS would be one plausible approach.

- JavaScript APIs to the relevant resources

  The actual web application would need to use various resources through JavaScript APIs in a standardized manner. These APIs should include access to video/audio capture, video/audio playout and contacts. The W3C Device API Working Group API work in this area is relevant.

  All the beforementioned capabilities, such as codecs, media transport and session setup signaling should be exposed to the application through JS APIs. If the session setup protocol could be standardized, the application could be offered a simple-to-use high level API for session establishment. If the signaling protocol is left open, only lower level APIs could be offered.

## Special considerations for mobile devices

There are a few issues that are especially important when deploying browser based real time applications in a handheld size mobile device as well as for the low end notebooks..

- Connection maintenance with the server: Whether this is done with HTTP long polling, WebSocket or e.g. native XMPP, the issue is the same. In order to keep connectivity through NATs and firewalls (and in HTTP case also HTTP intermediaries), periodic keepalive traffic is required. Frequent keepalives are devastating to a mobile device battery when it uses cellular radio networks (2G, 3G, LTE, WiMAX). Thus, it would be essential that the keepalive scheme supports these two requirements:

  1. Client (browser) must be able control the rate and the exact timing of the keepalives. This makes it possible for the client to use potential external knowledge about the optimal keepalive rate and synchronize its keepalives with possible keepalives other applications are making.

2. The keepalive packets must be small.The rationale for this is that in the cellular networks there are different radio channel states and small packets can be delivered within states that use considerably less power than the ones required for larger packets. Typical size thresholds are around 100-200 bytes (including IP headers). Avoiding the more costly states during the keepalive cycle can have significant impact on the overall battery consumption in the device.

- Dealing with interface switching: Mobile devices often switch their Internet connectivity e.g. between cellular and WLAN interfaces. This is mostly an implementation issue, but the browser based application should be able to cope with such changes, for instance re-establishing its websocket or long poll connections and the logical session used over them to use the new IP address and interface.

- Avoiding real time media over TCP. This has abysmal performance over typical cellular access networks.

- Avoiding other TCP traffic in parallel to the real time media streams. Due to large buffers (up to several seconds) in typical cellular networks, having any significant TCP traffic, especially browsing that uses multiple TCP connections, in parallel to the real time media streams will make the real time streams to become unusable. Mechanisms such as IETF Ledbat work could be useful here to be used instead of normal loss driven TCP congestion control.