Evaluating two approaches for browser-based real-time multimedia communication

Marcin Davies
FTW Telecommunications
Research Center Vienna
Donau-City-Strasse 1
Vienna, Austria
davies@ftw.at

Joachim Zeiss
FTW Telecommunications
Research Center Vienna
Donau-City-Strasse 1
Vienna, Austria
zeiss@ftw.at

Rene Gabner
FTW Telecommunications
Research Center Vienna
Donau-City-Strasse 1
Vienna, Austria
gabner@ftw.at

ABSTRACT

This paper introduces a novel approach on how to integrate real-time multimedia communication into web applications. By using a Javascript API any web developer has the tools to use telecommunication functionalities in web apps. We aimed for an easy and efficient solution for developers and have performed an expert evaluation comparing our approach with the recent WebRTC/RTCWeb initiative driven by the W3C and the IETF. Our results indicate that our approach is well received among web developers and has advantages over similar solutions in terms of easiness, cross-domain availability, and compatibility with telco services.

Categories and Subject Descriptors

H.4.3 [**Communications Applications**]: Miscellaneous

General Terms

Design, Experimentation, Human Factors, Languages

Keywords

Telco, Web, WebRTC, RTCWeb, Browser, Javascript, Convergence, Browser-APIs, SIP, IMS, HTML5, Websockets, Real-time Communication

1. INTRODUCTION AND MOTIVATION

More and more innovative applications created for the Web are integrating typical telecommunication services — web and telco services are increasingly converging. Real-time multimedia communication is available for years in desktop applications like Skype, but will become more and more accessible directly through the browser without the need of additional software with new emerging standards such as HTML5 and WebRTC [3]. This will enable global and cross-domain communication possibilities and foster new and innovative Web applications and services.

MoMM2012, 3-5 December, 2012, Bali, Indonesia.

However, it is important to provide easy and efficient frameworks for developers to adopt these possibilities. In this paper, we are evaluating two frameworks for real-time browser-based communications: (1) a Javascript API and framework developed in the APSINT project and (2) the WebRTC project driven by the W3C and the IETF. Both frameworks offer APIs for real-time communication, but follow different design principles. In our evaluation we wanted to know how both API compare mainly in terms of efficiency and easiness. Furthermore, we were interested in possible improvements for the future.

The goal of the WebRTC initiative is to enable real-time communication in the Web browser natively. It is a set of protocols, codecs, and mechanisms that can be used (together with HTML5 extensions) to create multimedia applications with access to device capabilities from within the browser. With WebRTC one can establish a peer-to-peer connection between two web browsers connected to the same web server.

In the APSINT architecture we have followed a different approach, relying on the established SIP standard for communication between two parties, thus allowing interoperability and compatibility with classic Telco services and applications (e.g. breakout calls to any landline or mobile phone). At the same time, however, we were able to resolve the need for B2B relationships between operators and developers to access operator services. This is achieved by introducing the user as a man-in-the-middle between telco service and web page. While browsing a web site, pages rendered in the users browser will use communication facilities of the local device but which are programmed and controlled by Javascript code inside the web page. By this way the users B2C business relationship with the telco is acting on behalf of a B2B relationship between web application and operator.

This paper is organized as follows: Section 2 provides an overview on related work in this area, Section 3 describes our solution and architecture, Section 4 provides details on WebRTC, Section 5 discusses our evaluation, and, finally, Section 6 discusses the results and concludes the paper.

2. RELATED WORK

This section aims to give a brief overview on existing solutions to enable browser-based communication. There are two main approaches to realize real-time communication via the web browser. The first one (A) as depicted in Figure 1 takes advantage of remote communication services offered by 3rd parties via the web. Approach (B) utilizes commu-

nication capabilities available at the client device.

With method A, the developer of a web site uses a well defined Javascript library, i.e. API, to access server-side communication features (e.g. Tropo [11]). Real-time communication is initiated by sending a HTTP request to a web server, which establishes a network initiated call between the two users. Another approach to enable media handling in the browser is to use Adobe's generic Flash plugin to stream media directly to and from the browser. Flash is widely supported, however Adobe announced in a blog post [12] that they will discontinue the development of their mobile Flash plugin because of the increasing popularity of HTML5.

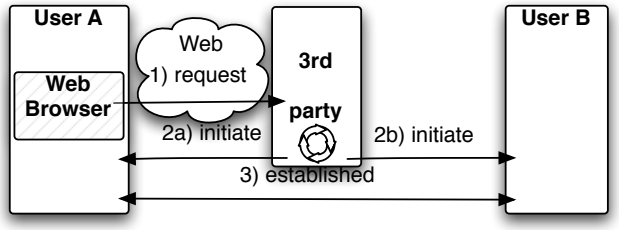

Figure 1: Communication initiated remotely (method A).

Figure 2: Communication initiated locally (method B).

A common solution to realize method B is to integrate local telephony features into the browser via plugins for already installed applications like Skype as shown in Figure 2. However, the main drawbacks for plugin-based solutions are: (i) only browsers with the plugin installed are supported, (ii) media (e.g. voice) cannot be handled by the browser directly, (iii) the communication software has to be installed at the client, and, finally, (iv) most plugins are not available for mobile browsers.

A hybrid solution of (A) and (B) is offered by Sipgate [10]. Sipgate is using a browser plugin to interface with their locally installed softphone, but also offers the integration of SIP based hardware phones. Thus by using the Sipgate plugin, it is possible to trigger calls either originated locally, or remotely at the 3rd party infrastructure at Sipgate.

Integration with the existing communication provider as for method (A) has the obvious advantage of an easy way to achieve terminal connectivity, quality of service, and interworking with other services to provide users with a mature solution. The obstacles of this approach are mainly because of the necessity to enter into a B2B relationship with every provider who would like to enable browser based real-time communication for his users. The goal of the APSINT project was exactly to remove this obstacle. The proposed solution is generic enough to be applied with different types of telecommunication architectures, e.g. VoIP. Special attention was paid to integration with the IMS architecture. IMS is the most advanced carrier-grade service delivery architecture which becomes the standard used by all mobile network operators.

Lately, a couple of initiatives pushed the standardization of browser based APIs to access local mobile device capabilities including real-time communication. A team from the Mozilla foundation started to work on their WebAPI [7], which allows access to telephony and messaging APIs via Javascript. Apart from that WebAPI also offers interfaces to battery status, contacts, camera, filesystem, accelerometer, and geo-location. WebAPI can control local communication, but audio is not handled in the browser.

Furthermore, Nishimura et al. [8] suggest a system that uses an architecture similar to APSINT (cf. Section 3). They also envisioned the possibility to deploy the software either locally on the client or remotely at a server. However their web to IMS integration is based on Flash plugins and relies on transcoding of media. Our solution presented in this paper does not need any modification or additional plugin in the browser and uses HTML5 instead. Also transcoding is not necessary in our approach.

Finally, one of the most recent initiatives is the already mentioned WebRTC [2] driven by the W3C and the IETF. We go more in depth on WebRTC in Section 4 and will compare it to our solution in Section 5.

3. APSINT ARCHITECTURE

This section describes the details of the APSINT architecture and the usage of the API. Section 3.1 gives an overview of the APSINT components and how they interact. User interface issues are discussed in Section 3.2. The detailed architecture and its representation in Javascript is explained in Section 3.3 and 3.4. The Javascript *Endpoint* and *Call* objects and their methods are discussed in more detail in Section 3.6 and 3.7.

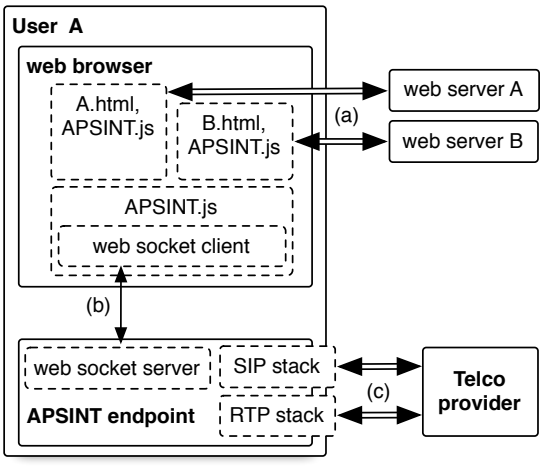

Figure 3: Overview of the APSINT Architecture

3.1 Software Framework

The interface (Figure 3 (b)) between browser and SIP stack as well as browser and media transmission is based on HTML5 web sockets asynchronously exchanging text based messages with a simple name value pair syntax . The browser part is implemented in Javascript and the native endpoint part is written in C using the sofia SIP library for Linux, OS-X, MeeGo and Maemo. On Android the Doubango software written in Java is being used. The architecture points out, or prepares for, security mechanisms to prevent malicious web pages from using communication resources. First, when starting the endpoint software, only the user can edit necessary data for registering the SIP/IMS client part to a SIP/IMS proxy. The SIP registration is completely decoupled from the browser software, hence web pages have no possibility to register themselves for communication services. Second, when a web page reserves and endpoint object, it might be asked for providing a credential string which may be a hash value over user id name and other user information using a common secret to create the hash value. If the APSINT browser software runs on a device with no native endpoint installed, the web socket connection towards the endpoint (Figure 3 (b)) will fail to open. Hence, reserving an endpoint will fail and the web page will know that it is not possible to use communication functionality. Errors in the SIP stack or media handling will result in an *ERROR* or *STATUS* message sent to the APSINT Javascript code in the browser. The Javascript APSINT objects will deal with these errors gracefully and inform the web page via callbacks, so that status changes and failed actions are reported to the web page and thus presented to the user. The web page may also implement its own error handling. Upon SIP registration/de-registration and IP connectivity loss/re-establishment, the endpoint native software will send a status message towards the browser, which is translated into triggering callback methods to inform the web page. All the different scenarios of registration/de-registration connection loss or re-establishment are translated into reporting an *ONLINE* or *OFFLINE* state towards the Javascript API.

3.2 User Interface Issues

In case the user closes the browser entirely, or the tab containing the Javascript code interfacing the endpoint, open call sessions will be closed and the endpoint connection between web page and native SIP stack is closed as well. In this case, the web sockets between native endpoint and APSINT Javascript code in the browser are closed, which results in a call tear down on the SIP native side and a call object destruction on the browser side. If the user hits the back/forward button and the page is being refreshed then all open call sessions are closed and the endpoint connection to the web page is closed in the same way as described above. Usage of multiple browser windows or tabs is also supported. A web page in a tab/window can reserve the endpoint incase no other web page has an ongoing communication session established (i.e. no other web page has a session that is not in idle state) For example in tab A web page A had reserved and endpoint and made and finished a call. Then in tab B web page B (or another instance of web page A) may reserve the endpoint during loading and use the endpoint while the page in tab A will lose the endpoint. Therefore, a web page should always (re-) reserve an endpoint object on page loading. If the browser runs in the background, the APSINT endpoint sessions (SIP sessions) remain active. If the browser crashes, all web sockets will be destroyed, which triggers closing of all SIP sessions on the native endpoint side. If the browser freezes then behavior depends on the type of the problem causing the crash. But by the time the browser is capable of closing the web sockets, or the browser is killed by the OS, all SIP sessions will be cleaned up by the native endpoint automatically. The existence of cashing proxies in between client and web page server will have no impact on the SIP session management, as connections are always controlled locally.

3.3 Endpoint Architecture

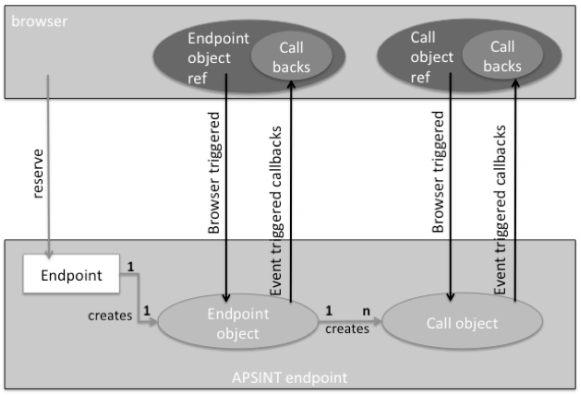

Figure 4: Basic Architecture of APSINT - Browser Endpoint Interaction

The basic technical challenge is the communication between browser and native call handling process. In direction browser to endpoint, simple local HTTP requests could be used but not in backwards direction when the call handling process talks back to the browser. The chosen talk back technology for the APSINT architecture is therefore the use of web sockets which allows a truly asynchronous communication towards and from web pages.

The web sockets protocol as proposed in [6] enables web browsers to establish a bidirectional channel to servers by upgrading a HTTP connection using an initial handshake. In APSINT the endpoint implements a web socket server, while the client side of web sockets is implemented by the web browser (as the web sockets protocol is part of the HTML5 standard all major browsers support that protocol). Hence the APSINT solution benefits from bidirectional connections and the low latency of the web sockets protocol while making browser specific plugins obsolete.

The basic architecture of the browser to endpoint interaction is shown in Figure 4, while the detailed message protocol between browser and endpoint is described in [13]. It should be noted that reserving a call at the endpoint Javascript object is not to be mixed up with registering at a SIP proxy. The *reserve* method call here means: Give me an object reference to the endpoint and reserve its services for me exclusively. On reservation, the endpoint will open a web socket connection between the APSINT Javascript library and the native SIP management of the endpoint in a process separated from the browser.

3.4 The Javascript API

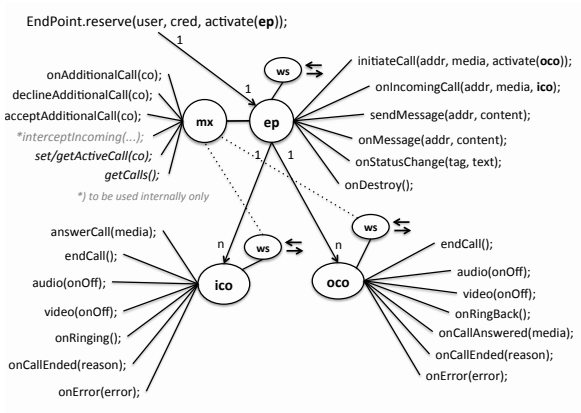

Figure 5: APSINT Endpoint Object Relationships

The following objects can be identified (only the methods relevant for usage by the web page are depicted in Figure 5):

ep: the endpoint object; an instance that is returned by the endpoint factory function *Endpoint.reserve*. Every initial/first incoming message of a new call is directed towards this object and every new call needs to be established via this object. Also plain non session related message sending between two parties is done via this object (i.e. sending a SIP MESSAGE message).

ico: the incoming call object is passed along with the *onIncoming* callback of the endpoint object, hence it is the object reference of a new incoming call. It has all the methods and callbacks necessary to deal with this new call and only this call.

oco: the outgoing call object is passed in to the activate callback provided for the *initiateCall* method of the endpoint object, hence it is the object referencing a newly created call. It has all the methods and callbacks necessary to deal with this new call and only this call.

mx: this stands for 'mixer' and denotes the coordination function of the APSINT functionality. It contains all necessary functions and callbacks to coordinate more than one call represented by their call objects (*ico*'s and/or *oco*'s). The mixer is listening to the web socket communication of each call object so that it may automatically coordinate and adapt call configurations. A default method can be overridden by the web page to deal with incoming add-on calls automatically in the desired way.

ws: this is the web socket that communicates between the APSINT native communication routines running in a separate process and the call objects or the endpoint object respectively. Both, method calls of the web page towards the APSINT endpoint and invocation of callbacks by the endpoint towards web page are mediated to sending or receiving text messages over these web sockets. Each object *ep*, *ico*, or *oco* has its own web socket connection. The number of open web sockets is determined by the browser or server system and resource constraints. If the user leaves the page, for what ever reason (e.g. pressing the back button), the web socket gets closed.

A web page (wp) holds only one ep object at a time but may hold many ico and oco objects. An ep object holds only one mx object. The object relationships in detail are: wp:*ep* $\widehat{=}$ 1:1; *ep*:*mx* $\widehat{=}$ 1:1; *ep*:*ico* $\widehat{=}$ 1:n; *ep*:*oco* $\widehat{=}$ 1:n; *ep*:*ws* $\widehat{=}$ 1:1; *ico*:*ws* $\widehat{=}$ 1:1; *oco*:*ws* $\widehat{=}$ 1:1.

3.5 Javascript Endpoint Factory

Calling the factory function is the very first thing a web page would do if it requires the APSINT endpoint functionality. It would do so by calling the static method *reserve* on the *Endpoint* class. Please refer to the description of this method below:

- *Endpoint.reserve(user, credentials, activate, ws_url)*
  - user, a String which may contain the url of the VTS, the call object returned will be assigned against this id, which could be the domain of the site. Any other meaningful value is possible to be set by the web page.
  - credentials (optional) if the APSINT endpoint would ask for authentication
  - the reserve function may be called by a web page at any point in time but usually on the onLoad event of the web page, the function returns a valid reference to an object (stored here in a variable called co) which can be used for further usage of the endpoint call object.
  - the namespace "Endpoint" here denotes the name of the call objects class where "reserve" is a static method of class "Endpoint"
  - the Endpoint class will call the "activate" callback, once the connection to the native endpoint via web sockets is established. It passes in the call object to be used by the web page. It can be used to initiate calls and override callbacks (i.e. listen to call events)
- *Endpoint.reserveSimple(activate)* This factory method offers the same functionality as the reserve method described above, but puts default values for user id, credentials and web socket URL (ws_url). The default address is: ws://localhost:1234.

The following code snippet shows a minimum example of how to obtain and set an endpoint object:

```
var callobject=null;
Endpoint.reserve("ftw.at/apsint", "",
  function(co) {callobject = co;},
  "ws://127.0.0.1:4567");
callobject.onRinging = function() { ... };
callobject.initiateCall(zeiss@ftw.at, "AV", newcall);
```

There are no means of freeing an endpoint object, this is done automatically by the native endpoint software, however the web page will be informed when an endpoint object is freed by the APSINT library via calling the onDestroy() method on the endpoint object. If the web page is interested in being informed about call object or endpoint destruction, it should override this method.

The rules for successful reservation of an endpoint instance are:

- If no other webpage has reserved the endpoint then the actual requesting webpage can grab the endpoint

- If some other webpage has reserved the endpoint but is no longer present (i.e. the page forgot to free the endpoint or crashed before freeing) then the actual requesting webpage can grab the endpoint. The endpoint running in a separate process will detect that the web page that has reserved it is no longer present by loss of web socket connection.

- If some other web page has reserved the endpoint but nothing is to be done – i.e. no open communication session – another webpage may request the endpoint reservation. The old webpage is informed by the endpoint if it loses the reservation in favor of another web page. It will - if possible - regain the reservation next time it tries to use the endpoint actively. The *onDestroy* callback is invoked by the endpoint to communicate reservation loss.

- If some other web page has reserved the endpoint and has a communication session running the reservation request will be denied. This is indicated by the reservation routine in returning a null pointer.

3.6 Description of the Endpoint Object (ep)

The following methods are available to the web page:

- initiateCall(address, media, activate); Will start a new call towards the specified *address* with the given *media* for communication; the object for call handling is passed on to the *activated* callback function. Parameters are: *address*: the address of the B-party in string format, could be any information URL or number, but in our case it will contain the SIP URI; *media*: the desired media for the communication, possible values are *A* for audio, or *AV* for audio and video; *activate*: takes the function name of a callback function that will be called by the endpoint object when the call object for the new call is ready to use, the reference of this call object is passed in to the callback (activate(callobject)); the caller of the *initiateCall* method, i.e. the web page, needs to provide this callback in order to obtain the callobject for call handling of the new call

- sendMessage(address, content); Will send a (text-) message to the specified *address*, the message content is specified in *content*. Parameters are: *address*: the address of the B-party in string format, could be any information URL or Number, but in our case it will contain the SIP URI; *content*: a string representing the message content to be delivered, may be human readable string or JSON or other. Interpretation of the content is left to the client software (e.g. the web page or a plain SIP client)

The following callback methods should be overridden by the web page if it is interested in getting notified about the related call events:

- onIncomingCall(address, media, callobject); Called by the endpoint to notify on incoming call from *address*, requesting communication media *media*, handled by object instance *callobject*. Parameters are: *address*: the address of the B-party in string format, could be any information URL or number, but in our case it will contain the SIP URI; *media*: the desired media for the communication, possible values are *A* or *AV* for audio and video; *callobject*: takes the function name of a callback function that will be called by the endpoint object when the call object for the new call is ready to use, the reference of this call object is passed in to the callback (activate(callobject)); the caller of the initiateCall method, i.e. the web page, needs to provide this callback in order to obtain the callobject for call handling of the new call

- onMessage(address, content); Called by the endpoint when a (text-) message was received. Parameters are: *address*: the address in string format of the messages originator, could be any information URL or number, but in our case it will contain a SIP URI; *content*: a string representing the content of the message that has been received, it may be human readable string or JSON or other. Interpretation of the content is left to the client software (e.g. the web page or a plain SIP client)

- onStatusChange(tag, text); Parameters are: *tag*: The short name of the status change; *text*: additional information or a longer human readable description of the status (this one may be used to be directly displayed on the web page)

- onDestroy(); Called by the endpoint to inform the web page that the endpoint it is currently using is about to be destroyed; the web page may no longer use the object, the web socket towards the native endpoint software is closed already. The web page may set all its references towards the objects to null to allow for garbage collection.

- onVideoData(id, videoData): This method should overridden by the web page only if really needed. The current functionality does the following: Every time it is called it replaces the *src* source data of an image tag with *id="remotevideo"*, e.g. <img id="remotevideo" src="data:image/jpg;base64">. The image as a base64 encoded string received by the native endpoint. This way the video frames received on the SIP stack are transported to the browser. As pictures are replaced fast the image becomes a video (sort of flip-book mechanism)

3.7 Description of Call Objects (ico and oco)

The following methods are available to the web page (for distinction of which method is linked to what type of object, *ico* or *oco*, please refer to figure 5):

- endCall(); Invocation of this method will end the call. The method may be called regardless of call state, meaning it may be called while talking, while the B-party is waiting for an answer or because of any other reason. Calling this method while no call is in progress will have no effect at all.

- audio(onOff); Invocation of this method will turn on or off audio output, i.e. the microphone will be turned "mute". The audio stream towards the B-party will be handled by re-invites. The method can be called at any time but the current value of audio output will take only effect during call. The audio output status is

also stored in the audio_out property of the call object. Parameters are: *onOff* takes a Boolean value *true* to turn audio on or *false* to turn audio off - NOTE: This method is implemented by the Javascript code but currently not in the native endpoint.

- video(onOff); Invocation of this method will turn on or off video output, i.e. the camera will be turned off, it will no longer record the user. The video stream towards the B-party will be handled by re-invites. The method can be called at any time but the current value of video output will take only effect during call. The video output status is also stored in the video_out property of the call object. Parameters are: *onOff*: takes a Boolean value *true* to turn video on or *false* to turn video off - NOTE: This method is implemented by the Javascript code but currently not in the native endpoint.

- answerCall(media); This method needs to be invoked, if the user wants to answer an incoming call. At this point in time the web page may also indicate the *media* it accepts for the communication in case it differs from the offered media by the B-party, received in the *onIncomingCall* callback. Communication media channels will be opened immediately. Parameters are: *media*: the communication media allowed by the user (web page), passing this parameter may be left out (*undefined*) or set to *null*, meaning that communication media are accepted as suggested by the B-party. Currently supported values are *A* for audio calls or *AV* for audio and video calls.

The following callbacks may be called by the endpoint software:

- onCallEnded(reason); This method may be called by the APSINT endpoint at any time during call or call setup. Optionally a reason for the end of the call may be given, i.e. *DECLINED*, *ONHOOK*, or *ERROR*. Parameters are: *reason*: the reason for ending the call by endpoint or B-party, may also be *undefined* or *null*.

- onError(error); This method may be called by the APSINT endpoint at any time during call or outside a call. Optionally a human readable description of the error may be passed. Parameters are: *error*: a human readable description of the error, may also be *undefined* or *null*.

- onRinging(); This method is called by the endpoint in case a incoming call request has been received and the local client has sent a ringing information towards the call initiating B-party. This is for the web page to indicate that the calling party has been signaled that this call may be answered (sort of "the line is free").

4. WEBRTC

The goal of the WebRTC/RTCWeb initiative is to enable real-time communication in the web browser natively. The standard is driven by the W3C(WebRTC)[2] and the IETF(RTCWeb)[1] and is a set of protocols including, amongst others, codecs and mechanisms for media transport and signaling, security mechanisms, and NAT traversal. Together with HTML5 extensions to access the local microphone and video camera using *getUserMedia*[5] and the W3C WebAPI, developers can create multimedia applications with access to device capabilities just from within the browser. As depicted in Figure 6, with WebRTC one can establish a peer-to-peer connection between two web browsers connected to the same web server. As the web server manages sessions for both browser A and B, it can easily exchange IP addresses and ports of the two parties and help to establish the peer-to-peer session. In case the browsers are in local networks (NAT), or behind a firewall, WebRTC utilizes well known and established NAT traversal mechanisms like STUN/ICE or TURN.

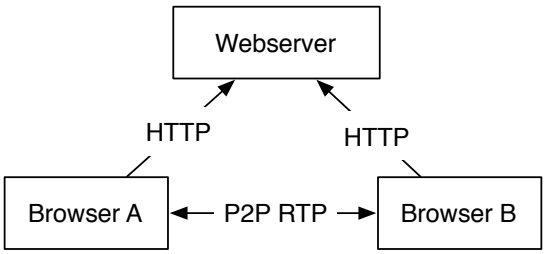

Figure 6: Basic Architecture of WebRTC

4.1 Signaling and Identity

To setup a media connection between two web browsers it is necessary to exchange and negotiate parameters for such a stream. WebRTC does not specify the signaling methods nor the protocols to transport the signaling data: "Communications are coordinated via a signaling channel which is provided by unspecified means, but generally by a script in the page via the server, e.g. using *XMLHttpRequest*."[3] However it specifies the way for session negotiation, namely the RTCWeb Offer/Answer Protocol (ROAP) which is based on the Session Description Protocol (SDP) specified in RFC 3264. It's an offer/answer protocol, which means that the initiator sends an offer including codec, IP address and port to the called party. The called party replies with an answer which also includes the needed data to setup the media link between the two. As the coordination is usually handled by the server of the web page, there is no standard for global identities yet (like in SIP[9]), and therefore there is no standardized way to establish a multimedia session between two users who surf on different web sites.

4.2 Media Transport

In general, real-time media data is transferred according to the RTP/RTCP protocol. WebRTC specifies its own version of RTP in an IETF draft[4]. The proposed WebRTC RTP standard requires a more strict implementation of specific features sometimes omitted in original RTP implementations.

4.3 Security and Privacy

Solutions for WebRTC security are still under discussion. There are some guidelines for example that all media connections should be secured via SRTP and the signaling (e.g. ROAP) should be transmitted using HTTPS. In respect to privacy, the web browser, as the "trusted computing base" should take care and ensure the user's privacy. E.g. if the web site calls the *getUserMedia*[5] API, the browser will ask

the user for consent. This could be achieved with pre-defined policies or just with pop-ups at run-time.

4.4 Summary

WebRTC is still not fully standardized and there are many different browser implementations and versions of WebRTC web sites. In addition and at the current time of writing, WebRTC is not enabled by default, therefore requiring users to manually enable certain browser features to use WebRTC.

5. EVALUATION AND COMPARISON

In this section we present our evaluation of our APSINT Javascript API and the WebRTC API. Section 5.1 introduces the method and the test setup, while Section 5.2 presents the results of the evaluation.

5.1 Method and Setup

We have recruited three programmers for a 2-day in-house expert evaluation of both APIs. Two of the programmers had a high experience in web technologies (HTML/JS/CSS) and were 21 and 22 years old (male). The third developer was 36 in age (also male) and had a medium experience level with web programming. The candidates had no or only little experience with telco protocols or services (like SIP, POTS, ISDN, ParlayX, etc.). None had prior know-how with WebRTC.

We have provided the candidates with the APSINT API, software, and documentation as well as links to WebRTC software and documentation approximately 10 days before the evaluation started. None of the programmers performed significant preparation work with WebRTC. The same was true with APSINT except one of the users indicated that he prepared himself more intensively with the provided materials.

Each day of the evaluation was dedicated to one API. Starting with APSINT on day one, we have followed a use-case based approach. Our programmers were given the task to develop a web-based and small-scale version of Skype. In its simplest form this is a web page with a list of buddies that can be contacted. We wanted to see if this task could be realized within one day of working with the respective API and how far developers could get in implementing functionality.

After a short briefing our users had two slots of 2.5 hours for working on the use case with a break of one hour in between. During the evaluation we were present to answer questions and observe the progress. We took notes of evident problems and opinions of the users. At the end of day we interviewed each participant separately with a short questionnaire. After collecting some demographic data (see above), we asked 14 open questions to receive qualitative data about the API. Seven questions followed to gather MOS (mean opinion scores) for specific aspects of the API. Finally, we made note about the progress of each developer, i.e. how much of the task could be fulfilled, recorded lines of code and collected the source code for further analysis.

5.2 Results

In general, our developers were able to develop much of the functionality of our Skype use case on day one with our APSINT API. All three were able to implement chatting between users, two had working audio connections between two parties, and one of our users even implemented online/offline status within the buddy list.

The picture was a bit different with WebRTC on day 2, however. Only one developer managed to implement chat functionality, and two were able to establish audio/video connections. No one could develop a buddy list within the time available.

In the next sections we go into detail on the qualitative and the quantitative results of our evaluation.

5.2.1 Qualitative Analysis

We have collected some general feedback about each API with the following two questions:

- In general, how was the handling of the API?
- Which knowledge and skills were necessary to work with the API?

Table 1 summarizes the answers our developers have given (same answers are marked in brackets with the number of repetitions)

	APSINT	WebRTC
Handling	easy (2), events and hooks positive (2)	no overview, unclear, difficult to find relevant pointers, examples did not work, steep learning curve
Necessary knowledge	Basic to intermediate Javascript skills (callbacks, overwrite methods), client/server principles	deep Javascript knowledge (2), server-side web programming (3), protocols (SDP), Turn/Stun server, session management

Table 1: Qualitative Analysis – General Feedback

After that we were curious about the positive aspects of the APIs. The questions were as follows:

- What was easy in implementing the use case?
- What are the advantages of the API?
- What did you particularly like about the API?

Table 2 summarizes the answers our developers have given (same answers are marked in brackets with the number of repetitions)

Finally we wanted to gather the negative aspects of the APIs. The questions were as follows:

- What was difficult in implementing the use case?
- What were frustrating moments?
- What are the drawbacks of the API?
- What could be improved?
- What was very distracting?

Table 3 summarizes the answers our developers have given (same answers are marked in brackets with the number of repetitions)

	APSINT	WebRTC
Easy aspects	Message sending (2), reserve endpoint, call methods, establish connections	local media and video (3)
Advantages and Likings	simplicity (2), abstraction of Telco features, flexibility, openness, possibilities, convenience, lightness	efficiency, open standards, features, video effects, browser support, more general concept, no dependency on SIP

Table 2: Qualitative Analysis – Advantages, Positive Aspects

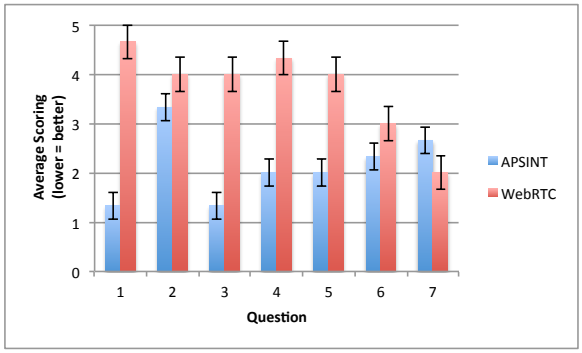

Figure 7: Quantitative Evaluation Results (Averages with Standard Errors)

5.2.2 *Quantitative Analysis*

In addition to the qualitative questions we have also gathered quantitative data during our evaluation. Table 4 shows the questions asked, while Figure 7 summarizes the results received (average scorings with standard error as error bars).

Regarding comprehensibility (Question 1), experience level needed in server-side web programming (Question 3), easiness of creating a new or integrating into an existing web site (questions 4 and 5), APSINT scored significantly better, while there was no statistically significant difference for questions 2, 6, and 7.

6. DISCUSSION AND CONCLUSIONS

In this paper we have presented our solution for browser-based real-time multimedia communication and compared it against the new upcoming WebRTC initiative. Although the number of participants in our study was low and quantitative results may not be statistically significant, our qualitative analysis could clearly reveal drawbacks and advantages of both APIs.

At the current stage, WebRTC has a steep learning curve and lacks proper documentation. It was difficult for our developers to implement our rather simple use case. In contrast, our APSINT approach was regarded as easy to adopt, simple and lightweight – the main drawbacks were rather minor problems with callback information and hooks. Consequently, the evaluation strengthens our assumption that our APSINT API is capable of providing an effective way to integrate real-time communication into web browsers and bridging the gap between web and telco services.

	APSINT	WebRTC
Difficulties and frustrating moments	Call establishment, onDestroy, onCallEnded hooks not working, SIP address bug, fixed to certain browser versions	Understanding API, getting overview, concept of PeerConnection, establishing remote connections (2), complicated signaling, without the use of node.js very difficult
Drawbacks and distractions	Callbacks do not carry enough information, not failsafe, static endpoint object vs. dynamically generated objects	no real Javascript API, missing documentation, no easy way to send a message, lots of things to care about
Proposed Improvements	Offer javadoc-style documentation, consistency of documentation, some properties not documented, endpoint should run remotely, more information provided by callbacks, easy way to query endpoint status, getter methods	documentation (3), examples, guidelines, tutorials, more high-level functions

Table 3: Qualitative Analysis – Disadvantages, Negative Aspects

At the same time, we believe that WebRTC will for sure gain momentum as more and better documentation will follow, standardization will progress, and browser support will be extended. With the use of open standards, its efficient media transmission, and no need of plugins or additional software WebRTC has a lot of potential. We are aware that our solution needs a software component (the endpoint) on the client side and is dependent on SIP, which might be seen as a major drawback compared to WebRTC, but given the easy adoption by developers and the advantages of e.g. call handling without an open web browser, the broad compatibility with any SIP-enabled client, cross-domain and mobile browser support, we believe that our solution is a viable alternative to and even more preferable than WebRTC.

Finally, future work will be offering a Javadoc-style documentation for our API, improved documentation, and maybe a second trial with a larger set of developers.

7. ACKNOWLEDGMENTS

The Competence Center FTW Forschungszentrum Telekommunikation Wien GmbH is funded within the program COMET - Competence Centers for Excellent Technologies by BMVIT, BMWA, and the City of Vienna. The COMET program is managed by the FFG.

We would like to thank the APSINT project team and especially our colleague Goran Lazendic for his contributions

1	In general, how understandable is the API? (1 – understandable, 5 – not understandable)
2	Which level of experience in client-side web programming is needed to get productive? (1 – low level, 5 – high level)
3	Which level of experience in server-side web programming is needed to get productive? (1 – low level, 5 – high level)
4	How easy it is to develop a (multi-user capable - that does not mean support for multiple sessions on the client) communication web site from scratch? (1 – easy, 5 – difficult)
5	How easy it is to integrate communication facilities into an existing web site? (1 – easy, 5 – difficult)
6	In general, how do you perceive the possibilities (in terms of feature richness) provided by the API? (1 – very rich, 5 – very limited)
7	How do you perceive the customization possibilities of the API (e.g. customizing the call flow, etc.) ? (1 – very rich, 5 – very limited)

Table 4: Quantitative Analysis – Questions

to this work.

8. REFERENCES

[1] RTCWeb Status Pages. `http://tools.ietf.org/wg/rtcweb/`, July 2012.

[2] WebRTC Webpage. `http://www.webrtc.org/`, July 2012.

[3] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan. WebRTC 1.0: Real-time Communication Between Browsers. `http://dev.w3.org/2011/webrtc/editor/webrtc.html`, July 2012.

[4] J. O. C. Perkins, M. Westerlund. Web Real-Time Communication (WebRTC): Media Transport and Use of RTP. `http://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage-04`, July 2012.

[5] A. N. Daniel C. Burnett. Media Capture and Streams. `http://dev.w3.org/2011/webrtc/editor/getusermedia.html`, June 2012.

[6] I. Fette and A. Melnikov. The WebSocket protocol. `http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-17`, November 2011.

[7] Mozilla.org. WebAPI is an effort by Mozilla to bridge together the gap, and have consistent APIs that will work in all web browsers, no matter the operating system. `https://wiki.mozilla.org/WebAPI`, 2011.

[8] H. Nishimura, H. Ohnimushi, and M. Hirano. Architecture for Web-IMS Cooperative Services for Web Terminals. In *Intelligence in Next Generation Networks, 2009. ICIN 2009. 13th International Conference on*, ICIN 2009, New York, NY, USA, 2009. IEEE.

[9] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. `http://www.ietf.org/rfc/rfc3261.txt`, June 2002.

[10] sipgate.com. Move your phones to the cloud. `http://sipgate.com`, 2011.

[11] tropo.com. Tropo - Cloud API for Voice, SMS, and Instant Messaging Services. `https://www.tropo.com`, 2011.

[12] D. Winokur. Flash to Focus on PC Browsing and Mobile Apps; Adobe to More Aggressively Contribute to HTML5. `http://blogs.adobe.com/flashplatform/2011/11/flash-to-focus-on-pc-browsing-and-mobile-apps-adobe-to-more-aggressively-contribute-to-html5.html`, 2011.

[13] J. Zeiß, M. Davies, G. Lazendic, R. Gabner, and J. Bartecki. Integrating communication services into mobile browsers. In K.-H. Krempels and J. Cordeiro, editors, *WEBIST*, pages 753–762. SciTePress, 2012.