INTEGRATING COMMUNICATION SERVICES INTO MOBILE BROWSERS

Joachim Zeiß[1], Marcin Davies[1], Goran Lazendic[1], Rene Gabner[1] and Janusz Bartecki[2]
[1]*FTW Telecommunications Research Center Vienna, Vienna, Austria*
[2]*Kapsch CarrierCom, Vienna, Austria*

Keywords: Convergence, VoIP, Browser-APIs, SIP, IMS, HTML5, Websockets, Real-time Communication.

Abstract: This paper introduces a novel approach on how to integrate communication services into Web applications running in the browser. The solution is based on two major design decisions: To resolve the need for a business-to-business (B2B) relationship between Web provider and communication service provider, and to distribute the Model, View and Controller components of an application across different processes. Our approach helps to answer the question on how to efficiently integrate network operator's assets into applications from over the top (OTT) players. The separation between application control by the Web page and the actual command execution by the native capabilities of the user device opens new opportunities for global reachability of telco services, easy deployment and re-deployment of applications with zero configuration need for users and developers as well as privacy protection by keeping sensitive data within the user domain, e.g. the user's communication device.

1 INTRODUCTION

More and more innovative applications created for the Web are integrating typical telco services. Users in turn, get accustomed to the business model of the Web and perceive telecommunication services offered outside the web context as reliable and of high quality although being a bit old fashioned, detached from the social web community and too technical. Application developers concentrate on globally marketable products with simple and unified interfaces. Telcos, even when operating globally, serve a smaller community compared to Google, Apple, Facebook or other over the top service providers. They struggle to unify their activities to participate in the application business and to avoid becoming only bit pipes.

Therefore, the following question needs to be answered: How can telco assets be efficiently and commercially feasible integrated in applications from over the top (OTT) players? Or, to put it down in a more provocative statement: OTT players providing Web applications use other OTT player communication technologies to accomplish their services. How can operators achieve that Web applications from OTT players and content providers preferably use their communication services? In order to make this possible the "Advanced Prosumer Service Integration Intelligence" project, called APSINT, provides a software architecture that integrates seamlessly into the mobile operators network infrastructure.

Telco operators do a good job in reliability, quality of service, network convergence and interoperability when it comes to connecting people by text, voice and video. On the other hand operators lack in offering their services globally and easy to be used by Web developers and in delivering simple yet powerful human interfaces to end-users. The APSINT architecture resolves the need for B2B relationships between operator and application providers and developers for them to use operators services. This is done 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 within the Web page. By this way the users B2C business relationship to the telco is acting on behalf of a B2B relationship between Web application and operator. This separation between control (by the Web page) and actual execution (on the user's device) has the advantages of (i) global reachability of telco services, (ii) easy deployment, (iii) zero configuration need for user and developer as well as (iv) privacy protection as there is no need for the user to share authentica-

Zeiß J., Davies M., Lazendic G., Gabner R. and Bartecki J..
INTEGRATING COMMUNICATION SERVICES INTO MOBILE BROWSERS.
DOI: 10.5220/0003934907530762
In *Proceedings of the 8th International Conference on Web Information Systems and Technologies* (WEBIST-2012), pages 753-762
ISBN: 978-989-8565-08-2

tion credentials with Web pages for 3rd party service usage.

The remaining paper is organized as follows: Section 2 provides an overview over related work in this area, Section 3 describes our solution and architecture, Section 4 provides implementation details, Section 5 discusses our outcome and experiences made while evaluating the prototype and finally Section 6 gives an outlook on further work.

2 RELATED WORK

This section aims to introduce and compare existing solutions to enable mobile browser-based communication. There are two main approaches to realize real-time communication via the Web browser. The first one (A) takes advantage of remote communication services offered by 3rd parties via the Web. In this case, as mentioned in Section 1, a B2B relationship is needed between the developer of the Web site and the telco. Approach (B) utilizes communication capabilities available at the client device (e.g. smart phone).

As depicted in Figure 1, method (A), the developer of a Web site uses a well defined Javascript library to access server side communication features (e.g Tropo (Tropo.com, 2011)). Real-time communication is initiated by sending an HTTP request to the 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, which is more flexible as almost every browser is Flash enabled. This way it is possible to stream media directly to and from the browser. However Adobe announced in a blog post (Winokur, 2011) that they will discontinue the development of their mobile Flash plugin because of the increasing popularity of HTML5.

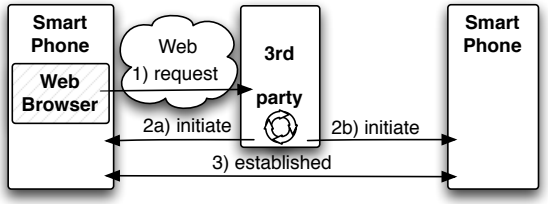

Figure 1: Communication initiated remotely (method A).

Looking at method (B) as shown in Figure 2, a common solution to integrate local telephony features into the browser is via plugins for already installed applications like Skype. This way the user can access the locally installed application via the Web

Figure 2: Communication initiated locally (method B).

browser. 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 (iv) most plugins are not available for mobile browsers.

A hybrid solution of (A) and (B) is offered by Sipgate (Sipgate.com, 2011). Sipgate is using a browser plugin to interface with their locally installed softphone, but offers also 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 resulting as mentioned from the necessity to enter into a B2B relationship with every provider who would like to enable browser based real-time communication for his users. APSINT's goal 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 people 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 (Mozilla.org, 2011) which allows access to telephony and messaging APIs via JavaScript, besides 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., 2009) suggest a system that uses an architecture similar to the one presented in this paper. They also envisioned the possibility to deploy such software either locally on the client or remote at a server. However their Web-IMS

cooperation is based on flash plugins and transcoding of media. Our solution presented in this paper does not need any modification or additional plugin in a browser and uses HTML5 instead. Also transcoding is not necessary in our solution.

Another initiative is W3C WebRTC (Google, 2011), whose main purpose is to enable streamed real-time communication from and to the browser without interacting with local device capabilities.

3 OUR SOLUTION & ARCHITECTURE

This section gives an overview of the APSINT architecture. Its components and interfaces are depicted in Figure 3. An APSINT enabled Web site is downloaded from *Webserver A* via a standard HTTP connection (a). A Web browser, running on a smartphone, renders and executes a Javascript (JS) as soon as it is downloaded to the client. The developer of the Web site can easily integrate real-time communication by just using JS API calls. The *APSINT.js* connects to the endpoint via local Websockets (b). This communication is transparent to the Web site developer. There is a 1:1 relationship between Web browser and endpoint. Only one Web site can connect to the endpoint at the same time. Usually this is the last page which sent a communication request to the endpoint. In case of an existing session (e.g. an active call) the Web site used to initiate/terminate the call would keep full control over the endpoint. The phone's local features are accessed via the endpoint which uses device specific APIs (c) e.g. for SMS messaging or circuit-switched (CS) voice calls. An interface (d) connects the endpoint with the Telco operator. In our case this is realized via SIP/IMS signaling.

3.1 APSINT Protocol

The software architecture of APSINT is shown in Figure 3, which contains the *endpoint* component to link communication between a *Web browser* and a signalling stack for real-time communication such as SIP. In general the endpoint must implement asynchronous event driven communication between Javascript on browser side and the SIP stack on the other side. Browser side communication is interfaced by Websockets (Fette and Melnikov, 2011) as the transport protocol for bidirectional message delivery to and from the SIP stack. Websockets avoid strong binding of the endpoint to the browser on the one side, like with browser specific plugins, and minimize protocol specific overhead on the other side, as it is the

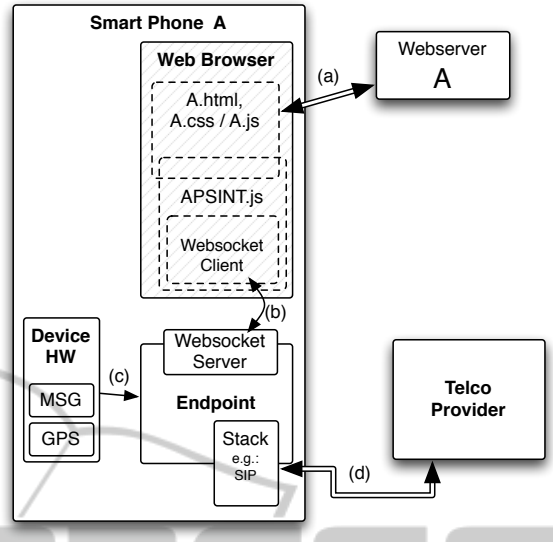

Figure 3: APSINT system architecture.

case when long polling is used.

The Websockets protocol as proposed in (Fette and Melnikov, 2011) enables Web browsers to establish a bidirectional channel to servers by upgrading a HTTP connection using an initial handshake. In APSINT the endpoint software component is required to implement a Websocket server, while the client side of Websockets is implemented by Web browsers. As the Websockets protocol is part of the HTML5 standard all major browsers offer a Websocket client. Hence the APSINT solution benefits from bidirectional connections and the low latency of the Websockets protocol while making browser specific plugins obsolete.

In the APSINT endpoint the Websocket server and the SIP stack run independently in their own event loops. Messages originating from Javascript are received by the Websocket server stack and passed over to the SIP stack in an asynchronous manner. In the opposite direction SIP events are interpreted by the SIP stack and passed as messages to the Websocket server service.

Messages *originating from Javascript* are

- RESERVE, Web page is registered to the endpoint,
- INITIATE, call is initiated by the Web page,
- MESSAGE, message is sent by the Web page,
- END, call is ended by the Web page.

Messages *originating from the SIP stack* consist of

- CALL_EVENT, triggers various events in a call session,
- STATUS, forward states of endpoint and SIP stack to the registered Web page,

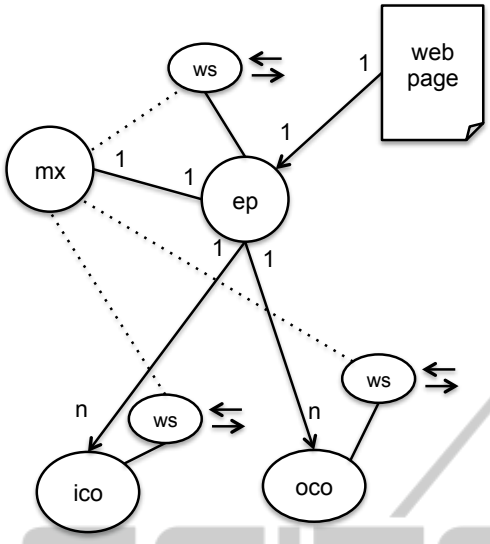

Figure 4: APSINT Javascript call objects.

- MESSAGE, forward received message by SIP stack to Web page,
- END, ending call session by remote party.

The APSINT architecture relies on SIP for signalling purposes. Although other signalling protocols exist, SIP was adopted by most telephony providers in connection with IMS. Hence for the APSINT endpoint it is necessary to integrate a SIP stack to make use of the telco IMS infrastructure. The SIP stack is an integrated service of the APSINT endpoint running uncoupled from the Websocket service, to provide asynchronous event triggering coming from the SIP layer. Furthermore the SIP stack service controls the media engine within the APSINT endpoint. The media engine is responsible for receiving and transmitting RTP media streams, and to play ring tones and ringback tones when triggered by the SIP stack.

3.2 Javascript Library and API

The APSINT Javascript library and API (apsint.js) is composed out of a set of object types and their relations as shown in Figure 4. A Web page would obtain a single endpoint object (`ep`) on successfully reserving the endpoint service. This `ep` object may be used to initiate new calls or sending messages. As well, call backs to be overriden by the Web page will inform about incoming calls and messages.

Initiating a new call via the `ep` object will instantiate a new outgoing call object `oco` which is handed over to the Web page. One `oco` object per call session will exist. Same as for the `ep` object the Web page is responsible to override the callbacks of that object to get notified on important call events.

On receiving a new incoming call the APSINT library will invoke a dedicated callback on the ep object passing along a newly created incoming call object (`ico`). For each incoming call session one `ico` object is instantiated. Similar to the `oco` this object contains callback methods for notifying the Web page on certain events and to provide utility methods to answer or end a call.

In case a callback function in any of the `ep`, `ico` or `oco` objects is not overriden by the Web page, the APSINT library will invoke a default implementation which may lead to automatically accepting or declining a call or displaying information in a default manner.

The mixer object (`mx`) is instantiated with the `ep` object at the time the Web page grabs the endpoint and is responsible for coordinating multiple call sessions, keeping track of active calls and other call handling tasks. By setting certain coordination methods to different (predefined) functions, behavior of how to deal with multiple session can be influenced. The architecture also gives respect to future enhancements of the library for toggling between calls or setting up three-way calls or conferences. Currently options for declining and reporting new calls to the Web page or taking over new calls while quitting existing sessions is implemented. The mixer object does its job by intercepting and manipulating the messages of endpoint end call objects towards and from the endpoint.

All objects except the mx object contain their own Websocket (`ws` in figure 4) for communication with the endpoint. Lifetime of the `ep`, `oco` and `ico` objects is tightly coupled to the lifetime of their Websocket. As long as the Websocket towards the endpoint is open the related object exists.

4 IMPLEMENTATION

4.1 Endpoint

The APSINT endpoint is designed as a background service. As shown in Figure 5 the endpoint implements three services:

- Websocket Server, for receiving and sending messages to the Web page and the SIP stack. The Websocket server is an asynchronous task in non-blocking mode.
- SIP Stack, for handling SIP protocol messages and controlling the media engine. The SIP stack has to be an asynchronous service like the Websocket server.

- Media Engine, for sending/receiving RTP media streams and for audio playback of ringtones. The media engine is completely controlled by the SIP stack.

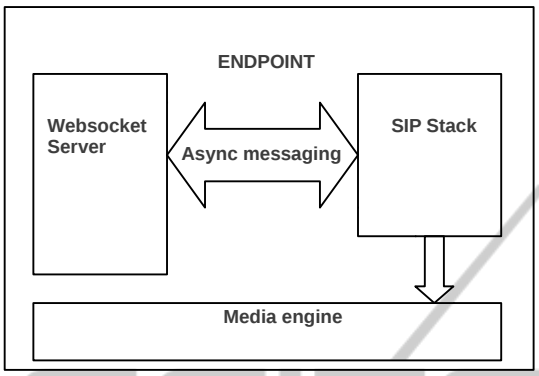

Figure 5: Overview of the APSINT endpoint.

A prototype of the APSINT endpoint was implemented for x86 personal computers on Linux and for Android 2.2 and above. Software and libraries used for the Linux PC prototype are:

- Sofia-SIP (Pessi et al., 2011), a SIP stack to implement SIP functionality in the APSINT endpoint software,
- libwebsockets (Green, 2011), a C library implementing the Websockets protocol,
- GStreamer framework for the media engine implementation.

A second prototype for the Android platform utilizes the following software components:

- IMSDroid (Diop, 2011), an IMS compliant SIP stack with integrated media engine for Android phones and tablets.
- Java-Websocket (Rajlich, 2011), a pure Java library implementing the Websockets protocol both for servers and clients.

On both platforms the endpoints can handle multiple SIP sessions and support simple SIP messaging. The endpoint on Android was extended and is capable to access system services offered by the phone platform. As an example, messaging on Android includes SMS sending and receiving.

4.2 Javascript Library

Any Web page on the system, even across browsers, may reserve the endpoint. However, only one Web page at a time may use the endpoint services. The last one asking for the endpoint may use it, if it is not busy with serving another Web page. The rules for reserving the endpoint are:

- If no other Web page has reserved the endpoint then the actual requesting Web page can use the endpoint
- If some other Web page 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 Web page can use the endpoint. The endpoint running in a separate process will detect that the Web page that has reserved it is no longer present because it lost the Websocket connection.
- If some other Web page has reserved the endpoint but is not using it any more, i.e. has no open communication session, then this Web page loses its reservation. It may regain the endpoint services by reserving the endpoint again some time later.
- If some other Web page has a communication session still running the reservation request will be denied

The initial call to obtain endpoint services is the invocation of the reserve function as part of the ENDPOINT class. User ID and user credentials may be passed depending on optional security mechanisms. In addition, any local or remote Websocket URL towards the endpoint can be configured. If not provided a default local address will be used. However, the website must provide an activation callback for reservation. Once connection with the endpoint is established this function is called by the APSINT library providing the target function with the actual endpoint object (`ep` as depicted in figure 4).

The `ep` object may in turn create `ico` and `oco`, incoming and outgoing call objects depending on who is the call originator (`ico` for incoming calls, `oco` for self initiated outgoing calls). The three objects have a common basic structure, which is related to the need of communicating independently with the APSINT endpoint. This is guaranteed by each object instance using its own Websocket. The following list shows common and distinct functions of the APSINT communication objects:

Common functions for `ep`, `ico` and `oco` objects are:

- `parseEvent` and `dispatchEvent` for message and event handling in interaction with the endpoint
- `makeWebSocket` used by the object creation factory to obtain and connect the instance to a Websocket
- `sendMSG` for sending a (SIP or SMS) message

- `onDestroy` - is called when the object is about to be released by the library or if the Websocket connection went down.

Functions of the ep object only are:

- `initiateCall` to make a new call (session) providing address string media (audio and/or video) and a notification callback which will deliver the new `oco` call object for the session once the endpoint started to process the connection
- `sendMessage` to send a text message via SIP or as SMS (Android only)
- `onIncomingCall` invoked on being called via audio or video providing the new ico object used to deal with the new connection
- `onMessage` called if a SIP Message message or an SMS was received
- `onStatusChange` used to communicate status changes of the endpoint

Functions common for both `ico` and `oco` objects:

- `endCall` to take down the communication session regardless of its current state
- `onCallEnded` to inform about the termination of he session by the communication partner at any point in time
- `onError` to inform about call related errors
- `audio` and `video` prepared for future releases to turn on/off media during the session

Functions for `ico` objects only:

- `answerCall` used to acknowledge the incoming call request leading to immediate call establishment
- `onRinging` to inform the Web page that the call originator has been signaled a ”free line”

Functions for `oco` objects only:

- `onRingBack` to inform that the called party has signalled a ”free line”
- `onCallAnswered` to inform that the call has been answered be the terminator and that the session is now established

The mixer object (`mx`) for multi session coordination is a little different. It does not inherit from the current base class of the APSINT objects and does not use its own Websockets. However, the factory for creating `ep`, `ico` and `oco` objects ties in observation calls so that the `mx` object is informed on each incoming and outgoing message that any of the other objects is sending towards or receiving from the endpoint. The `mx` object can also modify, insert or remove these messages to perform coordination actions. The member functions of the mixer object are:

- `getCalls` to obtain a list of all currently managed call objects
- `getActiveCall` to obtain the call object which is currently active, meaning the call currently transmitting or receiving media
- `setActiveCall` to make some other connection (i.e. call session) the active call
- `declineAdditionalCall` may be used to decline all incoming calls during an ongoing session
- `acceptAdditionalCall` makes the new incoming call the active one and terminates the former active session
- `onAdditionalCall` may be set to one of the two functions above or to some other function customized by the Web page

All functions/methods starting with ”on” in its name are notification callbacks meant to be overridden by the Web page. In a minimum configuration the `onIncomingCall` of the `ep` object needs to be provided to get interaction capabilities with the endpoint. More callbacks may be customized by the web designer as appropriate (cf. next section).

4.3 Web Application

We have implemented a web application to showcase the possibilities of our solution. *Sushify* is a Twitter-like microblogging platform written in Ruby on Rails. The main entities of Sushify are:

- User.
- Micropost.
- Relationship.

These are also reflected as Rails models and stored in a SQLite database. Sushify is fully based on a REST architecture (Fielding, 2000) thus rendering all the models as resources that can be manipulated via standard HTTP methods. Similar to Twitter a user can create microposts and users can follow each other thus creating a relationship. Posts from followed users are shown in an aggregated micropost feed.

Building upon this feature set we have included the possibility to trigger calls and send messages using the Javascript API discussed above. Fig. 6 shows the Sushify UI with an active call window. Basically three Javascript files are used and included in the file `config/application.rb`.

- `apsint.js` The Javascript library file outline above
- `apsint-vts.js` Contains specific call and message handling code by overwriting methods of `apsint.js` such as `answercall()`.

- `apsint-ui.js` Methods of `apsint-vts.js` usually call UI-related functions in this file, e.g. opening a message box with the caller id. User input (like declining a call) is handled here as well, and the corresponding method in `apsint-vts.js` is called (e.g. `declinecall()`).

Figure 6: Sushify with active call window.

Users can configure a SIP address where they can be reached. We have also implemented a feature that allows an auto-reply message to be sent when a user declines a call. In order to prevent unsolicited calls a user can only call and message users (and see their SIP/email address) that are his/her followers.

5 DISCUSSION

The advantages of our approach over other architectures is based on the following two design decisions:

1. Resolve the need for a B2B relationship between Web provider and communication service provider.
2. Distribute the Model, View and Control components of an application across different processes and even across the network.

The first point will allow applications to use any communication service provider the user has subscribed to and hence there is no need for configuration or contract signing to use the application. Furthermore, the resolution of the B2B relationship will not only work for delivering communications services via local device functionality but also to trigger any service invocation from the users device instead of from the providers server towards network provider or any other 3rd party service the user can authenticate with.

The second point enables mobile applications to be always up to date while running stable in close integration with mobile device capabilities. View and Controller will be downloaded via HTTP and presented and controlled via HTML, CSS and Javascript kept up do date on the server. At the same time apps are executed locally with all the required functionality provided natively by the device (in the Model component running in a separate process). View and main parts of the controller are downloaded from a web server and executed in the browser whereas, low level controller functionality and the Model component dealing with the SIP stack are running as a native background process communicating with the browser.

It is not necessary for the user to share his secrets with third parties such as authentication credentials, address book data or calendar information. The required data may be added only at the time of local execution in the mobile browser. The user keeps control over private data. For example, the user wants to call a friend on a social web platform. The name or nick name of the friend may be found on the Web page but not his phone number which is associated to the friends nick in the local address book of the users device. During APSINT Javascript execution both are combined in the browser of the users device. A call is made via the Web page without the Web page knowing the details.

Other real-time communication solutions or native functionality execution like WebRTC, plugin based solutions (Adeyeye et al., 2012) or implementation of a Web app with Googles NativeClient are not capable of reacting on external stimula, e.g. if no browser is running it is not possible to take an incoming communication request. With the APSINT endpoint running as a service on the local device it is always possible to start a browser and download a Web page to take an incoming call.

Also, while experimenting with our prototype, we introduced a simple method of rendering video directly in the browser by using the `data` URI scheme to put base64 encoded data right into the `src` attribute of a plain HTML `img` tag. Together with this functionality and the Websocket based communication between browser and communication stack it was possible to distribute functionality across devices. Doing this, it is possible to communicate via a smartphones SIP connection while displaying the related Web page GUI for communication and video display on the monitor of a nearby PC (in the same LAN). One would talk via the phone but control communication and watch video on the PC.

No plugin, especially no mobile flash support is required. Our architecture supports multiple commu-

nication sessions to be supported yet limits the usage of communication resources to one Web page at a time. The endpoint will provide its services for other Web pages if the current using Web app does have no open sessions and will detect crashes of a Web page or the browser automatically as reserving the communication stack is tied to keeping the Websocket session open.

Unlike other approaches, the APSINT architecture offers the possibility to communicate with clients of other type and via telco network features protocols and networks of other kind. With an APSINT enabled Web page one can communicate with any other SIP client reachable in the network or via breakout functionality of telco providers with any other party in the mobile or fixed line network. These solutions are standardized and available worldwide.

5.1 Security Issues

Security issues have a paramount importance for the all parties involved in consuming and providing services build on top of APSINT architecture. This is because of the specific setup which allows a Web page to take control over audio and video sessions started from the users devices like Smartphone. Hence on the usability level security requirements revolve around achieving trust in this new functionality by providing a reliable solution in term of authentication and authorization of communication sessions started by the Web pages, as well as, providing privacy and confidentiality. Security measures need to address specific focus of all parties involved, as discussed below.

Users may be accustomed to the dangers of the internet and accept the risks because of vital importance of this platform. However, they may be difficult to persuade to grant control over their phones to an internet application unless they can trust their security requirements are met. The users requirements cover different areas which stretch from preventing of starting unsolicited communication sessions and possibly turning their devices to spy on users communication or hijacking them for SPIT attacks, securing privacy of communication, up to mitigation of phishing, e.g. in form of persuading users to call a costly service numbers.

Similarly, network providers are interested in preventing SPIT or DoS attacks on their customers which may be caused when malicious Web pages could obtain control over devices connected to the providers network. They would as well prefer situation when they could unambiguously identify sessions originated by the Web pages and associate them with the specific page. This may be especially important when APSINT architecture capable devices would be branded by the operator. In this case users would surely expect the operator to take at least partial responsibility in case when allowing Web pages to control communication sessions would inflict substantial cost s to the user, e.g. due to phishing attack.

For the owners of the Web pages with capabilities to control communication sessions on the APSINT devices winning user trust is very important. User need to have a guarantee that they do not give control to start audio or video sessions to the malicious Web page. Therefore stealing the functionality for controlling APSINT terminals embedded in a specific Web page and reusing it in the context of a different Web page should be prevented. If the revenues from the voice and video traffic generated by the Web page need to be shared with the network operator, then unambiguous identification of such sessions is needed.

Traditional architecture for Web-IMS convergence is based on Parlay X interface in the IMS application server and it has well defined security framework. Critics of the efficiency of the Parlay X based architecture brought alternative proposal based on new functional entity on the IMS network border named Web Session Controller which shields the IMS terminal from direct interaction with Web application. APSINT project gives a new concept to Web and IMS convergence which takes place mainly in users terminal. Direct interactions between IMS terminal and Web applications require to explore ways of combining security solutions for Web applications with IMS security standards applicable for this novel architecture. Approach taken by the APSINT team tries to flexibly adapt security restrictions in consuming Web applications to the degree of trust that user expresses against a Web page with embedded APSINT application. Web applications being consumed in the users browser will be secured by the known technologies like SSH or digital signatures, however, user will be allowed to grant his permit to specific Web pages for establishing audio and video sessions either permanent or for the actual session only depending on his trust toward this Web page. IMS security standards will be fully supported. Additionally a new security measure is studied for providing to IMS identity of Web application which was allowed to start audio/video session from the particular IMS terminal.

6 CONCLUSIONS AND FUTURE WORK

As the APSINT solution integrates the endpoint on local device the browser communicates with the end-

point locally and all signalling is handled by the local endpoint. An other approach is to move the endpoint to a remote host as shown in Figure 7 where a local browser communicates over Websockets with a remote endpoint. For this solution to be applicable the local device still needs to implement a local media engine. In such an approach all signalling is handled by a remote server whereas the media stream is handled by the respective end devices. Finally the media engine could be integrated in the browser so the media resources both hardware and software are managed by the Web browser itself as proposed by WebRTC (Google, 2011).

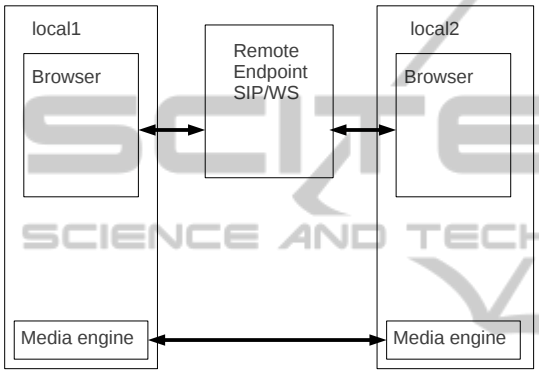

Figure 7: Architecture of a remote endpoint.

Extensions to the endpoint could be added by utilizing platform services offered by the operating system on user devices like for example on Android phones. Candidates for such usage are location services, camera, phonebook, GSM calls and SMS. A similar approach is made by Mozilla in WebAPI project (Mozilla.org, 2011).

6.1 Porting to other Systems

Currently the endpoint is running on the Android platform as well as on Linux and Mac (OS X) desktop systems. We have also investigated possibilities of porting the endpoint to other operating systems:

- *iOS*: The Doubango SIP stack has been already ported to this platform, thus porting the endpoint software should be relatively easy.
- *Symbian*: Should be also relatively straightforward as the Sofia SIP stack (developed by Nokia and also used in desktop versions of the endpoint) is fully supported on that platform.
- *Windows Phone 7*: To our knowledge it is not possible to use/compile external libraries for this platform due to security restrictions (mainly caused by the lack of a multiuser concept in the kernel). As a consequence the only way for implementation would be from the ground up with the SilverLight IDE, which is clearly not a feasible approach.

Finally, we expect no major problems in supporting Windows desktops (since all the necessary libraries/compilers are available).

6.2 API Evaluation

We are planning to perform an evaluation of our Javascript library to assess acceptance among the developer community. As a first step we will review and simplify our API, write documentation and provide code examples, best practices and such. The evaluation should be carried out in two phases:

- **Laboratory Test:** A two-hour test with 8-10 developers that should solve 2-3 tasks. We are considering quantitative criteria such as: task completion time, lines of code, iteration steps needed (Clarke, 2004). Also think-aloud and maybe video observation might reveal more hidden issues.
- **Real World Test:** Developers get the API/documentation to use it for free tasks/projects. They give feedback in form of diaries and are supported by us throughout the study (4-6 weeks).

Combining both a laboratory test and a longitudinal real-world study (Gerken et al., 2011) is a novel approach in evaluating a API and we expect richer results with this two-phase-approach.

ACKNOWLEDGEMENTS

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 colleagues Vincenzo Scotto di Carlo and Hans-Heinrich Grusdt from Nokia Siemens Networks Germany and Marco Happenhofer from Vienna University of Technology for their contributions to this work.

REFERENCES

Adeyeye, M., Ventura, N., and Foschini, L. (2012). Converged multimedia services in emerging web 2.0 session mobility scenarios. *Wireless Networks*, 18:185–197. 10.1007/s11276-011-0394-z.

Clarke, S. (2004). Measuring API usability. *Dr. Dobbs Journal*, pages 6–9.

Diop, M. (2011). High Quality Video SIP/IMS client for Google Android. http://code.google.com/p/imsdroid/. Accessed: 15/11/2011.

Fette, I. and Melnikov, A. (2011). The WebSocket protocol. http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-17. Accessed: 15/11/2011.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Gerken, J., Jetter, H.-C., Zöllner, M., Mader, M., and Reiterer, H. (2011). The concept maps method as a tool to evaluate the usability of APIs. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 3373–3382, New York, NY, USA. ACM.

Google (2011). WebRTC is a free, open project that enables web browsers with Real-Time Communications (RTC) capabilities via simple Javascript APIs. http://www.webrtc.org/home. Accessed: 15/11/2011.

Green, A. (2011). C Websockets Server Library. http://git.warmcat.com/cgi-bin/cgit/libwebsockets. Accessed: 15/11/2011.

Mozilla.org (2011). 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. Accessed: 15/11/2011.

Nishimura, H., Ohnimushi, H., and Hirano, M. (2009). 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. IEEE.

Pessi, P. et al. (2011). Sofia-SIP - a RFC3261 compliant SIP User-Agent library. http://sofia-sip.sourceforge.net/. Accessed: 15/11/2011.

Rajlich, N. (2011). A barebones WebSocket client and server implementation written in 100% Java. https://github.com/TooTallNate/Java-WebSocket. Accessed: 15/11/2011.

Sipgate.com (2011). Move your phones to the cloud. http://sipgate.com. Accessed: 20/11/2011.

Tropo.com (2011). Tropo - cloud api for voice, sms, and instant messaging services. https://www.tropo.com. Accessed: 20/11/2011.

Winokur, D. (2011). 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. Accessed: 20/11/2011.