Towards a Server-Centric Interaction Architecture for Wireless Applications

Jussi Saarinen[1], Tommi Mikkonen[1], Sasu Tarkoma[2], Jani Heikkinen[2] and Risto Pitkänen[3], *Non-Members*
[1]Department of Software Systems, Tampere University of Technology,
P.O. Box 553, FI-33101 Tampere, Finland
[e-mail: jussi.p.saarinen@tut.fi, tommi.mikkonen@tut.fi]
[2]Telecommunications and Multimedia Laboratory, Helsinki University of Technology,
P.O. Box 5400, FI-02015 Espoo, Finland
[e-mail: sasu.tarkoma@hut.fi, jani.heikkinen@hut.fi]
[3]Atostek Ltd., P.O. Box 107, FI-33721 Tampere, Finland
[e-mail: risto.pitkänen@atostek.com]
*Corresponding author: Tommi Mikkonen

Received February 3, 2008; revised March 10, 2008; accepted April 10, 2008; published April 25, 2008

Abstract

Traditional web-based services that require users to browse via documents and fill out forms, are difficult to use with mobile devices. Moreover, as the web paradigm assumes active clients, further complications are introduced in cases where the server is the active entity, instead of the client. This paper presents a Server-Centric Interaction Architecture (SCIA) for wireless applications. The architecture enables servers to initiate communication with clients as well as push secure targeted data to them, in a piecemeal fashion. It further enables the development of highly collaborative wireless services with interactive user interfaces.

Keywords: Server initiated interactions, push semantics, mobile services, service platform

Digital Object Identifier 10.3837/tiis.2008.02.003

1. Introduction

The traditional paradigm of web applications is strongly dependant on the client-server architecture. Servers of the system are considered as somewhat passive holders of content, and clients are actively downloading data from suitable servers. If necessary, clients can also modify and upload data, which results in servers updating their databases. The change in the database is reflected in recently loaded web pages, but those that were previously downloaded by clients are not automatically updated. Instead, either the user should reload the page, or updating capabilities should be explicitly coded in the page, which may add considerable networking load.

While this scheme should be adequate for systems where data access is the main issue, it is fundamentally flawed for other types of applications, due to the inherent restrictions. These restrictions are overcome by introducing new methods for client-server interactions. In particular, when developing new types of applications, where interactive services are provided to users, the server must play a more active role. On a small and primitive scale, this is already occurring, via the use of services such as push email, for instance, as popularized by Blackberry.

Assigning a more active role to the server enables the creation of services where the infrastructure can make decisions about information delivery. For instance, assume a situation where an aircraft will overrun its schedule and passengers need to be rerouted. While the plane is still in the air, it will be possible for the ground system to perform necessary rerouting. When the plane finally lands, information regarding the changed routes will be distributed wirelessly (via mobile phones) to all passengers who need the information. Other similar examples includes the reservation of concert or sports event tickets when they become available, notification and opportunity to trade stock options when market events occur, and changes in web auctions, such as eBay, which can presently be implemented with techniques such as email notification. We believe that cases where the service provider knows how to best serve the client are common – people are generally interested in updated information, not information they previously accessed – but restrictions of available technologies essentially prevent straightforward implementation of this. Enabling such services to be pushed to known clients results in improved user experience.

Another issue requiring resolution is user-provided input and associated updates. Clearly, current mobile Internet applications are mostly based on conventional documents and forms, even though browsing documents and filling out forms are cumbersome on a small mobile device. On the other hand, highly interactive web applications are the latest Internet trend. For example, Google Maps can update the map view in real-time, when the user drags the map with his mouse pointer. Richer and more flexible interfaces are possible by the use of technologies such as Ajax [1] and Sun Labs Lively Kernel [2]. Utilizing such technologies in mobile devices would liberate designers from restrictions associated with documents and forms, thus, providing the potential for enhanced usability and user experience.

This paper involves constructing a Server-Centric Interaction Architecture (SCIA) that utilizes push-enabled servers in web applications. In addition, we enable the introduction of application and user specific information, which should be advantageous for composing web applications. We developed a proof-of-concept system, where a special-purpose web browser was extended to handle server-initiated activities. In addition, the system was constructed so that it is possible to download partial web pages and modify the user interface on-the-fly, according to the received data, with techniques resembling Ajax. This improves usability of implemented services, as there is no need to wait for the download of complete pages or the

upload of frames that contain user-defined data. The work was done as a part of the Wesahmi project, a joint effort of three Finnish universities, Tampere University of Technology (TUT), Helsinki University of Technology (TKK), and University of Helsinki (UHE). TKK provided the client-side implementation and security support implementation, and UHE provided necessary communication protocol implementation. The paper focuses on the server architecture and information pushing, which were designed and implemented at TUT.

The structure of the rest of this paper is as follows. In Section 2 we briefly present the most essential related work. Section 3 introduces the theory of server initiated interactions and compares it to existing technologies. Section 4 describes an architecture that supports these interactions, and Section 5 introduces a sample execution of our system. Then, Section 6 presents the security mechanisms supporting the architecture. Finally, in Section 7, conclusions are drawn and future work is presented.

2. Related Work

Ajax is the most common current method that enables asynchronous updates to web page content. It is based on user triggered events, but can also be used to implement server initiated push via client initiated sessions. The sessions can then be used to relay updates from server to client without further user intervention. However, Ajax does not allow development of truly server initiated interactions without significant extensions.

The Ajax derived method Comet uses the same technologies in a different manner. It supports updates to web page contents via server initiated push. It was shown that it drained server resources in [3] where the number of clients increase and traditional servlet containers are used. There are several server-side implementations that address this issue. For example, the Java-based server Jetty provides an implementation of Asynchronous Request Processing (ARP) called Continuations [4]. It frees server resources tied to idle request threads by returning them to the thread pool. Another limitation of Comet applications is imposed by the HTTP 1.1 standard [5] which requires that a single client must not maintain more than two persistent connections to a server. A quick resolution to this issue is multiplex messaging using the Bayeux protocol [6] which is implemented in [7].

Elvin [8] is a framework that supports content-based messaging on mobile devices. It focuses mainly on supporting occasional disconnection of devices, rather than providing support for services that require high interactivity. Another significant framework which is similar is Java Event-Based Distributed Infrastructure (JEDI) [9]. It provides a framework for large scale publish-subscribe systems and addresses issues related to client mobility. However, it does not actually handle practical details associated with user interactions in a mobile environment.

3. Server-Initiated Interactions

We use the term Server-initiated interaction to describe an interactive session formed between a client and a server launched and controlled by the latter. Thus, the server can push targeted data to clients, which reduces unnecessary user interference. For example, server-initiated interactions could be used to guide a user via a check-in procedure at an airport. When a potential user's mobile device contacts the wireless network of the airport, he or she can automatically be shown a dialog box that inquires whether he or she would like to check-

in or not. If he or she answers yes, he or she can then be guided via the rest of the check-in process. Suppose that the user is later spending his time at an airport cafe and the flight he or she checked in for is delayed. The server at the airport receives information about the flight from the back-end system of the airline company and sends a notification to the user's device. Since the system knows that the user is at the airport, an advertisement based on his user profile can be pushed to his or her device or he or she are guided to the nearest VIP lounge for example, for a more comfortable waiting period. Next, we study different implementation alternatives.

The most common technologies during the past few years which provide users interactive services on the Internet were Hypertext Transport Protocol (HTTP) [5] in combination with Hypertext Markup Language (HTML) [10] and JavaScript [11] or other similar scripting languages. However, these do not support information pushing. Most calculations are performed by the server, and the user is forced to wait for a new document to be loaded. This general application model is described in **Fig. 1**.

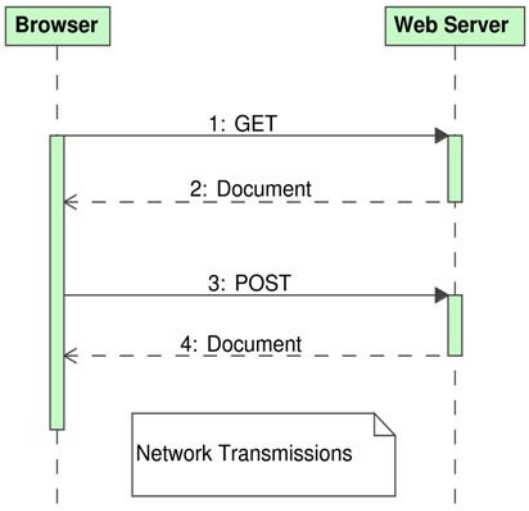

Fig. 1. Traditional web application model

More sophisticated systems exist. Ajax is a relatively new approach that simply uses old technologies in a new way [1]. It reduces the data flow between a client and a server by performing most functions on the client side, and provides better bandwidth utilization and enhanced interaction than earlier methods. It is still not the optimal solution in an environment where client devices have very limited processing power and battery capacity. The basic Ajax application model is illustrated in **Fig. 2**.

A method to achieve event-driven, server-push data streaming called Comet was defined in [12]. It uses the same technologies as Ajax in a different manner. A Comet application uses long-lived HTTP connections to enable the server to deliver data to clients. This enables it to enhance the responsiveness of multi-user applications by eliminating polling, thus, reducing latency. The Comet application model is described in **Fig. 3**. First, the Comet Client initializes a persistent connection to the Comet Event Bus subscribing to a certain topic or topics. Then, each event that matches the subscription is forwarded to the client via the connection and finally displayed by its browser. The topic-based subscription model provides limited granularity, thus, it is not ideal for a mobile environment. Furthermore, the method does not clearly isolate presentation from content, thus, hindering reuse of

applications.

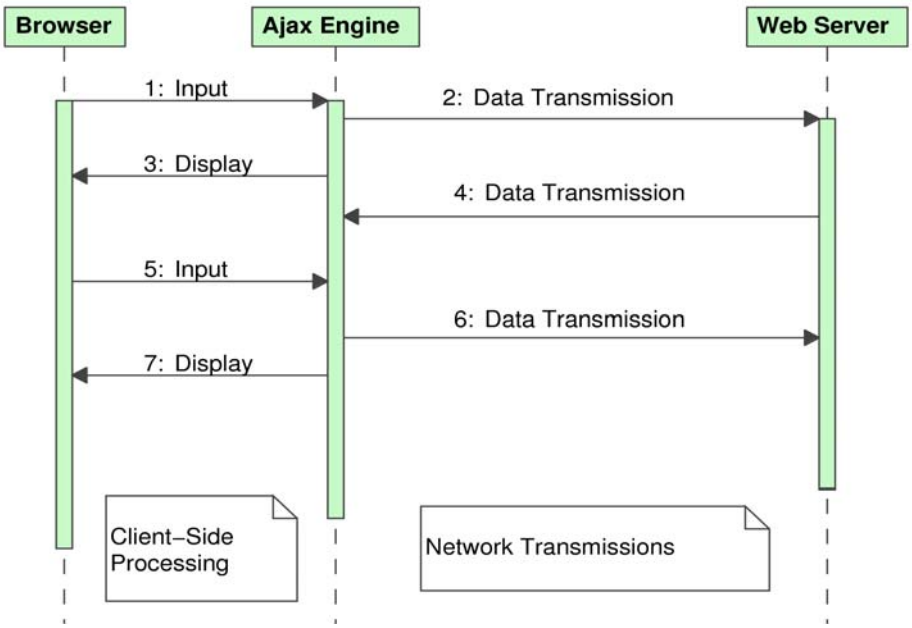

Fig. 2. Ajax application model

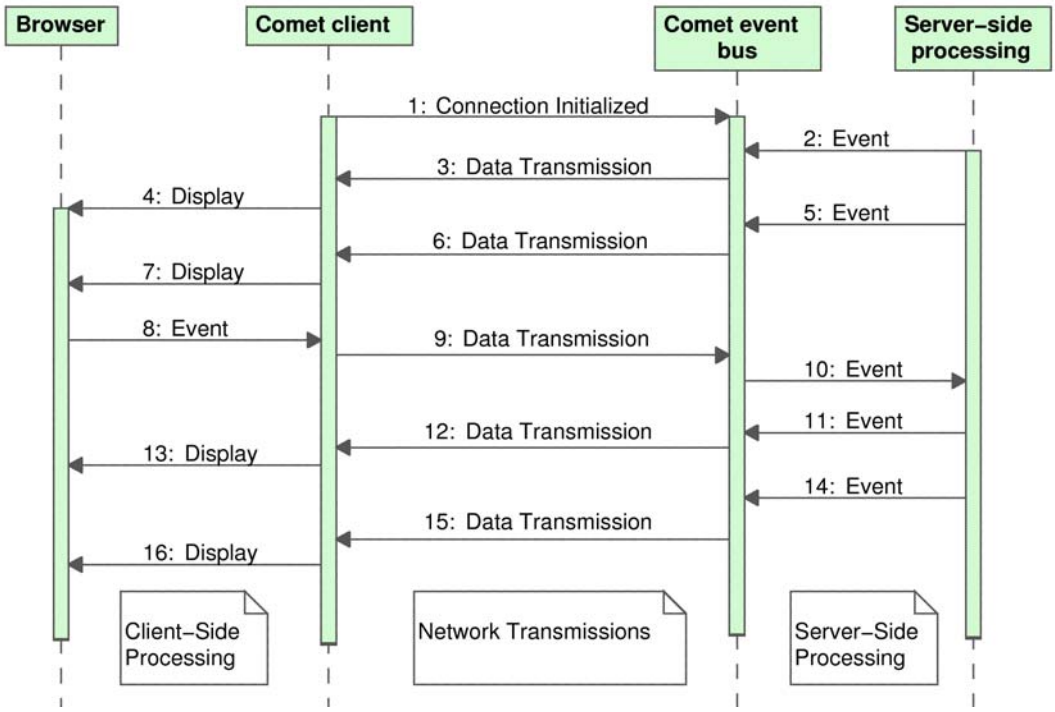

Fig. 3. Comet application model

Our server-initiated interaction model aims to enable true targeted and session-based information pushing. It also aims to be mobile device friendly, by concentrating the

computationally demanding tasks to the server as well as minimizing the amount of transferred data. These characteristics are also supported by the security model defined later in this paper. An illustration of the interaction model's functionality is displayed in **Fig. 4**.

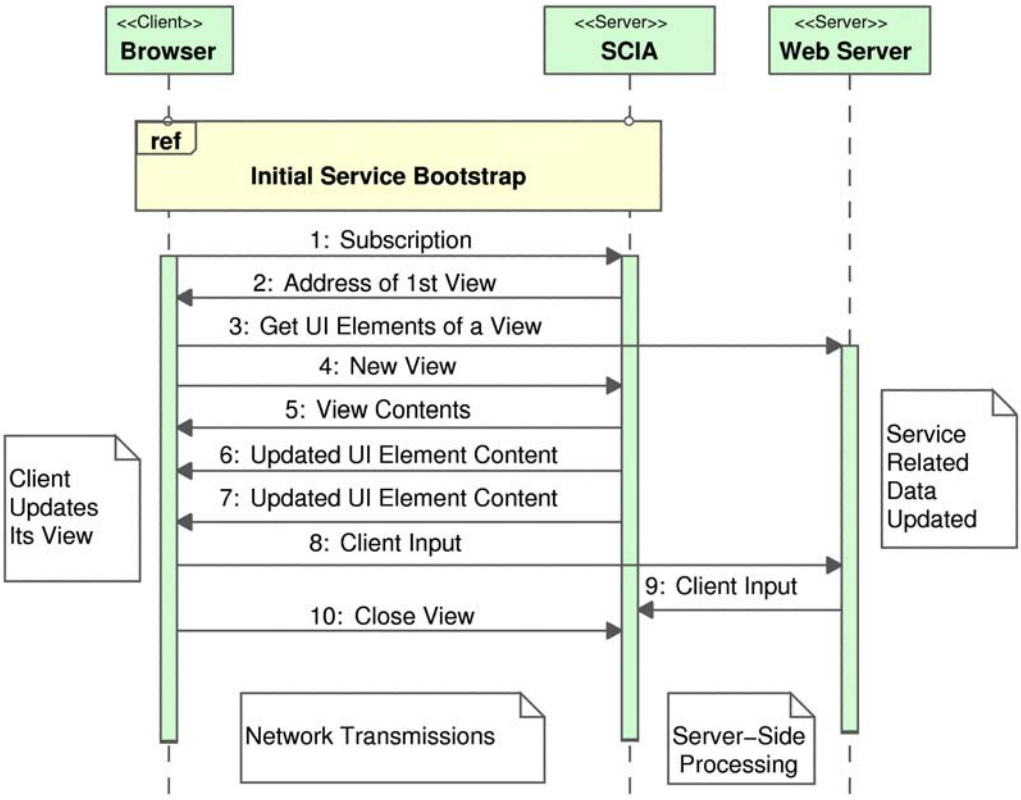

Fig. 4. Server initiated interactions application model

After initial service bootstrapping and subscription to the service (1), the client's browser receives a notification containing the address of the first View (2). In our model a View equals a set of User Interface (UI) components, each having a unique identifier. The browser retrieves UI components from the web server (3) and issues a New View request (4) to SCIA, which responds with the initial contents of the UI components associated with given component IDs (5). These UI components can then be dynamically updated by the server, without any user interaction (6-7). All updates the browser receives are tailored on-the-fly, according to components such as the user's profile or previous input. After the user submits input via the web server (8-9), his or her browser retrieves UI components and the content of the second View, in a similar manner to the first. Then, the browser closes the first View by issuing a close view request. Next, the server is ready to send updates for UI components of the second View. A procedure performed by an imaginary traveler using our special-purpose browser executing an airliner application is provided in **Fig. 5** - **Fig. 8**, with the following events:

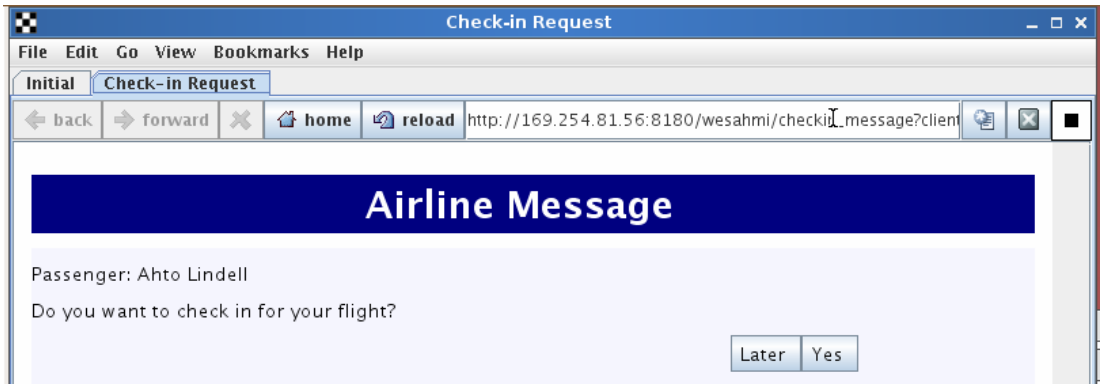

Fig. 5. Performing check-in procedure

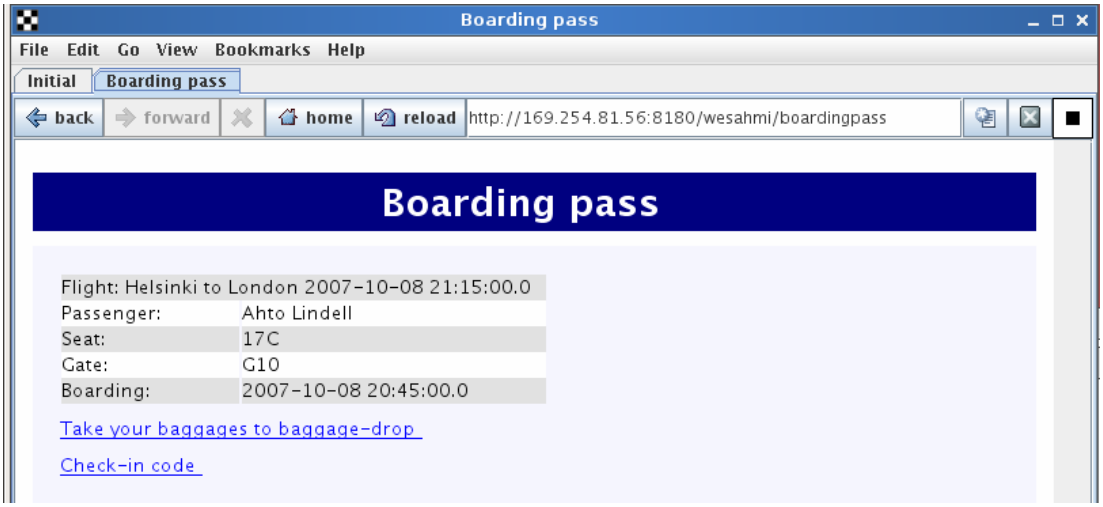

Fig. 6. The user receives a boarding pass

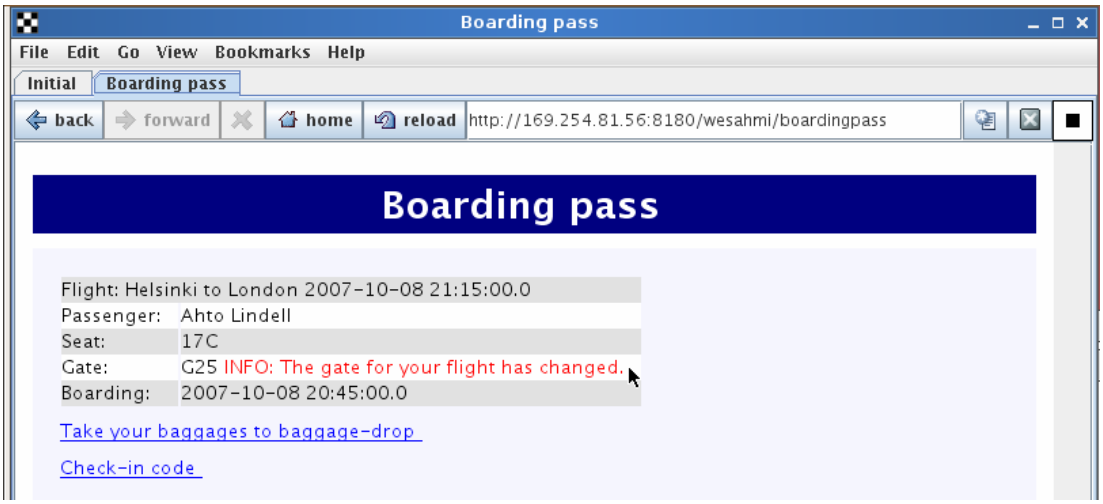

Fig. 7. Gate changes

Fig. 8. Boarding time changes

- When the traveler arrives to the airport, our system advertises available services, and provides an opportunity for the user to check in for the flight (**Fig. 5**).
- Next, the user selects check-in from the displayed view, and is given a seat to a previously received flight (**Fig. 6**). The user is also instructed to proceed to baggage-dropping area and security inspection.
- While the user waits for his flight, gates of the airport are reallocated. The system informs the user about the change (**Fig. 7**).
- Finally, due to the reallocation of the gates, the boarding time is delayed, which is again notified using our system (**Fig. 8**).

For test purposes, we have also implemented a simulator that can be used for composing notifications (**Fig. 9**). The main differences from other systems introduced, including traditional web applications and using Ajax, Comet or Elvin as described above, in our approach are the following:

- The user can go directly to the correct web site when needed using the boot-up system we have developed. This is convenient when targeting use cases where the user enters an area where certain services are assumed to be used, such as an airport.
- Sessions associated with users are established at the level of implementation infrastructure, which requires special measures such as cookies, when relying on the conventional web infrastructure.

4. Proposed Architecture

We propose an architecture, presented on **Fig. 10**, for server initiated interactions. In our architecture the *External Model* module represents the service provider's back-end system that provides the *SCIA Server* application specific data. The *SCIA Server* module plays a central role in the architecture, as it utilizes the remaining modules to deliver updates from *External Model* to clients, and conveys client input in the reverse direction. In the following, we introduce the main components of our system in more detail.

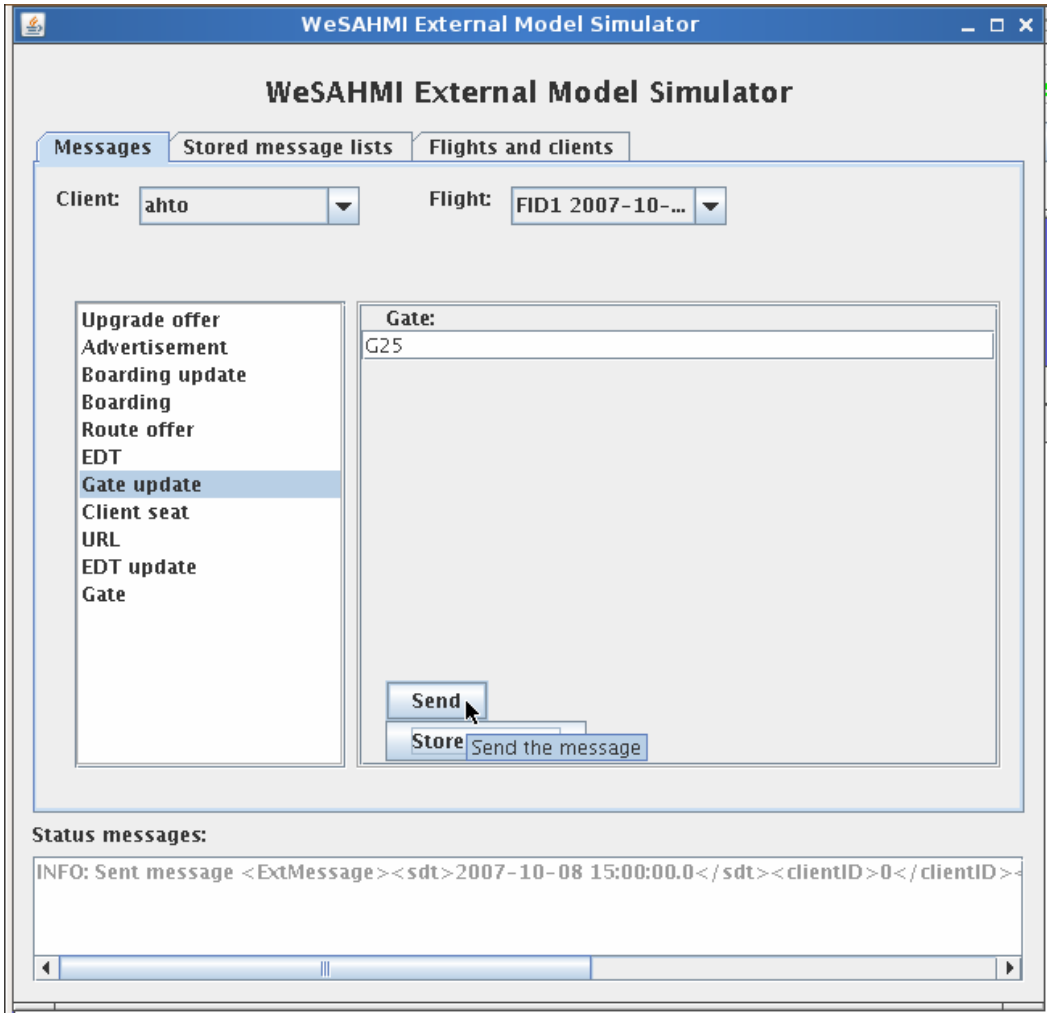

Fig. 9. Event simulator UI

Scheduling Service component is used to filter updates relevant to the state of each client and enable efficient distribution of the entire server architecture. A client's state is defined in the server's context by its open *Views*. *Notification Service* is used to deliver updates, which are encapsulated in notification messages, to corresponding clients. *WWW Server* provides User Interface (UI) components for the clients' *Browsers*. Each UI component is assigned an identifier, which is unique within the scope of a single site. *WWW Server* also hosts server-side code that handles client input, and passes it to the *SCIA Server* Module. *Browser* listens to service advertisements, subscribes to the service, and retrieves UI components from *WWW Server*. It can have multiple open *Views* simultaneously, each associated with a unique server assigned identifier. This enables *Views* to be independently updated by the server, and *Browser* to close specific *Views*.

Table 1 contains an example of a notification message used to pass information from *SCIA Server* to *Browser*. The message is encapsulated using *Simple Object Access Protocol* (SOAP), thus, containing standard SOAP components *Envelope* and *Body*.

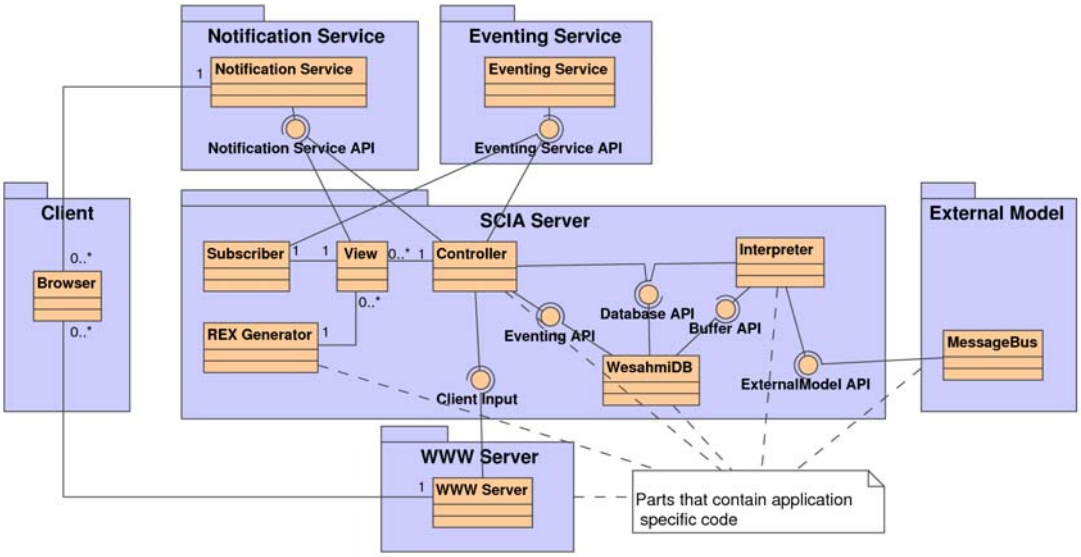

Fig. 10. Overview of the architecture

Table 1. Update to component *edt* in the *ContentChanged* SOAP method

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>

    <wes:ContentChanged xmlns:wes="http://www.tml.hut.fi/Research/wesahmi">
    <wes:service>finnair</wes:service>
    <wes:viewID>view1</wes:viewID>

    <rex xmlns='http://www.w3.org/ns/rex#'>
      <event target='id("edt")' name='DOMNodeRemoved'>
        <span xmlns='http://www.w3.org/1999/xhtml' id='edt'>
          2007-02-01 15:00:00
        </span>
      </event>
    </rex>

    </wes:contentChanged>
  </env:Body>
</env:Envelope>
```

The message contains an update to a single UI component with identifier *edt*. The SOAP method is indicated by *ContentChanged*, and the service name by the *service* child component. The target *View* is indicated by *ViewID* of the method. It has an attribute *target* that contains the identifier of the UI component to be updated, and an attribute *name* that indicates the type of the event. *Span* contains the update for the UI component.

The architecture of the *SCIA Server* package conforms to the commonly used Model-View-Controller (MVC) pattern. *SCIA DB*, which is used as a cache for *External Model* specific data, represents the Model of the pattern. It enables the storage and timely retrieval of application specific data. A View component of MVC is represented by an instance of a *View* class and the corresponding set of UI components on the *WWW Server*. Each *View* uses a *Subscriber* to subscribe to updates about UI components using the *Scheduling Service*. The content IDs of each UI component are used to filter updates so that only those associated with each *View* are forwarded to *Browser*s accessing it. The Controller component of the pattern is represented by the *Controller* class, which acts as the central module of the overall architecture. It initializes an instance of *SCIA DB* at startup, and instances of *View* whenever a request for a new view is received from *Browser*. It also processes client input received from *WWW Server*, and forwards it to *SCIA DB*.

The architecture of the *SCIA Server* package includes helper classes *Interpreter*, *REX Generator*, and *Subscriber*. The *Interpreter* class translates messages from the format used by *External Model* to the internal data structure used by the *SCIA Server* architecture and vice versa. The module hosts an outbound message buffer, from which *External Model* may retrieve new messages containing client submitted data. *REX Generator* translates given data into REX [13] format, which is an XML-based format for representing Document Object Model (DOM) events, which in turn refer to a data structure inside the browser and that can be interpreted by it. *Browser* interprets the REX message and updates its internal DOM presentation of the user interface content accordingly. *Subscriber* registers to *Scheduling Service* for updates about UI components, according to component IDs received from *View*. It also provides indirection between *View* and *Scheduling Service*, thus, enabling the implementation of the latter to be switched with minimal effort.

We have developed a proof-of-concept implementation for the architecture in Java. The size of the *SCIA Server* implementation is approximately 1,200 lines of source code. The framework is based on a number of open source software components. Components and their purpose are described in **Table 2**.

The application specific parts of the architecture implementation have been indicated in **Fig. 10**. The implementation of *SCIA DB* is mainly application specific, because it supports caching of the back-end system's data. The code in *Interpreter* is mostly application dependent, because the class processes the back-end system's messages. Code segments in *Controller* that use the *SCIA DB* are also application specific, because they need to pre-extract application dependent key components before data retrieval. Furthermore, UI components and scripts that handle the user input in *WWW Server* are application specific. Also, the *REX Generator* contains application specific code used to generate the XML content. The rest of the structure is application independent.

5. Example of Execution

Fig. 11 presents our implementation for when a client enters the network. After initial bootstrapping with SLP is finished (1) and an SIP session was established, the client sends a SIP SUBSCRIBE (2) message to the server, to establish a notification channel. *Controller* polls for subscriptions and responds with a SIP NOTIFY message containing the address of the first UI page (4).

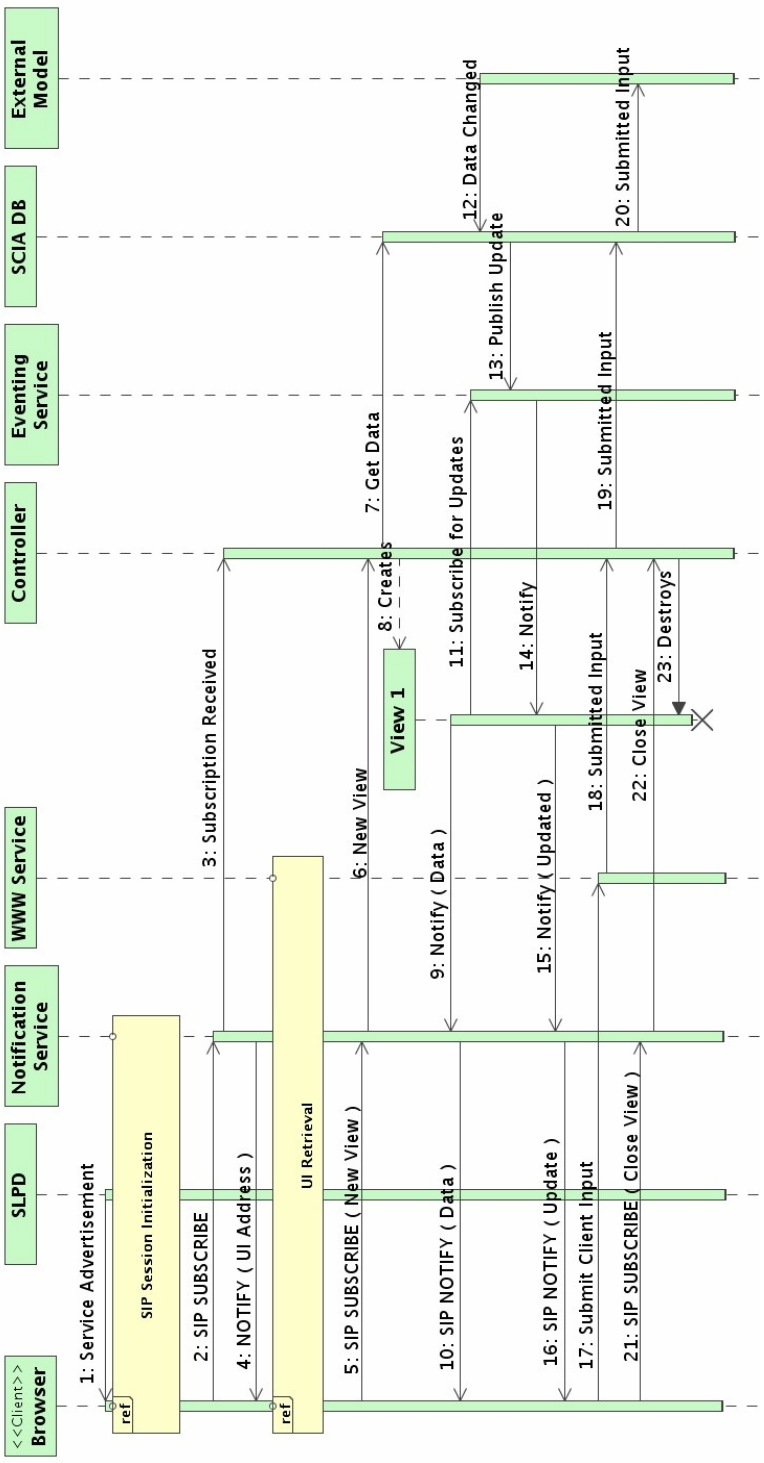

Fig. 11. Implementation functionality

After retrieving UI components, *Browser* sends their IDs to the server, encapsulated in a SUBSCRIBE refresh message via *Notification Service* (5-6.). *Controller* retrieves the data from *SCIA DB* based on IDs (7), and initializes a new *View* (8). Next, *View* sends its initial

content to *Browser* (9-10) and subscribes to updates to the content using *Scheduling Service* (11).

When data in *External Model* changes, it delivers an update to *SCIA DB* (12), which then publishes it using *Scheduling Service* (13). After matching IDs associated with the updated data to current subscriptions, *Scheduling Service* delivers the data to *View* (14), which then forwards it to *Browser* (15-16).

When a person using *Browser* submits input to the system, it is forwarded via *WWW Service* to *Controller* (17-18), which then delivers it to *SCIA DB* (19). Finally, after storing the input data locally, the database relays it to *External Model* (20.). *Browser* closes the *View* by issuing a Close View message (21-22), which causes *Controller* to delete related objects (23).

Table 2. Open-source software used in the framework

Software	Description
OpenSLP [14]	A C-language implementation of Service Location Protocol (SLP) [15]. Used in initial service bootstrapping. Augmented with support for broadcast service advertising.
oSIP [16]	A C-language implementation of Service Initialization Protocol (SIP) [17]. Used to implement *Notification Service*.
Fuego [18]	A Java-based scheduling middleware platform. Used as *Scheduling Service*. Enables event publishing as well as delivery and filtering of events.
X-Smiles browser [19]	A Java-based web browser that supports various XML languages, including XHTML and XForms [20] natively. Used as *Browser* to display UI components and their content.
Apache HTTP server [21]	A common web server. Used to host UI components at *WWW Server*.
Apache Tomcat [22]	A web container. Used as a container for servlets that pre-process client input at *WWW Server*.
MySQL [23]	An SQL Database management system. Used by *SCIA DB* to cache application specific data.
OpenSSL [24]	A C-language implementation of the TLS and DTLS protocols. Used by *Edge Proxy*.

6. Security

Connectivity and security problems introduced by the wireless network environment could be solved by requiring secure, persistent, client-initiated connections between clients and edge proxies. Our architecture is security protocol agnostic, in the sense that several protocols may be employed in this context, including TLS [25], DTLS [26], and HIP [27]. By default, TLS does not have a nonce for the initiator, but DTLS and HIP do have a nonce to prevent denial of service (DoS) attacks. The HIP protocol also has mobility and multi-homing support.

Fig. 12 illustrates the initialization of a secure session between the server and a client. In the context of **Fig. 11** this message exchange occurs during SIP Session Initialization. Initial bootstrapping is used to create identities for users and edge proxies (1-2). The identity provider (IDP) is trusted by all entities. After receiving an SLP service advertisement, and verifying its validity, the client starts a secure session with the edge proxy (4). The service advertisements may also be directly pushed to the terminal using 3G/B3G SIP MESSAGE, if this is supported.

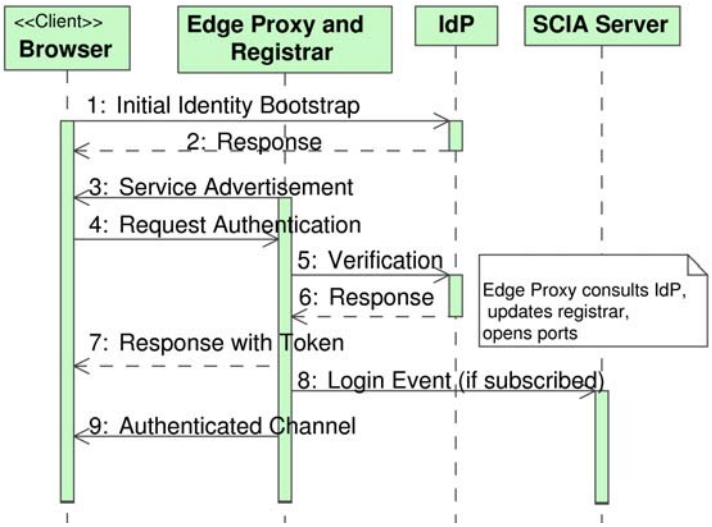

Fig. 12. Initiation of a secure channel

Successful establishment of a client-initiated secure session enables packet filtering and a client-initiated content channel. Hence, a secure session realizes the control plane of the system. Our baseline solution uses DTLS and SIP REGISTER messages. When the registrar receives a REGISTER message with verifiable credentials, it updates the current user contact information, reflecting the current client-serving edge proxy (4). The response to a successful registration includes a token originally created by the edge proxy, and signed by the registrar after verification of credentials. The client stores the token for later communications with the application server. At this point, the edge proxy configures the packet filtering mechanism, in order to enable the client-initiated content channels. After the secure channel was configured, an open channel is maintained by periodic keep-alive messages. When a failure is detected, the channel is closed and packet filtering is configured to block traffic.

The secure message exchange during UI Retrieval in **Fig. 11** is presented in **Fig. 13**. The client includes the token from the secure channel establishment phase in the HTTP request (1). The application server only replies to the request (4) if it verifies the token successfully with IDP (2-3).

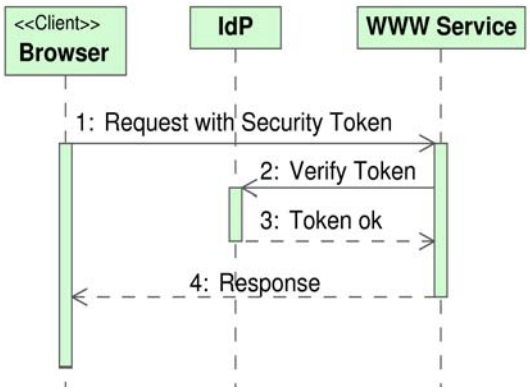

Fig. 13. UI retrieval via a secure channel

Direct signaling may occur in the same access network as the edge proxy or via an untrusted access network, such as the 3G or B3G access networks. The latter is needed to support push to an arbitrary location. The role of the edge proxy differs in these two cases. In the former, the edge proxy also configures the packet filtering rules for inbound and outbound traffic to and from the client. In the latter, the edge proxy configures rules for accessing content within the service domain.

7. Conclusion

The restrictions of the client-server architecture on the web are becoming a major obstacle to the implementation of more innovative services. In particular, there are cases where the server should have the right to initiate interactions, enabling the future Web to be a more powerful platform, as proposed in [28]. In this paper we aimed to tackle this issue by proposing a server-centric interaction model for wireless applications, where user interventions and input are minimized. Our model provides support for rich user interfaces via server initiated interactions, and reduces the amount of user activity required, by enabling the server to filter the data sent to clients residing in a wireless network. We have developed a proof-of-concept implementation of the architecture, which uses dummy implementations of an interface to *External Model* and *Interpreter* that processes all messages. It was designed in co-operation with a number of industrial companies to meet their future system's requirements.

In addition, we have also tackled the issue of limited user interface, and solved some of the related problems, by allowing partial updates of previously down-loaded web pages. This results in an Ajax-type user experience, where updates can be performed on-the-fly, and the user does not need to wait while they are loaded. Instead, the user interface remains available for other tasks the user is performing.

Finally, the mobile wireless environment introduces various security and connectivity problems. Regarding this, we described a security protocol agnostic approach to enable user authentication while maintaining sufficient user privacy. The solution is based on client-initiated persistent connections, which are authenticated. The introduction of security measures in this environment creates challenges for device and system performance. In the presented work, performance and connectivity issues were solved using channel authentication and a flow token to minimize state information related to the connection.

There are several means to continue our research. First, incompatibility with widely deployed regular browsers and the general web infrastructure is a serious handicap of the approach. Therefore, the most important aspect of future work is to devise an architecture where only minimal deviations from existing web architecture and browsers are implied, but including the opportunity for all enhancements we have proposed. In practice, a revised implementation, where Comet [7] is used for implementing push on top of the server architecture, and a system such as Sun Labs Lively Kernel [2] for implementing client-side, user-specific data processing for applications, would be a practical step in this direction.

Acknowledgments

This work has been supported by the Finnish Technology Agency (TEKES) and companies BookIT, Elisa, Finnair, Finnet, Nokia, and TietoEnator. The research has been performed in collaboration by University of Helsinki, Helsinki University of Technology, and Tampere

University of Technology.

References

[1] "AJAX: A new approach to web applications," http://adaptivepath.com/publications/essays/archives/000385.php.
[2] "Sun Labs Lively Kernel," http://research.sun.com/projects/Lively
[3] E. Bozdag, A. Mesbah and A. van Deursen, "A comparison of push and pull techniques for AJAX," Technical Report TUD-SERG-2007-016, Delft University of Technology, 2007.
[4] Webtide, http://www.mortbay.org.
[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," 1999.
[6] A. Russel, D. Davis, W. Greg and M. Nesbitt, "Bayeux Protocol - 1.0 draft 0," http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html.
[7] A. Russel, D. Davis, W. Greg and M. Smith, http://www.cometd.com.
[8] P. Sutton, R. Arkins and B. Segall, "Supporting disconnectedness-transparent information delivery for mobile and invisible computing," *In CCGRID '01*, page 277, Washington DC, USA, 2001.
[9] G. Cugola, E. D. Nitto and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Trans. Softw. Eng.*, 27(9):827-850, 2001.
[10] T. Berners-Lee and D. Connolly, "Hypertext markup language - 2.0," 1995.
[11] D. Flanagan, "JavaScript: The Definitive Guide," O'Reilly Media, 2006.
[12] A. Russel, "Comet definition," http://alex.dojotoolkit.org/?p=545.
[13] R. Berjon, "Remote Events for XML (REX) 1.0," Working Draft, W3C, Oct. 2006.
[14] OpenSLP homepage, http://www.openslp.org/.
[15] E. Guttman, C. Perkins, J. Veizades and M. Day, "Service location protocol, version 2," RFC 2608, IETF, June 1999.
[16] "The Gnu oSIP library," http://www.gnu.org/software/osip/.
[17] J. Rosenberg et al., "SIP: Session initiation protocol," RFC 3261, IETF, June 2002.
[18] "Fuego middleware," http://hoslab.cs.helsinki.fi/homepages/fuego-core/.
[19] Xsmiles homepage, http://www.xsmiles.org/.
[20] M. Dubinko, L. Klotz, R. Merrick and T. Raman, "XForms 1.0," W3C Recommendation, Oct. 2003.
[21] Apache HTTP server project, http://httpd.apache.org/.
[22] Apache Tomcat, http://tomcat.apache.org/.
[23] MySQL homepage, http://www.mysql.com/.
[24] OpenSSL project homepage, http://www.openssl.org/.
[25] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, IETF, Apr. 2006.
[26] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," RFC 4347, IETF, Apr. 2006.
[27] P. Nikander, J. Ylitalo and J. Wall, "Integrating Security, Mobility, and Multi-homing in a HIP way," *In Proceedings of Network and Distributed Systems Security Symposium*, San Diego, CA, USA, Feb. 2003.
[28] A. Taivalsaari, T. Mikkonen, D. Ingalls and K. Palacz, "Web Browser as an Application Platform: The Lively Kernel Experience," Technical Report TR-2008-175, Sun Microsystems Laboratories, Menlo Park, CA, USA, 2008.

Jussi Saarinen had his MSc degree from Tampere University of Technology, Tampere, Finland in 2006 and has worked as a researcher in the Institute of Software Systems, Tampere University of Technology for four years. His research topics include service interworking in heterogeneous networks, server initiated interactions in mobile networks, and model-driven development of mobile business processes. While conducting research in these fields, he has contributed to numerous papers as well as participated in the development of prototype software.

Tommi Mikkonen had his MSc in 1992, a Lic. Tech in 1995, and a Dr. Tech in 1999, in software engineering from Tampere University of Technology, Tampere, Finland, respectively. He is currently a professor of wireless and wired distributed applications in Tampere University of Technology, Tampere, Finland. Dr. Mikkonen's research interests lie in Web and mobile programming, software engineering for the Web, and distributed applications, in which he has authored numerous scientific articles. In addition to academic contacts, Professor Mikkonen also has close links to industry. Prior to joining the university, he worked as a Symbian chief architect at Nokia, and Dr. Mikkonen is currently a visiting professor at Sun Microsystems Laboratories, where he participates in the development of Sun Labs Lively Kernel, a browser-based system designed for implementation of interactive applications.

Sasu Tarkoma received his M.Sc. and Ph.D. degrees in Computer Science from Department of Computer Science, University of Helsinki. He is currently a professor of Department of Computer Science and Engineering, Helsinki University of Technology. He is Docent at the Faculty of Science, University of Helsinki. He has managed and participated in national and international research projects at University of Helsinki, Helsinki University of Technology (TKK), and Helsinki Institute for Information Technology (HIIT), respectively. He has over 100 publications, including 56 refereed scientific articles, and contributed to several books on mobile middleware. He has reviewed articles for many scientific journals, conferences, and workshops.

Jani Heikkinen is a Ph.D. student at Department of Computer Science and Engineering, Helsinki University of Technology (TKK). He obtained his Master of Science degree from this department in 2007. He has participated in the WeSAHMI and TrustInet projects, which were funded by Finnish National Technology Agency Tekes and industry. His research interests include distributed software systems, focusing on information system security.

Risto Pitkänen had his MSc. degree, 1997 and Dr Tech degree, 2006 from Tampere University of Technology. He researched and taught technologies related to distributed systems for many years at Tampere University of Technology. He has written several articles on formal methods and specification-driven development of distributed and embedded systems. At present, he is a software architect at Atostek Ltd, a software development company specializing in solving challenging problems.