WeSAHMI Software Architecture and Interface Description (D6,D7)

February 28, 2007

Abstract.

WeSAHMI Software Architecture and Interface Description document contains two Wesahmi deliverables D6 and D7. It describes the software architecture and interface design for a platform that enables development of highly interactive services for mobile devices. The document contains descriptions of server, browser, and security architectures. Additionally, it presents also SIP, Client, Eventing, and Database interfaces. Application specific components and interfaces are discussed separately as well as few minor open issues that are to be solved.

Contents

1 Introduction

This document defines the software architecture and interfaces designed by the WeSAHMI project for a platform that enables development highly interactive services for limited mobile devices. The platform supports separation of user interface and the actual content to two channels. This dual channel approach allows granular content updates in real-time thus enabling richer user experience than traditional approaches. Furthermore it enables more complex applications to be built over the platform. The Wesahmi Server is also designed to scale in order to support high-traffic solutions. Most of the internal elements can be run in parallel on separate nodes to balance load effectively.

The WeSAHMI use cases [7] identified two principal need for communication between the Finnair application server and the customers: pull- and push services. Both of these services to applications are provided through SIP. SIP enables clients to register to certain services. Once registered, clients can either pull information out from the application server, or the server can send asynchronous notifications to the clients. These notifications can carry application-specific data, e.g., in XML form. The notifications can be also be used to inform the customer that certain information would be available, and the customer can decide when she wants to pull the information to her device. One example could be a relatively large applcation payload, which waits for the customer's approval before being sent (through customer-initiated pull) to her device.

The WeSAHMI use cases identified information that is valuable for stakeholders. Moreover, the availability of the push service is valuable for stakeholders. In this document, architecture for secure push service over insecure wireless environment will be described.

This document is structured as follows. Section 2 presents a general overview of the document contents. In Section 6 we discuss SIP API and in Section 7 an API for client input is presented. Section 3 discusses the core server architecture and Section 4 discusses the client's browser design. Next, in Section 5 a security architecture is presented. Then, Section 10 discusses all application specific parts of the architecture design. Finally, Section 11 discusses few open issues that will be solved as the project progresses.

2 Overview

A general overview of Wesahmi Architecture is presented in Figure 1. The main components of the architecture are:

- Wesahmi Server,
- Browser (Client),
- Security Architecture, and
- WWW Server.

Wesahmi Server has a central role as relaying data from the External Model to Browser in Clients and vice versa. It uses Fuego eventing service [8] to deliver only relevant notifications to each Client. The main purpose of the Wesahmi Server is to provide a backbone for the whole platform. Client represents any client node using its X-Smiles browser [6] to connect to Wesahmi Server and WWW Server. The Client's purpose is to provide a dynamic user interface that can be updated by the Wesahmi Server when relevant information becomes available from the External Model that represents the existing real-world system to which the Wesahmi framework is connected. The Security Architecture is used to establish authentication and authorization between Clients and the Wesahmi Server. Furthermore it is used to protect integrity and secrecy of the SIP communication. We use an unmodified Apache WWW Server [1] to host user interface components and also to relay Client input to the Wesahmi Server which then delivers it to the External Model. The architecture components that are highly dependent on the External Model design need to be application specific. These components include the External Model API, Wesahmi Database, SOAP/REX Generator, and the user interface stored on the Web Server. This enables the framework to be adapted to support different External Models.

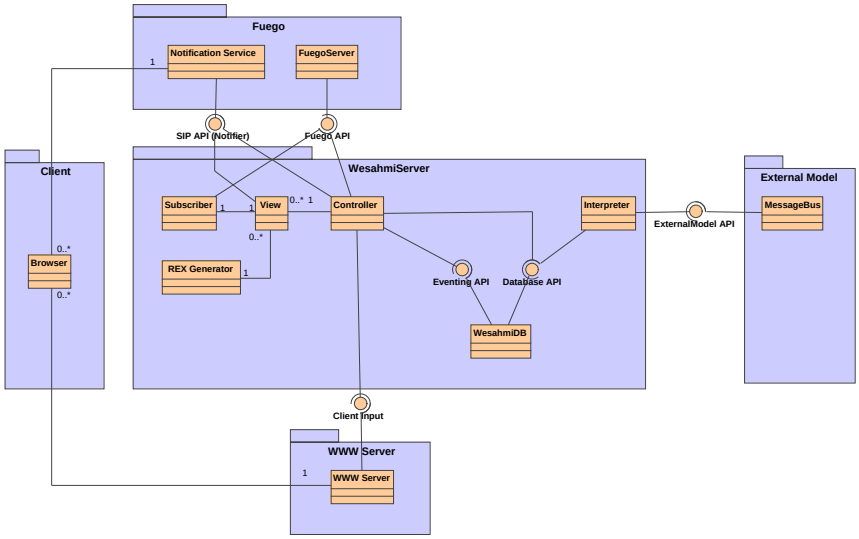

Figure 1: Wesahmi Architecture.

The architecture contains the following main interfaces:

- SIP APIs,
- Client API,
- Eventing API,
- Fuego API,

- Database API. and
- External Model API.

SIP APIs are used by Browser and the core server components to communicate using SIP SUBSCRIBE-NOTIFY framework. Client API is used to relay client input from the WWW Server to the Wesahmi Server. Eventing API is used by the Wesahmi Database to publish updates when changes in its state occur. The Controller that implements this API uses in turn Fuego API to perform the actual publication. Database API is used by Controller to retrieve data from the Wesahmi Database and also to store data. The API is used also by Interpreter to store data to the Wesahmi Database. External Model API is an application specific interface that enables the External Model to send messages to the Wesahmi Server and also to retrieve messages from it.

The data delivery push provided through SIP leverages security mechanisms on multiple layers. Authentication solution includes transport layer security protocol to authenticate networking component located at the trusted WeSAHMI domain. In addition, SIP registration procedure provides user agent authentication. Furthermore, application level security measures can be used to provide end-to-end security qualities.

3 Server Architecture

This section describes the core components of the Wesahmi Server architecture: Wesahmi Database, Controller, View, Subscriber, and REX Message Generator. They implement the logic which keeps track of each client's state information, delivers targeted updates, and forwards client's input to the External Model. A general usage scenario is presented in Figures 2 and 3.

3.1 Wesahmi Database

The Wesahmi Database operates as a local cache for data of the External Model, which is the operational database whose contents is the source of real life data. Wesahmi Database is based on MySQL database [5], enables persistent data storage, and implements Database API that provides methods for data retrieval and data updating. The API is presented in Section 9. Simple `ModelElement` class is used to contain name-value pairs.

Database is application specific component in the infrastructure and thus the implementation of both functions described in this section are dependent on the structure of the database.

3.2 Controller

Controller is a central element of the Server architecture. It is responsible for initializing the Wesahmi Database and SIP Notification service. It also establishes connection to the Fuego server and creates a `wesahmi` channel used to convey update messages from the Wesahmi Database to Subscriber instances. The Controller uses SIP API function `checkForSubscriptions` to poll for new subscriptions. When a subscription is received, it uses local `newClient` function to initialize a new Client instance and stores it locally. Then it uses SIP interface to send an initial notification to the client's browser containing the URI of the first user interface page on the Web Server. When the browser sends in a request for a new page, the Controller retrieves the View's initial contents from the WesahmiDB based on the tags in the request message. Controller then initializes a new View instance and gives it SIP interface, Client, the tags and the initial contents as parameters. Controller provides also an Eventing API for the Wesahmi Database. It provides means for publishing updates through the Fuego eventing system. A detailed description of the API is in Section 8.

3.3 Subscriber

The Subscriber encapsulates the Fuego interface. It creates notification subscriptions to the Fuego service and uses SIP API to deliver notifications to subscribers.

At initialization Subscriber sets EventConnection parameters and opens a new EventSession to the Fuego server. Then it uses the corresponding View's keys and tags to create Fuego subscription template vector, and creates a new subscription using the EventSession and the template vector.

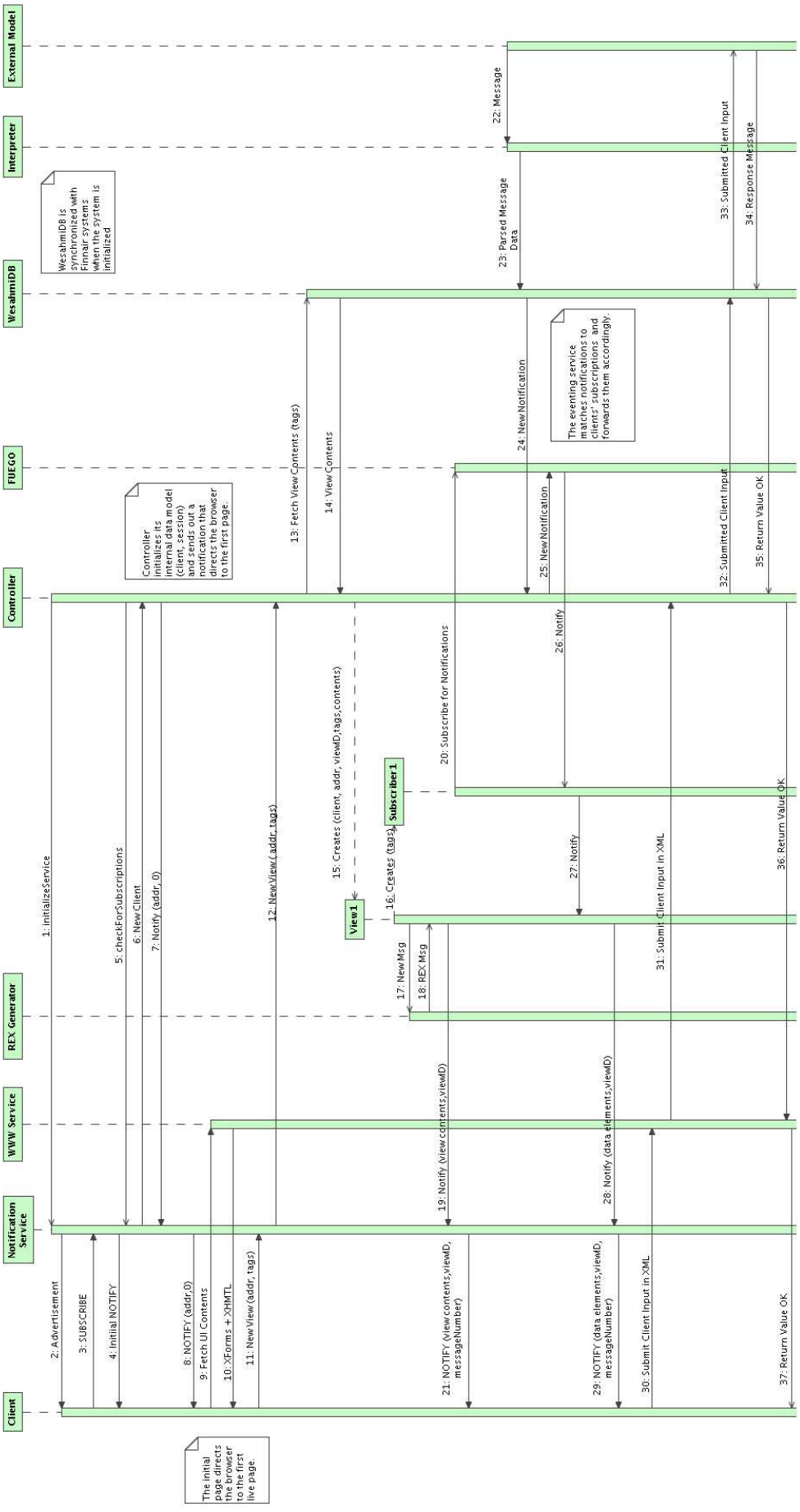

Figure 2: Operation of Wesahmi Server Architecture part 1/2.

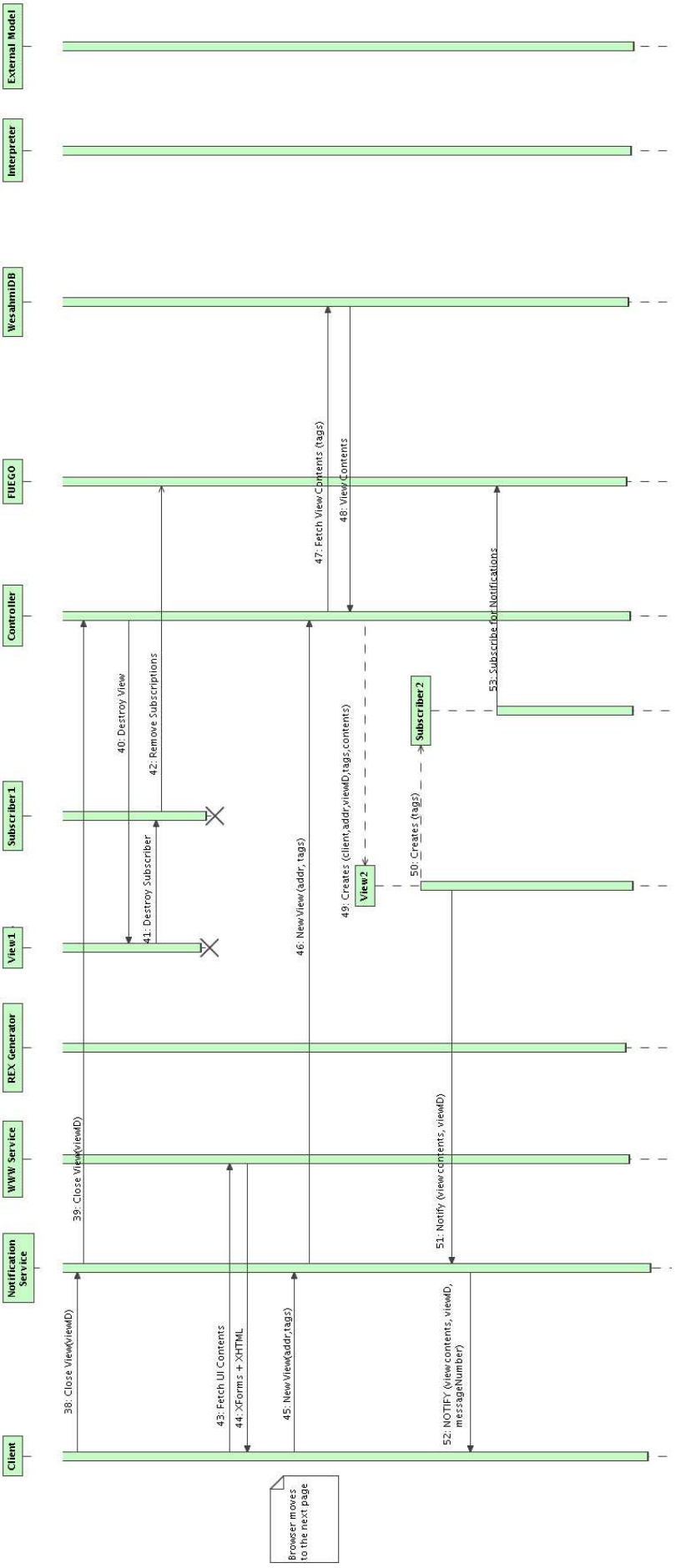

Figure 3: Operation of Wesahmi Server Architecture part 2/2.

Subscriber implements Fuego's `Notifiable` interface and thus also `public void notifyIncoming(Notification n)` method which is called every time Fuego receives a notification that matches the Subscriber's subscription. The method stores the contents of the notification into an array of `ModelElements` and then either forwards this array directly to the parent View or stores it into a buffer of received notifications. Subscriber provides method `boolean hasNextNotification()` that returns `true` if one or more notifications are buffered or `false` otherwise. Subscriber provides also method `ArrayList getNextNotification()` that returns the next array of `ModelElements` corresponding a buffered notification or `null` if the buffer is empty.

3.4 View

Each View instance in the Wesahmi Server represents one client specific open view to the system. View receives notifications, transforms and forwards them to the client browser. The View could also be extended to perform data transformation or filtering based on Client's profile.

When a View instance is initialized it receives as parameters Client, array of tags as `ModelElements`, and initial contents as an array of `ModelElements`. First, View initializes a Subscriber that makes notification subscriptions for Fuego on its behalf based on the tags. After this View checks for new notifications by calling the `hasNextNotification` and `getNextNotification` methods of the Subscriber. View has method `boolean notify(ArrayList notification)` which is called by the Subscriber to deliver new notifications. The parameter `notification` is an array of `ModelElements` and containing the new notification. The method returns `true` if it completes successfully and `false` otherwise. The method uses REX message generator to form a new REX-message based on the notification and then encapsulates it to a SOAP message. The SOAP message is then sent to the client's browser using the SIP API.

3.5 SOAP/REX Generator

The SOAP/REX Generator is an application specific component used by the Views. It creates the content of the notifications based on the provided data and interprets client input gotten via the messaging service.

Creating Messages

The View provides the data for the Generator as name-value pairs. The pairs are stored in a Hashtable. To generate the content, the View calls `public String generateMessage(Hashtable data)` method. The method returns a String object. The content is a SOAP message. The payload of the SOAP message is either a URL or REX message depending on the input data. Examples of messages are shown in Section 10.3.

Interpreting Client Input

The View calls `public Hashtable getData(String message)` to receive the client input as name-value pairs. The generator receives the name-value pairs within a SOAP message, which it parses and returns the pairs in a Hashtable.

4 Browser

The browser needs to support several technologies to be able to utilize the WeSAHMI messaging service. It must be able to register for the service according to the SLP advertisements, order data updates by data references, handle data update feeds, and make partial submissions. The required components and their relations to the other client side components are depicted in Figure 4. The general browser core components are not shown in the figure.

The interaction of the client side components and with the servers are shown in Figure 5. The browser components are discussed in more detail in the following subsections.

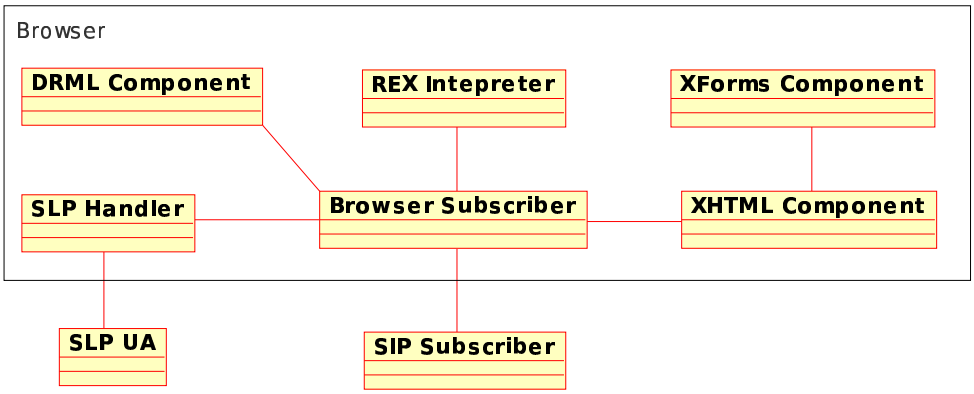

Figure 4: Required components on the browser.

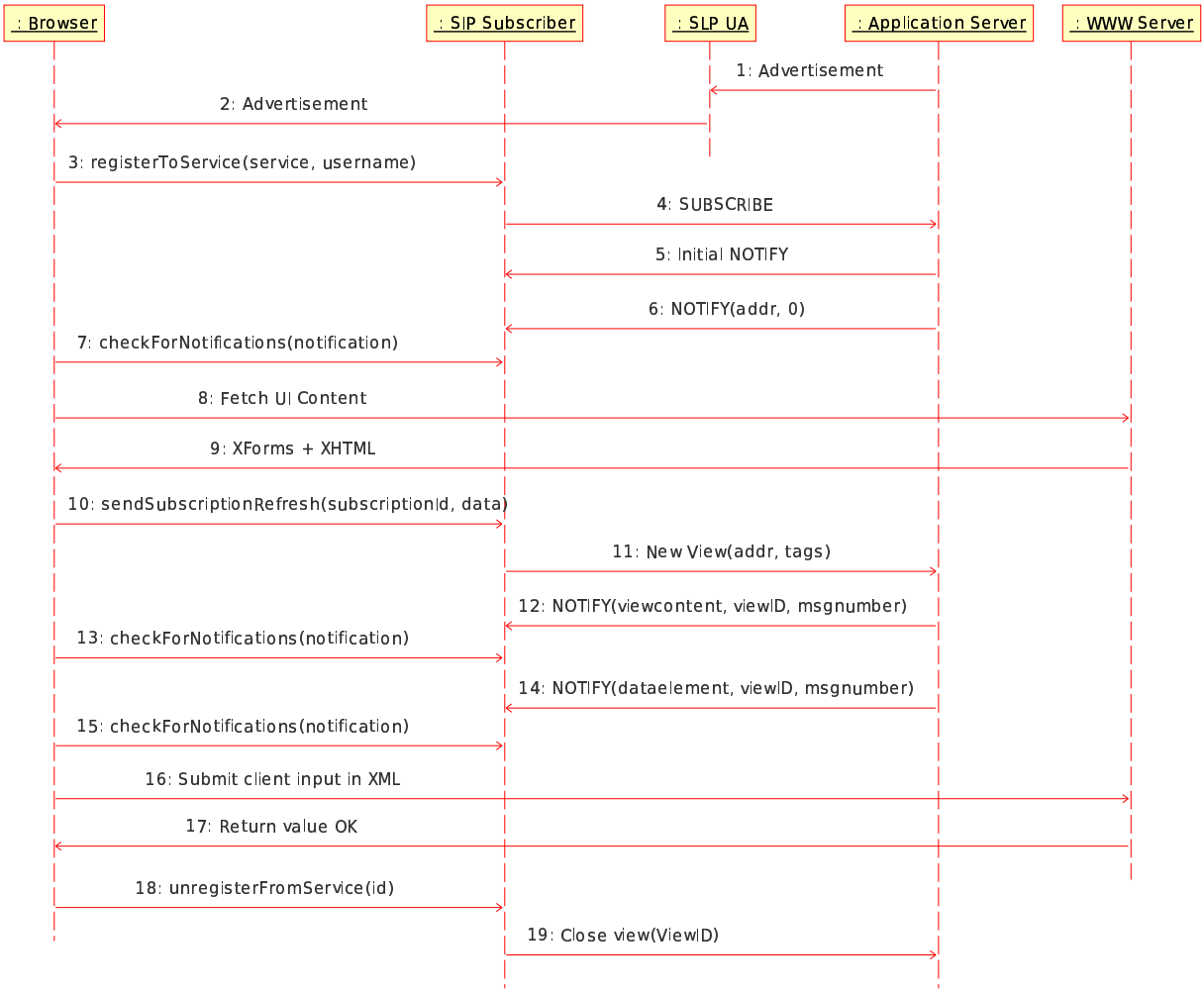

Figure 5: Interaction of the client side components.

4.1 Browser Subscriber

The Browser Subscriber implements the SIP Subscriber API. All the browser components uses the Browser Subscriber to communicate with the SIP Subscriber. When the Browser Subscriber has registered to the service, it polls the SIP Subscriber for the new notifications. The Browser Subscriber receives the notifications as SOAP messages which includes a REX [3] message or a URL. The URL points to a new document which browser is to open. The Browser Subscriber forwards the URL to the XHTML Component, which opens the document. The REX message, for one, is forwarded to the REX Intepreter.

4.2 SLP Handler

The browser receives an initial advertisement of the service from SLP UA. The SLP UA communicates with the browser via predefined socket. The SLP Handler listens the socket. When an advertisement

arrives, the Handler subscribes to the service via Browser Subscriber. The ad includes a service ID and a user name which are used for the registration to the service.

4.3 DRML Component

Data Reference Markup Language (DRML) specifies the content which is fetched from the data base for the UI through the messaging service. The DRML Component on the browser handles the DRML documents. It recognizes the data references in a document and orders the content from the messaging service via Browser Subscriber. See Appendix B for DRML specification.

4.4 REX Intepreter

As mentioned above, the data update notification contains a SOAP message, which identifies the document the update is targeted. The REX message within the SOAP message identifies the element within the document and the mutation event on it. The REX Intepreter modifies the document according to the REX message.

4.5 Submissions

The traditional HTTP POST method is used to submit complete forms to the Web Server. The Web Server forwards them to the Controller through Client API (cf. Section 7). It is also possible to submit smaller data fragments via the messaging service. The XForms Component submits the data to the Browser Subscriber, which creates name-value pairs from the data and sends them to the service within a SOAP message.

5 Security Architecture

This section describes security architecture of the data delivery push from trusted WeSAHMI environment to insecure wireless network environment. Whether the used access network includes third party address translators or network level access controllers allowing client-initiated connections, an additional networking element (Edge Proxy) is applied to enable secure data delivery push.

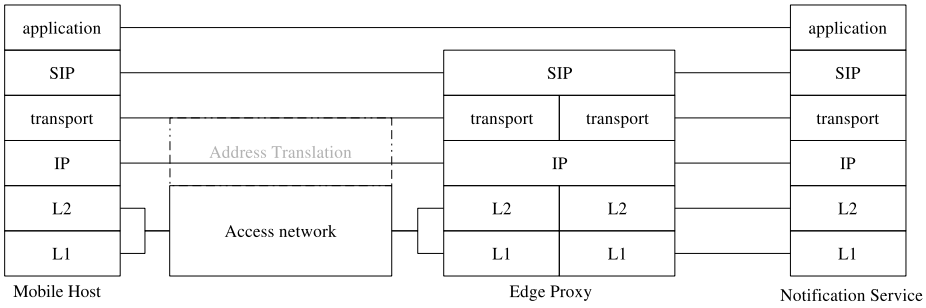

Figure 6: Layer view of the system

End-to-end considerations of this architecture include both the communication and security aspects. Figure 6 shows the case in which end-to-end communications take place on the application level. This is the case when address translator is placed in the access network. If translators do not exist, the level of end-to-end communications can take place at the lower level, preferably on the transport or network level.

This architecture leverages security mechanisms on multiple layers. Transport layer security protocol (TLS/DTLS) is used to authenticate networking element located at the trusted WeSAHMI domain. In addition, transport layer security protocol can also be used to authenticate the mobile host. Authentication of the mobile host is always initiated by the server side, in this case by the Edge Proxy. Authentication on this level is based on signature verification. Furthermore, SIP registration procedure provides user agent authentication achieved using a challenge-response protocol inside a confidential communication. Application level security measures can be used to provide end-to-end security qualities.

Shape of the traffic in the push system is asymmetric. Depending on the amount and frequency of incoming data from the notification service, congestion can occur on the additional networking element. Overloading of the additional networking elements in the network can cause decrease in system availability.

In this architecture, server-side security protocol processing is not needed on the mobile host. This decreases mobile host's processing load, yet increases processing load and amount of states in the additional networking element. On the other hand, the processing load of the additional networking element can be balanced over several physical hosts. Furthermore, the decrease of the mobile host's processing load decreases mobile host's energy consumption.

The additional networking element serves as a barrier against denial-of-service attacks. In this architecture, mobile hosts will not contact notification service directly. For example, ratio of communication attemps may be limited at the Edge Proxy. Coarse grained access control is best suited on lower network layers. Fine grained control mechanisms can be implemented on the signaling and application layer.

This architecture favors soft-state signaling and aims to avoid additional, inconsistent hard-states in network, which have to be explicitly cleaned up due to sudden UA disconnection or reboot. Thus, this architecture leverages recently proposed SIP outbound extension [4] that carries additional state along with requests and responses. In addition, this architecture encourages to push states that are due communication and security mechanisms to the edges of the system. In network, states are due the additional networking element (Edge Proxy) consisting of transport and security mechanisms. If these mechanisms of the additional networking element fail, the data delivery push is not available for the mobile hosts. Nonetheless, multiple networking elements can be deployed to decrease the possibility of lost notifications due the element failure.

5.1 Edge Proxy

Edge Proxy forwards incoming notification messages to mobile host's user agent over implicitly authenticated flows.

In accordance with SIP outbound extension [4], a flow is conceptually considered as a bidirectional stream of UDP/DTLS datagrams or a TCP/TLS connection. The creation of these flows requires UA authentication by logical SIP registrar.

A method to associate flows with incoming requests is by using a token. This token, denominated 'flow token', will be created at the edge proxy when the UA REGISTER message crosses the edge proxy and will contain enough information such as the origin and destination IP addresses and ports as well as the protocol in use as illustrated in Figure 7. The flow token will be added to the body of the REGISTER message in order to let the registrar to store it along with other information related to the UA.

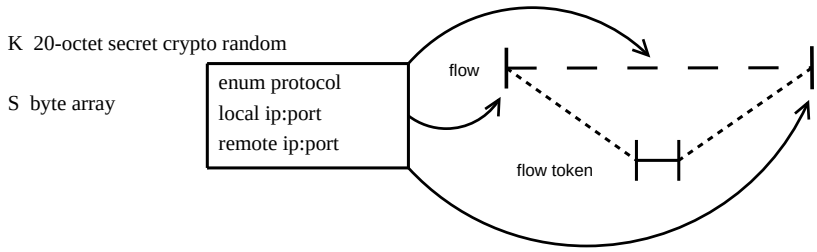

Figure 7: Flow token, eq. 1

$$flowtoken = BASE_{64}(HMAC_{K,SHA1-80}(S)||S) \quad (1)$$

A future notification message delivered to certain UA (identified by a SIP URI) will require a search into the registrar content that will return the contact information about where the message must be forwarded and also a respective flow token related to the UA that must be reached. In addition, a Path vector value is used to indicate to which Edge Proxy the incoming notification should be sent.

When the message reaches the Edge Proxy, it identifies the presence of a flow token from the SIP body and decodes it in order to know over which flow the incoming notification must be sent to reach the UA.

Other system elements are mobile host and notification service. Both of these include SIP user agent functionality that process stateful SIP transactions.

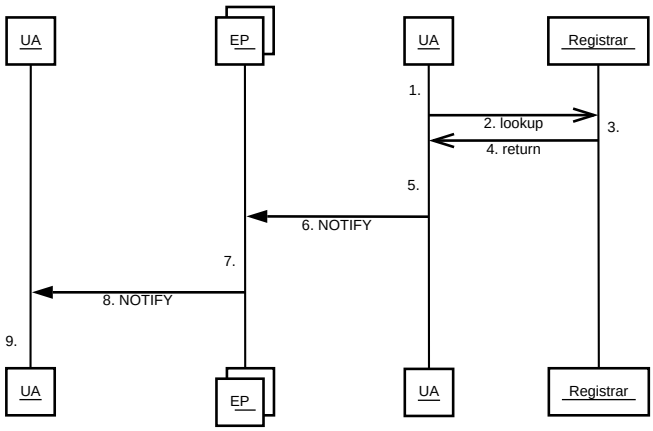

Figure 8: Notify over authenticated flow

5.2 Mobile Host

The mobile host as a SIP user agent (UA) can initiate a flow to Edge Proxy by sending REGISTER request to the registrar as shown in the step 1-5 in Figure 9. Then the registrar will challenge the mobile host for authentication. After successful registration indicated by receiving 200 OK response as shown in the step 6-9 in Figure 9, the mobile host sends STUN Binding requests over the same flow for sending SIP messages to keep the flow alive. This established and ongoing flow can later be used for secure push.

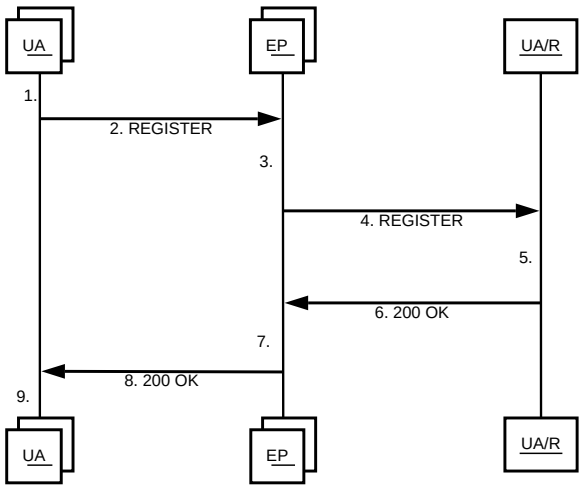

Figure 9: Authenticated flow establishment

Reliability is achieved by having the mobile host form multiple flows to the Edge Proxy. Each mobile host is pre-configured with a set of outbound-proxy-URI. For each outbound proxy URI in the set, the mobile node should send a REGISTER request. If any of these flows fails, the mobile node has to recover the flow by using backoff mechanism to avoid avalanche restart on the Edge Proxy.

5.3 Notification service

As illustrated in Figure 8 step 1-4, the notification service user agent fetches the contact address of the mobile host by querying the registrar. Step 5-9 shows the procedure of pushing NOTIFY request to the mobile host. The NOTIFY request is proxied to the Edge Proxy and then the Edge Proxy forwards it to the mobile host through the existing flow initiated by the mobile host before hand.

6 SIP API

The SIP Specific Event Notification extension [2] is used to provide a messaging service to client and server side applications. The applications can tell what notifications they are interested. A daemon on both

end points keeps track of applications and what events they are interested in. The daemon handles the SIP-message transfer, and passes the notification to application (e.g., a SOAP body). The fundamental concept is that the user-level applications are SIP unaware, i.e., they do not need to understand how the SIP protocol works. The API described in this specification achieves this.

6.1 Overall architecture

The SIP Event Notification Services supports by default PUSH-operations, that is, the content server can push data to the client. Figure 10 shows the operation of the SIP-based PUSH service. First the client must register to the service it is interested in. The client first registers with the local daemon, and the daemon sends a SUBSCRIBE message, which the content server acknowledges with a NOTIFY. After the client has registered, it can query the local daemon for messages. If a message from the content server has arrived, the daemon will deliver the message to the client application

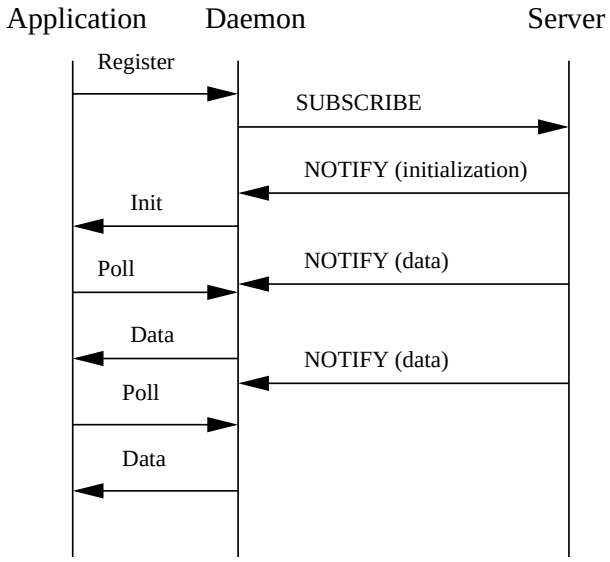

Figure 10: Operation of data PUSH.

Here the body on the NOTIFY is opaque to the SIP protocol. Any content can be delivered between the end points. Yet, in WeSAHMI we mainly focus on XML and SOAP messages.

The support for PULL-services is similar. The client first registers with the local daemon, and eventually with the content server. In order for the client to make a PULL-request, it must know what content to pull from the server. In our design, the content server can send descriptions of available content. Each content is described and identified with an *event ID*. When receiving these content or event notifications, the client can decide which content to fetch (pull) and when. For example, there could be some rather larger content available, and it is up to the client to decide whether it wants to receive it. For example, the client is connected to a low-speed GPRS network, and can decide that once the device is connected to a high-speed network, it will request the content to be delivered.

Again, the body of the event notification and the actual content data are opaque to the SIP stack. The *event ID* is included in the notification body and used by the application.

The logic of the PULL-service on the SIP protocol layer is presented in Figure 11. Here the content server sends a NOTIFY message which carries in the body a description of the available content, and a *event ID*. Once the client is ready to receive the cotent, it sends a (refresh) SUBSCRIBE message carrying the content *event ID*. This instructs the content server to deliver the requested content in a subsequent NOTIFY message.

6.2 SIP UA API on Client Side

Our SIP protocol stack in implemented in C. The existing interface can be used directly by C-language applications. Yet, in order to support Java-based applications, we have a Java library that communicates with the local daemon using sockets. This sections presents the API calls for this library.

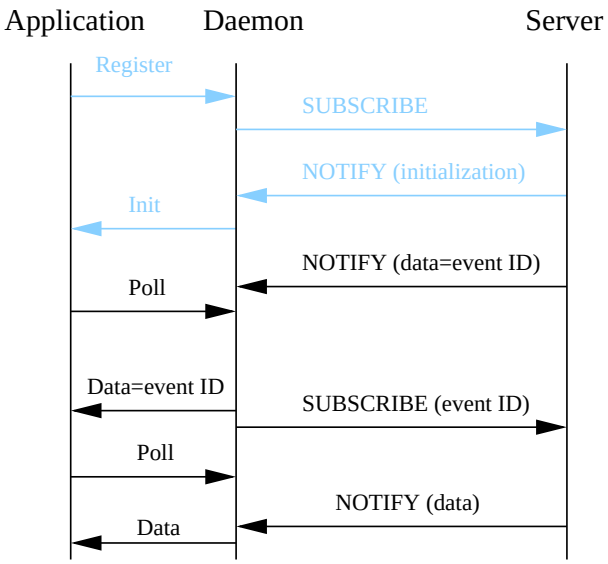

Figure 11: Operation of data PULL.

Subscriber-class

This class provides the methods for using the notification service to client. Client can use this class for receiving notifications from server. Here are the methods provided by the Subscriber class.

registerToService

Registers to the local notification client daemon. After the registration subscriber will receive notifications from the service it has subscribe.

Return value: int

Returns an identifier for registration or 0 if registration fails.

Arguments: String address, String service, String url

Address is a SIP-address used for subscription. Service is the service name that subscriber is interested. Url is the url of the service that subscriber is interested.

unregisterFromService

Unregisters from the local notification client daemon. After unregistration subscriber will not receive any notifications.

Return value: boolean

Returns true if unregistration succeeded.

Arguments: int id

Id is the identifier of the registration.

checkForNotifications

Checks if there is a notification waiting.

Return value: Notification notification

Returns an object containing information about the received notification. Null if there was no notification waiting.

Arguments: No arguments

fetchContent

Fetches the event data. Client can fetch the data when if it receives a notification telling that there is data waiting to be fetched at server side. Data will be received as a new notification. This is for event pull.

Return value: boolean

True if the query was sent to server.

Arguments: Notification notification

Notification is the object containing the notification data. This is the object received from checkForNotifications-method.

Notification-class

This class is just small class for holding information about received notifications. There is only one public method for the user.

getBody

With this method user can get the body of the received notification.

Return value: String

The body of the notification.

Arguments: No arguments

6.3 SIP UA API on Server Side

In order to support Java-based server side applications, with have implemented the interface between the application and the SIP daemon as a Java Native Interface (JNI). This sections presents the Java methods available to the server application programmer.

Notifier-class

This class provides the methods for using the notification service to server. Server can use this class for receiving subscriptions and sending notifications. Here are the methods provided by the Notifier-class.

initializeService

This method initializes the service. Must be called before any other methods.

Return value: boolean

Returns true if initialization succeeded.

Arguments: No arguments

checkForSubscriptions

Checks if there is a incoming subscription waiting.

Return value: Subscription

Returns an object containing the subscription data.

Arguments: No arguments

sendNotification

Sends a notification message with SOAP-message in a body to subscriber.

Return value: boolean

True if the sending of the notification succeeded.

Arguments: int subscriptionId, String body

SubscriptionId is the identifier for which the notification is sent. Body is the message body for the notification.

Notification-class

This class is just small class for holding information about received subscriptions. There are no methods, but few public member variables.

int id

Identifier for the subscription.

String sipAddress

The sip-address of the subscriber.

String service

The service name the subscriber wants to subscribe.

String body

The body of the subscription-message.

7 Client API

The Client API enables the Web Server to submit client input to the WeSAHMI messaging service and furthermore to the WesahmiDB. The communication is enabled with Java Remote Method Invocation (RMI) API.

The Web Server receives the client input as XForms instance. It parses the instance and forms name-value pairs from the input. The pairs are stored in a Hashtable. The Hashtable is sent to the Controller on WeSAHMI Server via sendClientInput method. The method is implemented in the Controller and can be called from the Web Server through Java RMI.

The Client API Specification

The Controller implements the `java.rmi.Remote` interface.

sendClientInput

Processes the HashTable containing the client input and transforms it into an array of `ModelElements`. It is then forwarded to the Wesahmi Database and eventually to the External Model.

Return value: boolean

Returns `true` if operation succeeds or `false` otherwise.

Arguments: HashTable input

Parameter `input` contains the client input.

8 Eventing API

Eventing API is provided by Controller and enables Wesahmi Database to publish updates of changes made to its data.

Eventing API Specification

publishUpdate

Constructs a Fuego notification based on the `ModelElements` given to it and publishes the notification.

Return value: boolean

Returns `true` if operation succeeds or `false` otherwise.

Arguments: ArrayList elements

Parameter `elements` is an array of `ModelElements` containing the data to be published.

9 Database API

Database API is provided by Wesahmi Database and enables Controller and Interpreter to retrieve data from the database and to store data to the database.

Database API Specification

getData

Enables data retrieval from the database based on given set of keys and tags.

Return value: ArrayList

Returns an array of `ModelElements` containing the requested data or `null` if data is not available or an error occurs.

Arguments: ArrayList keys, ArrayList tags

Parameter `keys` is an array of `ModelElements` containing key's names and values. They are used as keys in the SQL query. Parameter `tags` is an array of `ModelElements` containing field names. They are used as selected fields in the SQL query.

putData

Enables data storage to the database based on given set of keys and data elements.

Return value: boolean

Returns `true` if operation succeeds or `false` otherwise.

Arguments: ArrayList keys, ArrayList elements

Parameter `keys` is an array of `ModelElements` containing key's names and values. They are used as keys in the SQL query. Parameter `elements` is an array of `ModelElements` containing SQL table field names and values to be stored in them. They are used in a SQL query.

10 Application Specific Components and APIs

In order to make the Wesahmi architecture support various External Models, the External Model API, Wesahmi Database, SOAP/REX Generator, and the user interface stored in the Web Server need to be

application specific components.

10.1 External Model API

External Model API, i.e. the API between Wesahmi Server and Finnair systems, will be implemented as the Interpretor class. It transforms messages received from the external model to Wesahmi messages and vice versa. Both transformations have a FIFO message queue. Either the Interpretor class is modified depending on the format of the handled messages or we use a plugin based message transformation.

Further studies and material are needed to design implementation for the Finnair case. Contents of the incoming messages need to be analyzed and the format of the outgoing messages has to be agreed on.

ExternalModel API Specification

API provides the External Model means for sending messages to the Interpreter and to register as a message receiver.

sendMessage

Delivers a given message to the Interpreter for parsing.

Return value: void

Arguments: String message

Parameter `string` contains the incoming message as String object.

retrieveMessage

Retrieves the next message in the FIFO buffer of the Interpreter and removes it.

Return value: String

Arguments:

Return value contains the message as a String.

10.2 Database Design

The database design is definitely application specific since it depends on the data of the external model. As in case of any database, it requires a set of tables and primary (and secondary) keys to be set up based on the requirements set by the application. Also the code related to SQL queries need to be modified accordingly.

In case Finnair, we have initially decided to use one main table that contains all general information regarding the flights such as flight id, gate(s), boarding time, departure time etc. We are also planning to use one table per each flight to store the seat reservation status. Flight ID and departure time are used as a primary key to identify information concerning a certain flight.

10.3 Operation of the SOAP/REX Generator

The Generator API is discussed in detail in Section 3.5. In this section, the application specific functionalities are described. The generated message can contain either a URL or a REX message. An example of the SOAP message carrying URL:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
```

```
    <m:OpenLocation xmlns:m="Some-URI">
      <service>finnair</service>
      <url>http://www.finnair.fi/service.html</url>
    </m:OpenLocation>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Below is an example of creation of a REX message. The generator receives a name-value pair from the View, e.g., a new departure time:

```
(712021, 14:30)
```

The generator creates a markup, which is sent to the user interface. From the above name-value pair, a following XHTML element is created:

```
<p id="712021">14:30</p>
```

The element is transported into the UI via REX event. The REX contains the element itself and, in addition, target element in the document and type of the mutation event. To replace the existing departure time in the document, the REX would look like:

```
<rex xmlns='http://www.w3.org/ns/rex#'>
  <event target='id("712021")' name='DOMNodeRemoved'>
    <p id="712021">14:30</p>
  </event>
</rex>
```

The REX is sent within a SOAP message to the client side.

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:ContentChanged xmlns:m="Some-URI">
      <service>finnair</service>
      <rex xmlns='http://www.w3.org/ns/rex#'>
        <event target='id("712021")' name='DOMNodeRemoved'>
          <p id="712021">14:30</p>
        </event>
      </rex>
    </m:ContentChanged>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

10.4 Web Server

The Web server provides the user interfaces for the system. The Web Server implementation is the Apache Tomcat servlet container. The Web Server responds to the browser's page requests and handles the client input. The client input is forwarded to the Controller as discussed in Section 7.

The Web Server creates the user interfaces according to the page requests. The page request identifies the user and the task. The changing content for a UI is added on client side when the browser gets a corresponding notification. The notification is a respond to an order made by the browser. The browser places the order based on a DRML document which is embedded into a UI document.

11 Open Issues

The SIP transport for the PULL- and PUSH-service is currently based on polling done by the the client and server side applications. One idea that has come up during the design and implementation phase is

asynchronous operation. Currently the end points query their own daemon when they want for events. This provides a simple design for the implementer. Yet, it would be also useful to be able to provide asynchronous events. For example, the client application would be triggered in some way when a NOTIFY message arrives (whether the content is content data or just an event ID does not affect the SIP stack). Similarly, the server side application could receive notification when a client wants to register, or when the client sent a request as a SUBCRIBE indication that the server should send the client a given content. There are many ways to implement this functionality, therefore, we must carefully analyze the design options, and leave it as a future extension later this year.

References

[1] Apache website. At http://www.apache.org, February 2006.

[2] Roach A. B. Session initiation protocol (sip)-specific event notification. Request for Comments (Standards Track) 3265, Internet Engineering Task Force, June 2002.

[3] Robin Berjon. Remote Events for XML (REX) 1.0. Working Draft, W3C, October 2006.

[4] C. Jennings and R. Mahy. Managing client initiated connections in the session initiation protocol (SIP). Internet draft (work in progress), IETF, January 2007.

[5] Mysql website. At http://www.mysql.com, February 2006.

[6] K. Pihkala, M. Honkala, and P. Vuorimaa. A browser framework for hybrid xml documents. In *Internet and Multimedia Systems and Applications, IMSA 2002*. IMSA, August 2002.

[7] Mikko Pohja. WeSAHMI Use Cases. Technical report, WeSAHMI Project, April 2006.

[8] Sasu Tarkoma, Jaakko Kangasharju, Tancred Lindholm, and Kimmo Raatikainen. Fuego: Experiences with mobile data communication and synchronization. In *17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)* [8].

A Open source software and applying licenses

Table 1: List of Open Source Software and their Licenses.

Software	License	Usage
GNU oSIP Library	LGPL	Low layer SIP-library
eXosip - the eXtended osip Library	GPL	Higher layer SIP-library built on top of oSIP
OpenSLP	BSD	Bootstrapping of environment
X-Smiles browser	The Telecommunications Software and Multimedia Laboratory, Helsinki University of Tehcnology Software License, Version 1.0 (based on the Apache Software License Version 1.1)	Web browser
Apache Tomcat	Apache License, Version 2.0	Web server
OpenSSL	OpenSSL license	Open Source toolkit implementing the SSL and TLS protocols as well as a full-strength general purpose cryptography library.
Software Developed by Wesahmi	MIT	Deliverable

B Data Reference Markup Language (DRML)

B.1 Introduction

The Data Reference Markup Language (DRML) is an XML grammar representing data references in the WeSAHMI messaging service. The references indicate which data in the data base is displayed in a user interface (UI). The data is identified by ID both in the data base and in the UI. When user agent (UA) encounters DRML fragment, it registers itself into a WeSAHMI messaging service with the data references. The service will send updates to corresponding data when needed. An example of a complete DRML document below.

```
<dref xmlns="http://www.x-smiles.org/ns/drml">
  <item>701256</item>
  <item>432587</item>
  <item>673564</item>
</dref>
```

B.2 Structure of a DRML Document

The namespace of the DRML is: `http://www.x-smiles.org/ns/drml`.

All DRML documents must be contained within a `dref` element. DRML documents may be contained within other XML elements in other namespaces, but each DRML fragment must still have a `dref` element as its root.

The `dref` element

The `dref` element serves as the root container for the DRML document. It must be always present and it must contain one or more `item` elements. Otherwise it is in error and the user agent must ignore it entirely.

The `item` element

The `item` element must have a `dref` element as parent.

The `item` element must contain a text node. The text node is intended to be a single ID, which references to a data. DRML does not specify where the data is stored nor what the data actually is.

B.3 Processing DRML Documents

When the DOM of the document, which contains a DRML document, is created, the UA must order data updates from the WeSAHMI messaging service. In the order, the UA delivers the IDs, which are embedded into the `item` elements of the DRML document, to the service. The UA uses the SIP Subscriber class to register to the service.