WeSAHMI - Design of Experimentation (D9)

April 30, 2007

Abstract

This document describes an Airline Messaging System which can be run on WeSAHMI software. The Airline Messaging System is an experimental implementation of a possible real life system. This document discusses both hardware and software settings of the experimentation.

Contents

1 Introduction

This document describes an experimental design of an Airline Messaging System [6] which is implemented on the WeSAHMI software. The architecture of the WeSAHMI system is described in detail in WeSAHMI Software Architecture and Interface Description document [1]. This document focuses on application specific components of the system. The components are discussed in detail in Sections 3 and 4. In addition, the document describes an intended hardware and network environment of the experimentation as well as the required third party software to realize the experimentation.

The architecture of the WeSAHMI system is depicted in Figure 1. The main components of the architecture are:

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

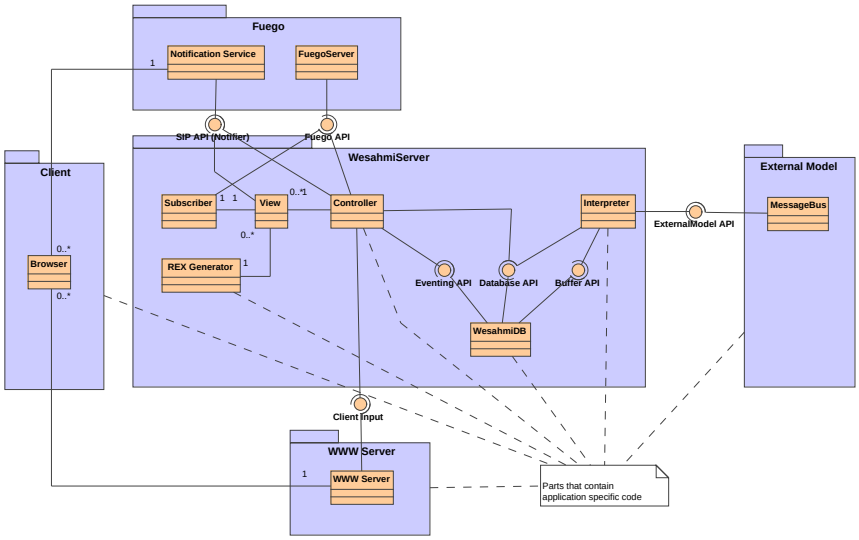

Figure 1: WeSAHMI Architecture.

WeSAHMI Server has a central role as relaying data from the External Model to a Browser in the Client and vice versa. It uses Fuego eventing service [7] 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 the X-Smiles browser [5] 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 [2] 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 (cf. Figure 1). 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.

2 Environment

To demonstrate the system, at least two computer are needed. One for client software and the other for the server side software. Figure 2, for one, represents a demo setting for two users where Web server is separated from the rest of server side software. Our experimentation is equivalent to the setting in the Figure 2. The following subsections describe the hardware environment of the experiment and the third party software on which the system is run. The application specific components, which are developed during the WeSAHMI project, are discussed in the following sections.

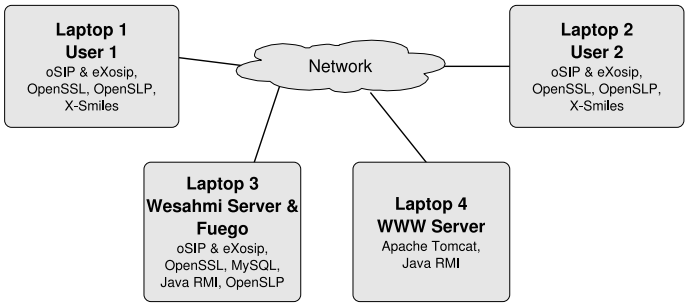

Figure 2: The software components on the laptops in the demo.

2.1 Hardware

The hardware of the demo setting is comprised of portable computers. Both server-side and client-side applications are executed on laptops. The Web server can be either on a separate computer as in Figure 2 or on the same one with the system itself. Each client must have its own computer. The computers are connected with each other via WLAN network. The computers are in Ad-Hoc mode on the network.

2.2 Software

This section describes the software used in the system. The messaging, the SIP communication, the secure connections, the Web server etc. are implemented with third party software and extended to fulfill the requirements of the system. The computers are running on Linux operating system. Both server and client side requires a Java virtual machine version 1.5. In addition, some components are implemented in C programming language. The software components are discussed below and their locations in the experimentation are depicted in Figure 2.

Fuego

Fuego is a Java based middleware platform implementing the publish/subscribe paradigm [7]. It is directed mainly for mobile devices. In WeSAHMI project Fuego is used for filtering data updates from the WesahmiDB to clients. It also enables distribution of the server architecture.

oSIP and eXosip

The Session Initiation Protocol (SIP) is an application-layer control (signalling) protocol for handling sessions on Internet communication. oSIP is a low level SIP-library written in C language. It provides functions for building and parsing SIP-messages. eXosip is a higher-level SIP-library built on top of oSIP. It provides functions for easily setting up, modifying and ending sessions. eXosip also supports SUBSCRIBE/NOTIFY-messages used in project.

OpenSSL

OpenSSL is an open source toolkit implementing the SSL and TLS protocols as well as a full-strength general purpose cryptography library. In WeSAHMI project, OpenSSL is used to provide TLS and DTLS support for the eXosip2 SIP-library.

MySQL

MySQL is an open source database management system. It supports database queries in Structured Query Language (SQL) and provides support libraries for multiple programming languages including Java. In the WeSAHMI project experimentation, MySQL is used in the WesahmiDB module as a cache for flight and client related information.

Java RMI

Java Remote Method Invocation (RMI) is a Java API that provides means for accessing objects remotely. It enables servers to register references for their objects to a RMI registry from where clients may discover them. WeSAHMI project experimentation uses RMI to enable servlets hosted on the web server to communicate with the server implementation.

OpenSLP

OpenSLP[4] is an open source C language implementation of SLPv2 defined by IETF in [3]. SLPv2 is a protocol that enables clients to discover service locations dynamically either directly or through service directories. The implementation of OpenSLP used in the WeSAHMI project has been modified to provide Passive (service) Discovery (PD) functionality. It enables the WeSAHMI server to advertise the SIP service with broad/multicast messages and thus clients can discover it by only listening to these advertisements.

Apache Tomcat

Tomcat is a Web server that supports Java Servlets and JavaServer Pages (JSP). Both of them allow to add dynamic content to a Web server using the Java platform. In the WeSAHMI project, we are using servlets to process user input and create Web pages. The servlets use system's client API to submit the input for the system.

X-Smiles

X-Smiles is an open source XML browser. It supports several XML languages and their combinations natively. In this project, we utilized the support of XHTML, XForms, Scalable Vector Graphics (SVG), and Cascading Style Sheets (CSS). In addition to the existing features of X-Smiles, we have extended it for the project. The extensions are discussed in Section 4.

3 Server-side Components

3.1 Database

WesahmiDB module is a cache for application specific data in WeSAHMI architecture. Its main function is to enable timely retrieval of data for new clients. It is based on MySQL database and has an API that supports both data retrieval and storage. The actual mechanism for synchronizing the WesahmiDB with the External Model is left open due to the fact that it is highly dependent on the APIs provided by the Finnair backend system, which we do not have access to. Instead, we assume that the state of the module is initially synchronized with the External Module when the experimentation is demonstrated. Database tables that contain the actual data are discussed in Section 3.2 and the functionality of the Database API implemented by the application specific code in WesahmiDB is presented in Section3.3. Data is identified in user interface and in the database with specific identifiers that are listed in Table 1.

Table 1: List of Data IDs Used in the Experimentation.

Data ID	Description
flightNumber	A number ID for each flight, does not identify a flight alone. Used in combination with sdt.
sdt	Scheduled Departure Time; fixed original time of departure used to identify flights
route	Description of the flight, consists of the departure city and destination city
gate	Name of the departure gate
boardingTime	The time when the boarding starts
edt	Estimated Departure Time; the actual time of departure, intially same as sdt
clientID	Identifies each client globally and uniquely
baggage	The number of baggage for each client
seat	The identifier of any seat in a plane
clientSeat	The identifier of the client's seat (for messages)
reserved	The reservation status; boolean value as an integer.

3.2 Database Tables

Data stored in the database is divided to four tables: flights, flight_boarding, flight_seats, and client_baggage. This section describes their contents in detail. Each table field is defined in terms of Field Name, Datatype, NVA, Key Type, and Default Value. Where NVA is an abbreviation for "Null Value Allowed" that indicates whether the field can have value NULL or not.

The contents of table flights is listed in Table 2. Field flightID contains an unique identifier for each flight matching a composite key consisting of flight number, stored in field flightNumber, and scheduled departure time, stored in field sdt. An example table is presented in Table 3.

Table 2: List of Fields in Table flights.

Field Name	Datatype	NVA	Key Type	Default Value
flightID	int(10) unsigned	no	unique	
flightNumber	varchar(10)	no	primary	
sdt	datetime	no	primary	

Table 3: Example Contents of Table flights.

flightID	flightNumber	sdt
1	FID1	2007-02-01 15:00:00
2	FID2	2007-02-01 20:00:00

The contents of table flight_boarding is listed in Table 4. Field flightID contains an unique key for data associated with each separate flight. Field route contains a description of the flight: the departure city and the destination city. Field gate contains the name of the gate from which the flight is boarded. Field boardingTime contains the date and time of the flight boarding in SQL DATETIME format. Field edt contains the date and time of the estimated departure time of the flight in the same format. An example table is presented in Table 5.

Table 4: List of Fields in Table flight_boarding.

Field Name	Datatype	NVA	Key Type	Default Value
flightID	int(10) unsigned	no	primary	
route	varchar(60)	no		
gate	varchar(10)	yes		null

boardingTime	datetime	yes		null
edt	datetime	yes		null

Table 5: Example Contents of Table flight_boarding.

flightID	route	gate	boardingTime	edt
1	helsinki - stockholm	G14	2007-02-01 13:40:00	2007-02-09 22:00:00
2	helsinki - madrid	G21	2007-02-01 14:50:00	2007-02-01 19:30:00

The contents of table `flight_seats` is listed in Table 6. Fields `flightID` and `seat` contain a combination key for data associated with a seat on a certain flight. Field `reserved` contains a boolean value indicating the reservation status of the seat. Field `clientID` contains an identifier of the client whom the seat is reserved to. An example table is presented in Table 7.

Table 6: List of Fields in Table flight_seats.

Field Name	Datatype	NVA	Key Type	Default Value
flightID	int(10) unsigned	no	primary	
seat	varchar(10)	no	primary	
reserved	tinyint(1)	yes		null
clientID	varchar(60)	yes		null

Table 7: Example Contents of Table flight_seats.

flightID	seat	reserved	clientID
1	A1	1	14528
1	B2	0	null
1	C3	0	null
1	D4	1	18093

Database table `client_baggage` contents are listed in Table 8. Fields `flightID` and `clientID` contain a combination key identifying each client on a certain flight whose baggage information is currently stored in the cache. Field `baggage` contains an integer value indicating the number of baggage the client has handed over at the baggage drop counter. An example table is presented in Table 9.

Table 8: List of Fields in Table client_baggage.

Field Name	Datatype	NVA	Key Type	Default Value
flightID	int(10) unsigned	no	primary	
clientID	varchar(60)	no	primary	
baggage	int(10) unsigned	yes		null

Table 9: Example Contents of Table client_baggage.

flightID	clientID	baggage
1	14528	1
2	14529	null

3.3 Database API

This section presents the implementation specific functionality of a Database API. It is implemented by the WesahmiDB module and provides means for retrieving data from the database with method `getData` as well as updating data in it with method `putData`. The API uses Java ArrayLists of Wesahmi specific container classes `ModelElements` to carry data in both directions. Each `ModelElement` has two fields: `String name`, which is holds the description of the data element and `String value`, which holds the value of the data element.

getData

Enables data retrieval from the database for a given set of keys and tags. The tags are an Array of `ModelElements` that have null `value` field but contain data IDs in their `name` fields

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.

The method's functionality is described in the following pseudocode:

```
select flightID from table flights
where flightNumber and sdt equal keys;

if( tags contains "seat" ){
select seat, reserved from table flight_seats
where table.flightID equals flightID;
return Array(seat, reserved)

}else if( tags contains "clientSeat" ) {
select seat from table flight_seats
where table.flightID equals flightID and table.clientID equals keys(clientID);
return Array(seat)

}else {
while( curTag in tags ){
select curTag from tables flights, flight_boarding
where flightID equals flights.flightID and
flights.flightID equals flight_boarding.flightID;
add curTag to resultArray
}
return resultArray
}
```

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

The method's functionality is described in the following pseudocode:

```
select flightID from table flights
where flightNumber and sdt equal keys;

if( elements contain "clientSeat" ){
update table flight_seats set table.reserved to
elements(reserved) and table.clientID to
elements(clientID) where table.flightID equals
flightID and table.seat equals elements(seat);

}else if( elements contain "baggage" ){
update table client_baggage set table.baggage to
elements(baggage) where table.flightID equals
flightID and table.clientID equals keys(clientID);

}else if (elements contain "newFlight" ){
while( elements contain curElement ){
insert into tables flights and flights_boarding (table.element)
values (elements.element);
}

}else{
update tables flights and flights_boarding set
table.element to elements.element
where flights.flightID equals flightID and
flights.flightID equals flight_boarding.flightID;
}

publish updated elements with Fuego
```

3.4 Controller

The Controller is a central entity in the Wesahmi Server architecture handling all incoming messages and relaying information between elements such as Views and the WesahmiDB. It is also responsible for handling client input and storing it into WesahmiDB. Due to the Database API implementation presented in Section 3.3, the Controller needs to be aware of the possible keys in incoming messages. Therefore it needs to identify ModelElements named `flightNumber`, `sdt` (scheduled departure time), and `clientID` and give them to the `putData` or `getData` methods in ArrayList `keys`.

3.5 External Model

The External Model in the Wesahmi architecture represents the system backend to which the rest of the architecture is connected to. It is simulated in the WeSAHMI experimentation by a command line tool. The tool enables user to send XML messages to the Interpreter either by giving their contents on the command line or by giving a path to a file containing several messages.

The tool is called `extModel` and it takes two command line switches `-m` and `-f`. The switch `-m` must be followed by space separated list of element element-value pairs. Each given pair is separated and stored in to a DOM node as a child for a `message` node. The DOM tree is then

written out as XML and given to the Interpreter as `message` parameter of `sendMessage` method. Command syntax:

```
extModel -m element1=value1 element2=value2 ...
```

The switch `-f` must be followed by a single file path addressing the file that contains the messages in XML format. The whole file is read to a DOM tree and each of the `message` nodes is then transformed to XML and sent to the Interpreter using `sendMessage` method in order but with random delays. Command syntax:

```
extModel -f messages.txt
```

The file containing the messages is of the following syntax:

```
<message>
<element1>value1</element1>
<element2>value2</element2>
...
</message>
<message>
...
</message>
```

3.6 Interpreter

The Interpreter functions as the mediator between the External Model and the Wesahmi Server implementation. It interprets XML messages received from the External Model to ArrayLists of ModelElements and uses the Database API to store them to the cache provided by the WesahmiDB module. It also formulates XML messages based on the ArrayLists of ModelElements in order to allow client input to be delivered to the External Model. Furthermore the Interpreter uses FIFO buffer to store outgoing messages. The XML messages accepted and formed by the Interpreter are of the following form:

```
<message>
<element1>value1</element1>
<element2>value2</element2>
...
</message>
```

3.6.1 ExternalModel API Implementation

API allows XML messages to be sent to the Interpreter as well as message retrieval from it.

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.

The Interpreter uses DOM parser to formulate a DOM tree from the XML message. The DOM tree is then used to formulate an ArrayList of ModelElements that can be stored to the WesahmiDB cache using the `putData` method of the Database API. The Interpreter needs therefore to be able to identify keys `flightNumber`, `sdt`, and `clientID`. The method has no return value.

retrieveMessage

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

Return value: String

Arguments:

Return value contains the next message of the buffer as a String.

3.6.2 Buffer API

API provides a method for adding outgoing messages to the Interpreter's FIFO buffer.

bufferMessage

Stores given message in XML format to the FIFO buffer of outgoing messages.

Return value: Boolean

Arguments: ArrayList message

Parameter message contains the contents of the outgoing message as an ArrayList of ModelElements.

The Interpreter formulates a DOM tree based on the input parameter and writes an XML message based on the tree. A String containing the XML message is then added to the end of the FIFO buffer. The method returns true on success or false otherwise.

3.7 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

Fuego provides the notifications for the View as name-value pairs. The View delivers the pairs further to the Generator, which generates the SOAP messages. To generate the content, the View calls `public String generateMessage(Hashtable data)` method on the Generator. The argument of the method is a Java Hashtable which contains the name-value pairs. The method returns a String object which is the SOAP message. The payload of the SOAP message is either a URL or REX message depending on the input data. An example of the SOAP message carrying URL:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <wes:OpenLocation xmlns:wes="http://www.tml.hut.fi/Research/wesahmi">
      <wes:service>finnair</wes:service>
      <url>http://www.tkk.fi</url>
    </wes:OpenLocation>
  </env:Body>
</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:

```
(edt, 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="edt">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("edt")' name='DOMNodeRemoved'>
    <p id="edt">14:30</p>
  </event>
</rex>
```

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

```
<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>
      <rex xmlns='http://www.w3.org/ns/rex#'>
        <event target='id("edt")' name='DOMNodeRemoved'>
          <p id="edt">14:30</p>
        </event>
      </rex>
    </wes:ContentChanged>
  </env:Body>
</env:Envelope>
```

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.

3.8 Web Server

The Web server provides the user interfaces for the system. The 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 of the system via client API.

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.

4 X-Smiles Browser

4.1 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 WeSAHMI Architecture document for DRML specification.

4.2 REX Interpreter

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 Interpreter parses the REX messages and modifies the document according to the event.

5 Summary

This document introduces the Airline Messaging System implementation on the WeSAHMI software. We have defined both the hardware and the software which are required to demonstrate the use case implementation. To fully utilize all the capabilities of the software, one needs at least two client side computers to simulate multi-user system. Also, we propose to separate the WWW Server from the Application Server because it seems to be a current industry practice. That is,

at least four portable computers are needed to demonstrate the system. It is noteworthy that along with the WeSAHMI software and application specific components, a number of third party software is needed in the system. They are also discussed in this document.

References

[1] WeSAHMI Software Architecture and Interface Description. Technical report, WeSAHMI Project, 2007.

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

[3] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, IETF, June 1999.

[4] OpenSLP Project Group website. At `http://www.openslp.org`, April 2004.

[5] 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.

[6] Mikko Pohja and Matti Alanne. WeSAHMI Messaging System. Technical report, WeSAHMI Project, December 2006.

[7] 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)*.

A Open source software and applying licenses

Table 10: 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
Fuego	MIT	Eventing system
MySQL	GPL	Database management system