Project CS 2008
Product Report

Amir Monshi   Christian Axelsson   Erdem Aksu   Jing Yao
Johan Vikman   Karl Sundequist Blomdahl   Muneeb Anwar
Niklas Ronnblom   Xiaoming Yu   Xiaoyan Ni
Yunyun Zhu   Zou Hanzheng

January 16, 2009

Abstract

During the autumn of 2008 a group of 12 computer science students at Uppsala University taking the Project CS course set out in cooperation with Mobile Arts and Ericsson to develop a next generation instant messaging system for mobile networks on top of Ericssons IMS or ”IP Multimedia Subsystem” platform. Not did they only do that but in the end they also implemented a framework for Erlang development on this platform. This document contains technical details about the developed product.

Contents

1 Introduction **3**
1.1 Original Product Proposal 3
1.2 Final Product – what we ended up with 3

2 Preliminaries **4**
2.1 Concepts 4
2.2 Glossary 4
2.3 Development tools / languages 6

3 IMS **7**
3.1 Architecture 8

4 Application Server (AS) **8**
4.1 System descriptions 9
4.2 Design / Architecture 10
4.2.1 API 11
4.2.2 SIP 11
4.2.3 Diameter / Sh 13
4.2.4 SMTP 13
4.2.5 Web front-end 14
4.3 Implementation details 14
4.3.1 API 15
4.3.2 SIP 16
4.3.3 Diameter / Sh 18
4.3.4 SMTP 18
4.3.5 Web front-end 18
4.4 Testing 20
4.4.1 Distributed development model 20
4.4.2 EUnit (Unit Testing) 20
4.4.3 OpenIMS Core + UCT IMS Client (System Testing) 21

5 Diameter / Sh **22**
5.1 Requirements 22
5.2 Design / Architecture 23
5.3 System descriptions 23
5.4 Functionalities / Features 24
5.5 Implementation details 25

6 Messaging Application Server (MAS) **26**
6.1 Requirements 26
6.2 Design / Architecture 27
6.2.1 Originating MAS 28
6.2.2 Terminating MAS 29
6.3 System descriptions 31
6.4 Functionalities / Features 32
6.5 Implementation details 33
6.5.1 User Configuration 33
6.5.2 Web front-end 34

6.5.3 Testing 34
6.6 Mobile Client 35
6.6.1 Requirements & System descriptions 35
6.6.2 Design 35
6.6.3 Functionalities / Features 37
6.6.4 Implementation details 37

7 Problems Issues **38**
7.1 Insufficient testing 38
7.2 Delayed Access to IMS environment 38

8 Known Unresolved Issues **38**
8.1 Use defensive programming in the AS API 38
8.2 Unfinished work in the MAS 39
8.3 Delivery Receipts 39

9 Conclusion and future work **39**
9.1 Extend the SIP/SIMPLE stack 39
9.2 XCAP 39
9.3 Spam filter for MAS 39
9.4 User Configuration security and functionality 39
9.5 Security 40
9.6 Removing unused tables from Configuration DB 40

A Installation / Upgrade Instructions **40**
A.1 Erlang installation 40
A.1.1 Compiling from source code 40
A.2 AS installation and upgrading instructions 41
A.2.1 Compiling the AS 41
A.3 Web front-end for administration setup instructions 43
A.4 MAS Installation and Upgrading 43
A.4.1 Installing 43
A.4.2 Upgrading 44
A.4.3 Configuration files 45
A.4.4 MAS Web Front-end 45
A.4.5 User configuration Manual 46

B User Manual **47**
B.1 Application Server API 47
B.1.1 ims 48
B.1.2 uac_email 48
B.1.3 uac_message 48
B.1.4 uas_message 49
B.2 Source Code documentation 50

1 Introduction

The course primary focus was to get experience of a larger scale project than what is common in the usual courses given by the University of Uppsala. This was done by giving the students a large project in cooperation with real companies, in this case Ericsson Research and Mobile Arts.

1.1 Original Product Proposal

Our original product proposal was to implement an instant messaging service in the Erlang programming language on top of Ericssons IMS platform. The IMS platform was supplied by Ericsson and the specifications for the instant messaging service was provided by Mobile Arts. A requirement from Mobile Arts worth noting was that the server side software should be written in Erlang while the existing IMS software from Ericsson was all written in Java. This way developers gets an alternative to the Java environment. The features that we were required to implement in the instant messaging service were:

- Basic SIP Instant Messaging (IM)
- Centralized message store while absent receiver.
- Delivery Receipt (DR) to sender at message delivery/discard.
- Receiver defined Auto Reply (AR) to sender.
- Receiver defined forwarding (FWD) of received messages to another receiver.

1.2 Final Product – what we ended up with

Beside implementing the required features mentioned above the following features where also implemented:

- Sender/Receiver defined Email Copy (EC) of sent/received messages
- Logging of sent/received messages in MAS Logging Data Base (LDB)
- Group conversation by Cloning (CL) received messages to one or more recipients
- Blacklists for blocking users the receiver do not want to receive any messages from
- IM/service preferences trough Subscribe Data Base (SDB) accessible for configuration through a web interface

Due to the lack of an already existing Erlang environment in Ericssons IMS network we also ended up writing and deploying an Erlang application server (AS). We focused on implementing only the features that we needed but it still ended up being a project as large as the instant messaging application. Since we did not get access to Ericssons Home Subscriber Server (HSS) we also implemented our own HSS together with the AS.

2 Preliminaries

2.1 Concepts

AS
: *Application Server*, an Erlang application server for IMS application development. It hosts a number of different applications and provides an abstract interface towards the hosted applications for the different services in the IMS platform.

MAS
: *Messaging Application Server*, hosted over the AS. It processes user "instant messages" and other requests like group-invitations, etc. There are two modes of operation – originating which processes sender's requests/messages, and terminating which process messages at the receiver's end.

2.2 Glossary

AR
: *Auto Reply*, according to the user configuration, the sender will receive an automatic reply if auto-reply is turned on.

Black list
: According to the user's configuration, a receiver will not receive the messages from the senders, whose user names are in the receiver's black list.

Check config
: The module that checks user's configuration while sending/receiving a message, exists in both Mas_Org and Mas_Term.

Clone
: A module in MAS and handles the message sent to a group. It copies the messages and sends them to the group members respectively.

Clone DB
: *Clone Database*, the database that stores the user groups.

Config DB
: *Config Database*, the database that logs all user configurations in the MAS. May also be called Subscriber database.

Diameter
: Diameter is a computer networking protocol for AAA (Authentication, Authorization and Accounting).

EC
: *E-mail Copy*, the module responsible for transcoding IM's to e-mails, if the user have such configuration enabled.

Group messages
: *Group Message*, a message that is sent to a group, which usually contains more than one person

Home Server
Hosts the AS and it's corresponding applications related to a particular user.

HSS
Home Subscriber Database, the component responsible for storing user data in IMS.

IM
Instant Message.

IMS
The *IP Multimedia Subsystem* (IMS) is an architectural framework for delivering IP multimedia services.

LDB
Log Database, logs all kinds of messages sent through the MAS.

MAS
Messaging Application Server

MAS_ORG
The originating side of MAS, processes sender's requests/messages.

MAS_TERM
The Terminating side of MAS, processes messages at the receiver's end.

MSDB
Message Store Database, the database in MAS that stores the offline messages.

Message Store
The module in MAS that stores the offline message into the MSDB and sends them later when the recipient comes online.

Offline Message
A message that is sent when the receiver is offline.

PGM
Presence and Group Management (PGM) is the component responsible for storing user accessible data in IMS.

Presence
The status of a user, e.g. online, offline.

S-CSCF
A SIP server in IMS responsible for session management. Short for Session Call Session Control Function. In this project used for checking register information (whether users are online or not).

SIMPLE
The *Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions* (SIMPLE) is protocol suite for instant messaging based on SIP.

Sh
Sh is an application interface for the Diameter protocol used to query and update user data in the HSS.

SIP
The *Session Initiation Protocol* (SIP) is a signaling protocol, widely used for setting up and tearing down multimedia communication sessions such as voice and video calls over the Internet.

SMTP
Simple Mail Transfer Protocol (SMTP) is an Internet standard for electronic mail (e-mail) transmission across Internet Protocol (IP) networks.

2.3 Development tools / languages

lighttpd
An HTTP server.

Coffee
Life giving substance made from hot water and grinded coffee beans. Revives people falling asleep at exactly 15:00 each day.

Coffee Machine
Useful for brewing coffee.

Danish Cookies
The best cookies ever made.

Dia
An open-source diagram/graphics tool.

Eclipse
An Java IDE.

Emacs
Escape Meta Alt Control Shift (Emacs), a text editor. There is a useful Erlang-mode for Emacs.

Erlang
A functional programming language with especially good support for concurrency. It was developed by Ericsson for use in telecom development, together with OTP it form the *Erlang / OTP* development platform.

J2ME
The *Java Platform, Micro Edition*, is a specification of a subset of the Java platform aimed at providing a certified collection of Java APIs for the development of software for tiny, small and resource-constrained devices based on micro-controllers.

JUDE
A UML modeling tool used to generate class diagrams.

Mediawiki
A widely used wiki software.

Mercurial
A distributed version control system.

OTP
: *Open Telecom Platform* is the standard library for the Erlang programming language, it is built with a focus on resilience and availability.

Microsoft Visio
: A diagram drawing tool.

OpenIMS
: Open Source IMS setup from Fraunhofer FOKUS NGNI. We used it for testing our AS and MAS.

Open Office
: An open-source office suite.

PHP
: *PHP* is a script language especially well suited for development of dynamic web pages.

Post-its
: We farmed a small forest for these.

SDS 4.1
: *Ericsson Service Development Studio*, is a service-creation tool that provides a unique environment for operators and independent software vendors to design and test their own IP Multimedia Subsystem (IMS) applications.

Sony Ericsson SDK
: A mobile phone emulator based on Java Platform.

UCT IMS Client
: Before we had our own client, we used this one.

Wiki
: A page or collection of Web pages that anyone who accesses it to contribute or modify content, using a simplified markup language.

Wireshark
: For tracking network traffic.

Whiteboards
: Very helpful in every part of the project.

3 IMS

IP Multimedia Subsystem (IMS) is an architectural framework for delivering IP multimedia services. It was standardized by 3GPP using existing IETF standard protocols (such as SIP) to be a next generation mobile network. IMS offers a number of advantages over previous mobile technology, (1) a common core layer and (2) new and exciting features such as:

- Find and connect
- Presence

- List and group management
- Message and data/file transfer
- Event subscription and notification
- Publish
- Automatic account provisioning

3.1 Architecture

The architecture of IMS, as can be seen in figure 1, is split into three layers. The application layer, the IMS layer and the transport layer. Our project is limited to the application layer or more specific the AS and the application part. However the AS do communicate with a few other entities, the *S-CSCF* and the *HSS*.

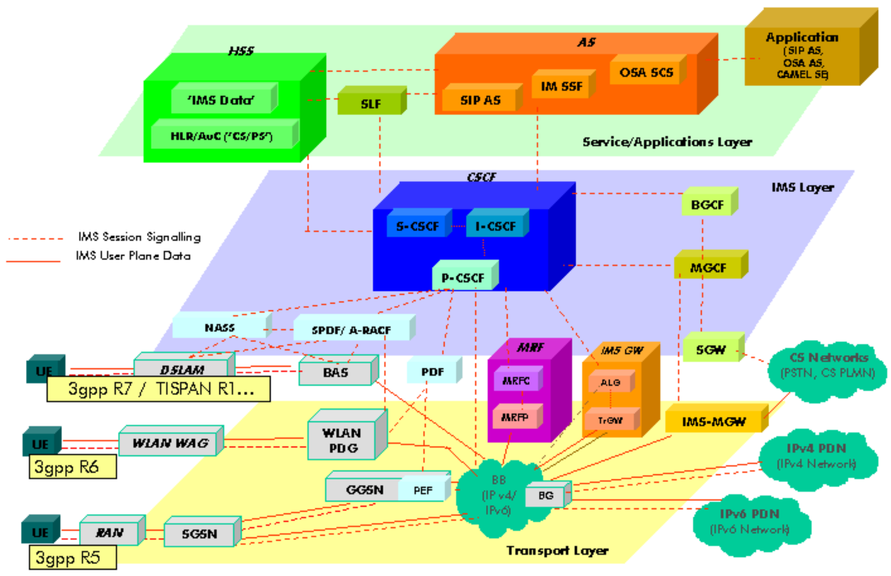


Figure 1: An overview of the IMS architecture

4 Application Server (AS)

The application server (or from here on the AS) was an attempt to create reusable components based on the more generic parts of the project. This means that while we had a fairly good idea of some of the components that would end up in the AS – such as the SIP stack – most of the requirements showed up as unplanned items. At the end of the course the following features were included in the AS.

- SIP/SIMPLE stack (mostly [2, RFC 3261] compliant)

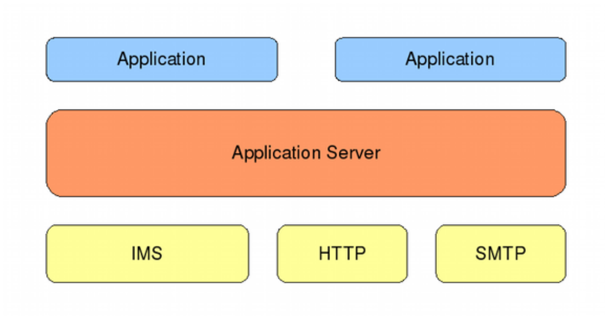


Figure 2: System architecture of the AS

  - Proxy, server and client behavior.
  - *MESSAGE* support.
  - Third party *REGISTER* support.

- Diameter / Sh stack
- Application message routing
- Application Programming Interface (API)
- OTP style code upgrade
- Web administration

4.1 System descriptions

The model used during the design of the AS can be seen in figure 2. It is split into three main layers, in a top-down order the first is the application layer, this is where the hosted applications live. Typically each application is only interested in a small subset of the services and traffic offered by the lower layers. Second is our middleware, the AS. It has the ability to interact with all of the different services available below and transcodes all of the different protocols into the language that is used between it and its hosted applications. In this way it provides an abstract interface for the hosted applications to communicate with the lower layers without having to reinvent the wheel in each of the applications. It also have a filtering function that prevents any application from receiving messages that it is not interested in, saving code complexity in each of the hosted applications. Last is the service layer or in this case the IMS environment. The IMS environment contain a great number of different services, part of these and the ones that the AS interacts with are the S-CSCF (Instant messaging + Third-party registration) and the HSS (User presence).

Rather than limiting our AS to only the IMS environment we also extended it to include a few other useful services that we judged to either be general enough for inclusion or just too useful to pass up. These kind of services include the SMTP stack and the web administration interface.

Above this it is also worth mentioning that the AS is written in Erlang and follows the OTP design principles. This means that in addition to being incredibly stable in the first place it also have a great number of fault tolerant features built-in.

The main services that the AS provide to its hosted applications are:

Application routing

As the AS can (and usually do) host more than one application at any point in time it have support for routing incoming message according to its application ID. This is one of the main features of the AS.

SIP / SIMPLE

The AS features a simple but mostly [2, RFC 3261] compatible SIP stack with support for both page mode instant messaging and third-party registration. This allows for the development of both peer-to-peer and multicast communication applications as well as user presence aware services. This is one of the main features of the AS.

Third-party registration

Using the SIP/SIMPLE stack the AS can take advantages of features offered by IMS such as third-party registration. This allow the AS to be aware of a users login and logout events effectively providing user presence awareness.

Diameter / Sh

Using the Diameter / Sh stack, we developed, the AS can and do provide its hosted applications with the connection status of a user on demand. (This is similar to the third-party registration mentioned above but not the same)

E-mail

An SMTP stack that provides support for sending e-mails.

Hot code-swapping

Due to properties inherit from the Erlang programming language and the OTP design principles the AS have support for hot code-swapping. This means that you can upgrade the AS to the latest version without ever stopping or having to restart the system.

Message routing

Whenever an application decide that it is no longer interested in a received message that was routed to it, it can forward it to the next interested node. This is one of the main features of the AS.

Web administration

For administration the AS uses a PHP based web front-end in true Web 2.0 spirit. This front-end have complete control over the whole AS and can not only change any part of it either through direct command-line access or through hot code-swapping but also have read-write access to all configuration files available and related to the AS.

4.2 Design / Architecture

The AS use a layered architecture with two main layers, the API which is responsible for communication with the hosted applications; and the service layer which communicate with the different services offered by IMS. The only component that is an exception to this rule is the web front-end which exists as a completely separate application running beside and interacting with the AS though the Erlang run-time system.

This architecture can be seen in figure 3 and have a number of advantages. The most important of these advantages are:

1. It provides a good level of abstraction.
2. It allowed each of the different protocols to be developed independently. (SIP, Diameter / Sh and SMTP)
3. If any one of the layers crash there is minimal impact on the surrounding environment (only the connecting two layers will be affected)

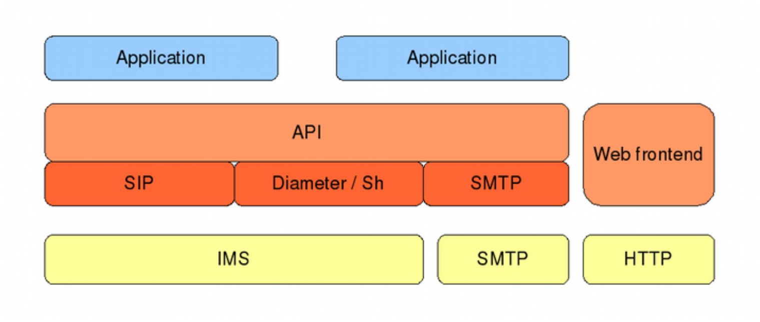


Figure 3: Design overview of the AS

4.2.1 API

The API is the top layer in our AS, it works as the bridge between the AS and the hosted applications. It is responsible for providing the necessary interfaces to all functions that the applications may need. We can have three different kinds of API:

1. API for SIP, this is used to provide instant message access for applications. Through this API, applications can create, get and set information when using instant messaging. Its also used for sending error- and different acknowledgment messages (such as 200 OK).
2. API for Diameter / Sh, used to give applications ability to get the registry status for users (online/offline status).
3. API for SMTP, gives the applications functions that enables sending of email messages.

4.2.2 SIP

IMS aims to help the multimedia access for applications. To this end, one of the IETF protocols that have been incorporated in IMS platform is the SIP protocol. SIP or Session Initiation Protocol as it name implies, is a protocol for creation, modification, and termination of multimedia sessions. SIP provides the means by which applications can agree on a set of multimedia capabilities such as text, voice, etc. Modification of session properties include multimedia properties such as the aforementioned agreements regarding text and voice, but it also includes dynamically adding or removing nodes(e.g. clients), and changing node addresses. The later feature is very useful to fulfill wireless and mobile applications requirements. In this regard we have implemented a mostly RFC compliant SIP protocol stack to both use some of these features and to be able to interact with other IMS applications.

Our SIP protocol stack is composed of three separate layers, where each layer provides a set of services to their upper layers, hence the term SIP protocol stack. Designing a protocol in layers allows ease of logical design and more importantly allows modification of each layer independently of the others as long as interfaces of layers are not changed improperly. Layers for the SIP protocol stack as implied by [2, RFC 3261] are from top to bottom the Transaction User, Transaction, and Transport layers. The general overview of their design and implementation considerations is described in the following sections.

Transport Layer Transport layer is the bridge between the upper layers and the TCP/UDP transport layers which makes the upper layers independent of the underlying transport protocols. Transport layer is responsible for delivering SIP messages over TCP/UDP and to do it efficiently and robustly. There are some features or functionalities incorporated in the transport layer based on [2, RFC 3261] which makes it more complicated than a basic client/server implementations. The functionalities of the transport layer comprise SIP persistent connections, P2P behavior, protocol switching, and address resolution.

- SIP Persistent Connection

  Connection persistence allows communication ends (e.g. a client and the application server) to use the same connection for sending and receiving requests and responses. This prevents the connection start-up overheads which are inevitable if persistence connections are not used. However, to make this transparent for the above layers there are some implementation considerations made, we will study this in the implementation section.

- P2P Behavior

  The AS transport layer provides a P2P kind of service which enables upper layers to implicitly act as both servers and clients for other servers when waiting for requests and sending responses. Or the other way around, act as both servers and clients for other server when sending requests and waiting for responses.

- Protocol Switching

  Transport layer is capable of switching between TCP and UDP when necessary. This might be necessary for instance when there is an error in making a connection to the other side over TCP and therefore it is desired to switch the transmission to UDP or the other way around. There might be application related requirements for this feature too.

- Address resolution

  There might be cases where a SIP message does not contain the real IP address of an originating side (e.g. the source address is in the nominal form or the originating side is behind a NAT).

Transaction Layer The SIP protocol is based on a transaction model very similar to the HTTP request/response approach. In other words, a transaction is started when a client forms and sends a SIP message requesting a function from a server and finishes when receiving a SIP response from the server. These requests and responses might also pass through intermediate proxies to get to the other end. The responsibility of the transaction layer in our implementation can be summarized in provision of client and server transaction management, and stateless proxy behavior.

- Client and Server Transaction Management

  Generally, transaction management hides away the details of dealing with creation, look up, error handling, and termination of transactions from the upper layers.

- Stateless Proxy behavior

  When a client sends a request it might be delivered via intermediary nodes which in turn will deliver the message to other intermediary nodes or the terminating server. Thus, intermediary nodes will just act as proxies which send forth and back the requests and responses respectively between the originating and terminating sides.

Transaction User This layer is the interface of the basic SIP stack functionalities to the higher level logic (which can be the SIP API or the raw user application messages). Modules in this layer are responsible for checking the validity of incoming SIP requests or responses from a previous hop of the communication. The responsibilities are divided between the UAC and the UAS.

- UAS(User Agent Server)

  The UAS is representative of a server application to the transaction layer. In other words when a request is received by the transaction layer it is forwarded to the UAS to handle it. There are a series of checks done according to [2, RFC 3261] on the content of the whole message in order to assure the validity of the SIP message. If any of the steps fail, an error message is sent back and the transaction is completed (i.e. terminated). If the request passes all the tests, then according to the method of the request, further processing will be done on it which can be seen in the implementation details.

- UAC(User Agent Client)

  UAC is representative of a client application to the transaction layer. In other words when a request is generated by an application it is forwarded to the UAC to handle it. There are a series of checks done on the content of the whole message in order to validate it.

4.2.3 Diameter / Sh

See the Diameter documentation section for an introduction to the Diameter / Sh stack in the AS.

4.2.4 SMTP

The SMTP Client is another component under the API layer in the AS. This component enables applications to send emails.

We are using a third part SMTP client module which was written by Michael Bradford for our project (The module can be downloaded at [7, trapexit]). It is a simple SMTP client using the gen_fsm behavior. It supports basic SMTP & ESMTP commands, including MD5, plain and login authentication.

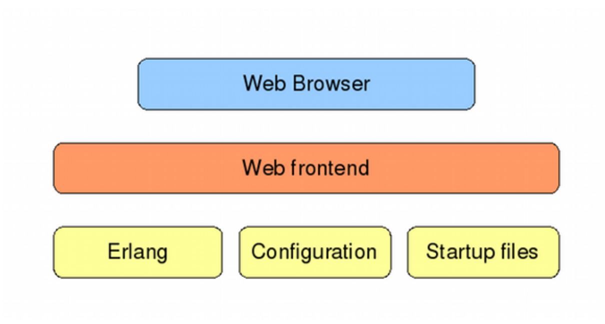


Figure 4: Design of the web front-end

4.2.5 Web front-end

The web front-end is as the name suggests an HTTP interface for the AS designed to provide administrative capabilities without any physical access to either the server or an SSH connection. The features offered by the web front-end are:

- Uploading of new AS versions (for OTP hot code-swapping)
- Erlang shell access.
- Editing configuration files.
- Editing Erlang start-up files.
- Restarting the physical machine (in the case of a panic situations)

In order to provide these functions the web fronted uses a standard MVC architecture, but instead of using a database as an exchange media between the Erlang run-time environment and the web front-end it interacts directly with the Erlang shell. The result can be seen in figure 4 and a short summary of each components role and design follows.

Web front-end
A group of PHP scripts.

Erlang
The Erlang run-time environment, the web front-end interacts with it directly instead of using some kind of in-between media such as a database.

Configuration
The configuration files that the AS reads from. These are located on the file system.

Start-up files
The Erlang environment start-up files
which defines things like location of databases, etc.

4.3 Implementation details

In the following sections we discuss implementation details regarding the different components of the AS. These details try to answer two questions, (1) how they are structures and how they work and (2) why this implementation was chosen.

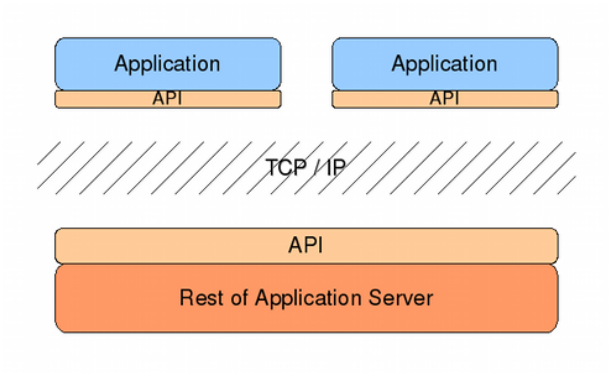


Figure 5: Implementation details of the AS API

4.3.1 API

The main tasks of the API is twofold, (1) to allow the user to access the functionality of the AS through a unified interface and (2) to allow the AS to inform its applications of various events (such as receiving of new instant messages). For this purpose a simple wrapper implementation would have been sufficient. However that would have imposed a large drawback, namely that both the AS and all of its hosted applications would have to run in the same Erlang environment and this would have a few implications.

1. You have to upgrade all of them at the same time.
2. They have read / write access to each other.

To prevent this we chose to make the applications live in a different Erlang environment than the AS, of course this means that you can no longer have simple wrapper functions as the API instead we had to add a Server-Client architecture with the AS hosting an API server and each application running a client as can be seen in figure 5.

The language spoken between the client and the server are in Erlang terms and form a simple request and response pattern. Typical messages look like:

```
{ ims, request, Tag, Request }
```

In this instance the message would have been sent from the AS to an application information them that they just received the SIP message `Request`. If the client decide that they want to respond to this message they would respond with:

```
{ ims, respond, Module, Tag, Response }
```

Where `Response` is the SIP response, `Module` the module that they respond through (typically an UAC) and `Tag` the same as the `Tag` element in the corresponding request. There are 6 more messages that can be sent and the full documentation for each of them can be seen in the `docs/api_flow.txt` file. However they all fall into one of these categories.

- Informing application / AS of events.
- Sending messages / responses.
- Register / Unregister applications.

lib_chan The library we use to implement the Server-Client architecture is lib_chan. It is a service layer running on top of TCP/IP that provide both authentication and streaming of Erlang terms in a transparent fashion. It was developed by Joe Armstrong and it available in the *Programming Erlang* book. For a full description of how it works read appendix D in aforementioned book.

There are however a few modifications made to this library for it to better fit into our architecture.

1. Make it run in a supervisor tree.

2. Only listen to loopback instead of any network interface.

3. Use our configuration interface instead of its own.

4. Use SASL error logging instead of its own.

4.3.2 SIP

In the following we will discuss general considerations in the implementation of our SIP protocol stack which was explained earlier.

Transport Layer

SIP Persistent Connections
To utilize persistent connections, one can use the simple method of spawning new threads for incoming TCP connection requests. However, there is a drawback to this method when we try to make persistent connections functionality transparent to the above layers. Consider the situation where the newly spawned child thread which is interacting with the above layer is terminated by client misbehavior in sending a premature TCP close message. In this case, the transport layer has no way of finding out if it was due to termination of the commitment of a transaction or a terminating due to misbehavior from the client. Therefore, the thread is terminated gracefully without knowing that the transaction is not finished yet. At this point, when the above layer wants to send back the response, it confronts a dead thread ID which it can not interact with. Hence, it must call on the main thread of the transport layer to spawn a new thread for a new connection with the client. This compromises the transparency of persistent connections service. To overcome this problem, we register the ID of each newly spawned thread in a general balanced tree (implemented in Erlang) and limit the interaction of the upper layer to the main thread of the transport layer. This way, whenever the above layer wants to send out a message on a supposedly-live connection, it sends it to the main thread which is keeping track of it's child threads liveness. In case a thread ID is not registered, it means that the thread was terminated. Therefore, the main thread makes a new connection with the other end, spawns a new thread, registers it and finally forwards the message to the new thread. The new thread will in turn deliver the message on the new connection to the other end and follows the further interaction with the above layer.

P2P Behavior
Transport layer provides the upper layers with a neat interface to send and receive messages to other nodes hiding away details about the underlying choice of protocols, connection and thread management, and other low level details,

thus providing a P2P behavior. It is also worth mentioning that in our implementation, if the TCP protocol is used as the underlying protocol, then all the requests and responses to the same node are sent over the same connection.

Protocol Switching
One problem occurs when one end wants to use a protocols for message transfer but the other end does not support that protocol. Thus the originating side will have to switch to the other transport protocol, this is all taken care of. Another problem is when a message needs to be transported over a congestion controlled transport protocol (which is TCP in our implementation) due to size constraint, the constraint in our case is that messages bigger than 1300 bytes needs to be transported over such protocol. This feature is also implemented. All these protocol changes result in changing the protocol type in the top Via header in a SIP messages [2, RFC 3261].

Address Resolution Transport layer is responsible for transmitting data to the other side by resolution of the IP used to send the message. The IP could be different from the address mentioned in the SIP message because it might be simply in its nominal form or the other end being behind a NAT. In order to achieve this, each incoming SIP message is inspected and if the address in the parsed top Via header field is different from the IP of the source, then a ”received” is added to the top via header field containing the IP of the source. This address will be the preferred address for further communication with the source.

Transaction Layer

Client and Server Transaction Management
We have divided these functionality into two different modules one for each of the client and server behaviors which are responsible of making request and response transactions respectively. Basically, both of this modules create a transaction and register it for each new incoming request from upper/lower layers and will look the transaction up again upon reception of response(s) based on a Branch field value generated and embedded in the transaction SIP messages. Transactions are removed by a function call provided to the layer that started them in the first place. It is worth mentioning that if a message is not acknowledged by the recipient after a time-out expires, then an error report is delivered to the upper layer.

Stateless Proxy Behavior
In order to keep track of incoming and outgoing messages and their correlations(if any) , stateless proxies use the notion of transactions as do stateful proxies I.e. originating and terminating sides. Each proxy adds a new header i.e. Via header, to SIP messages indicating address of the proxy and a unique parameter i.e. Branch field, which identifies the transaction. When the server sends back the response, it is routed on the same sequence of sender proxies but in the reverse order, allowing them to peel off the Via headers.

Transaction User

UAS
As we mentioned earlier in UAS behavior a series of checks are done on the

incoming request. One of those is to check for the type of incoming request. If a request is of type REGISTER, it is sent to all the applications in the application server. Then, the client is registered with an HSS(Home Subscriber Server) by the SH protocol specifying an expiry time-out or not according to the preferences specified in the request message. Finally, the appropriate SIP acknowledgment message is sent back to the client. However, if a message is not of REGISTER type, it will be directed to the proper application according to Accept-Contact2 header field in the request. In case a message is not matched with any residing application, it will be forwarded using the proxy behavior in the transaction layer. But If the request is sent to an application, the UAS waits for the response of the application(with a time-out) and sends back the response to the transaction layer when received.

UAC

A thing worth mentioning about the implementation of UAC is that it adds a new Via header and a newly generated Branch field to the outgoing request which will help the transaction layer identify the transaction. If any of the steps fail, an error message is sent back and the transactions is completed i.e. terminated. If the request passes all the tests, it is sent to the transaction layer to start a new client transaction. After that, UAC enters a time limited waiting state, expecting a response from transaction layer.

4.3.3 Diameter / Sh

The AS contains functionality for registering and querying presence via the HSS. The HSS is accessed through the Diameter/Sh protocol. For more information please see the Diameter documentation.

4.3.4 SMTP

As mentioned earlier we use an open source SMTP module. The module is good enough as is, but it still had some small defects when we evaluated it. To make it more suitable for our project, we made some modifications. That includes updating to make it support the latest version of Erlang, removing warnings that occur when its compiled, adding logging capabilities and fixing various bugs.

4.3.5 Web front-end

The web front-end is a set of operations, each written in the PHP programming language. Each of them are designed according to the template in figure 6. In the figure *Operation* is the feature being implemented, *Utilities* a common library containing shared code and *Configuration* all of the configuration variables. Further there is also a *Header* component not in the figure that each operation should include in the beginning, this component takes care of setting the correct CSS style and adding a menu at the top of each page.

The split between what should go into *Utilities* and what should go into the *Operation* part of the code have been decided according to a twist of the MVC model. Namely that the *Operation* part should only be the viewer and therefore contain a minimal amount of code, the bulk of each feature should be divided into a number of well defined functions and put in *Utilities*.

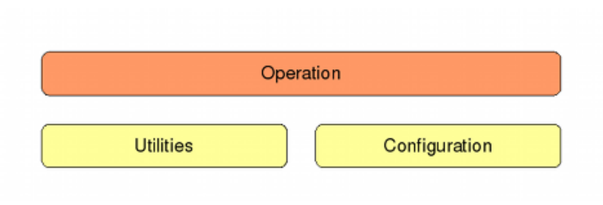


Figure 6: Implementation details of the web front-end.

Most of the operations performed by the web front-end are obvious, e.g. reading and writing of configuration and start-up files is simple file IO and restarting of the computer is executing a shell command. There are however two aspect of the web front-end that require further explanation, the interface between it and the Erlang run-time environment and its upgrading procedure.

Erlang run-time The web front-end interact with the Erlang run-time on a shell level, this means that its got access to the environment at the same level as if you had typed `erl` in the command line. However, the web front-end does not just spawn an Erlang shell, that would end up in a different run-time environment than the one the AS is running in. Actually it *connects* to the shell that the AS's run-time environment started.

This is done by reading and writing to a number of UNIX pipes. These pipes was created by the Erlang environment at start-up (for exactly this purpose), the pipes connect to the standard input and output of the shell for this specific shell instance. Typically these are located in `/tmp/erlang.[rw].1` but can be moved to a more suitable directory.

One of the modifications we did to the Erlang run-time environment was add possibility to set the permissions of these pipes, before our patch they were always set to `0500`.

Self upgrade Due to obvious reasons we needed to be able to upgrade the web front-end. Since the web front-end wasn't written in Erlang we couldn't depend on the OTP utilities for help on building a hot code-swapping feature. In order to be able to update the web front-end remotely, we had to invent our own system. The result consist of three main steps:

1. Create a backup copy of the original code.
2. Upload a tarball of the new web front-end.
3. Extract this new web front-end.

But is not as simple as it looks, since we ran into one complication – you couldn't overwrite a file that was being used by the HTTP server. This mean that the upgrade script itself and all of the related include- and configuration files which are used during upgrade process couldn't be upgraded. This was of course unacceptable, since that's more than half the code.

Our solution was simple, it meant inserting two extra steps:

1. Create a backup copy of the original code.

2. Upload a tarball of the new web front-end.
3. **Switch to running the backup code.**
4. Extract this new web front-end.
5. **Switch to running the new code.**

There is a race condition when someone else executes the code while we are doing the upgrade, but we decided that it's an acceptable risk.

4.4 Testing

One of the the goals with the AS project was to ensure a stable and correct application; and it is – in practice – impossible to accomplish this without at least some kind of testing outside of the intended environment hence following is a collection of the different kind of testing environments we used in the project.

Each section is dedicated to one of the types of testing that we have used, further each of these subsection discuss at least three different subject ”why”, ”how” and the results.

4.4.1 Distributed development model

During the project we used a distributed version control system, namely [6, Mercurial (Hg)]. This means that we did not use the (de facto) standard development model where everyone read and write to a central repository; but the model in figure 7 where each user have their own public repository that they work on. When they consider their work done or stable enough they tell a testing responsible person who will *copy* their changes to the main repository. Of course everyone can and do read from the main repository to ensure at least a decent level of sync between the individual repositories.

The model have several advantages over the standard model.

- It allow us to review changes that go into the main repository, and reject code that introduce regressions.
- Since each user have their own repository (that does not necessary have to be in sync with the main) you cannot *break each others code* daily due to small changes. This can only happen when you merge it with the main repository.
- Since each user have their own repository they can have as many local branches as they want.
- It is easier to keep local changes.

4.4.2 EUnit (Unit Testing)

Early in the development cycle it became obvious that in order for us to be able to meet our fairly strict quality goals on the code level of our project and to be able to take full use of the scrum development method (which encourage refactoring) we had to employ some kind of unit testing to our project. EUnit was our framework of choice for this, mainly because of the following points.

1. It was suggested by our lecturer during our early course in Erlang.

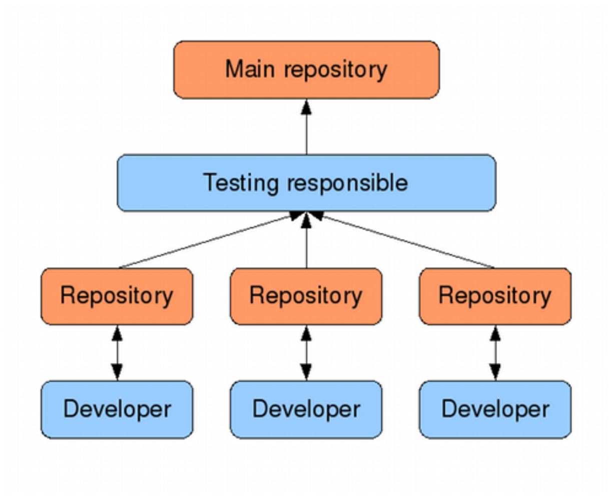


Figure 7: The distributed development model used.

2. It was included in the Erlang distribution (this was actually after we began using it).

3. It is easy to use.

Setup EUnit is included in any new version of Erlang (since version R12-B5) so to run it you should only have to run `make test` in the root of the source code directory.

Results The results of our unit testing was not as good as it could have been, mainly because three things:

1. Too many intra-module dependencies making it hard to test just one component.

2. Lack of good guidelines on how to write unit tests.

3. Simple lack of adaption during the very early stages of development (hence it was missing from some important modules and we just never got around to fix it).

4.4.3 OpenIMS Core + UCT IMS Client (System Testing)

Before we deployed our application at Ericsson we wanted to make sure that our AS worked, or at least didn't crash so hard we had to send a completely new disk to Ericsson (since that would take a few calendar days of work just to ship it there). So what we needed was a testing environment that we had complete control over down to the hardware – just in case we needed to push the reset button.

Platform Our solution was to install a number of virtual machines running the Open-IMS Core platform, an open source (GPL2) implementation of the IMS environment based on the *SER* SIP stack. And to interact with this environment we used the UCT IMS Client, an open source (GPL3) IMS client developed by the University of Cape Town, South Africa using the osip SIP stack.

Combining these two gave us a complete open source solution that we could test our AS (and MAS) in without any fear of crashing it or its environment. This had several advantages that we would make use of.

1. We could inspect the source code of any part of the project to figure out exactly what it was doing and why.
2. We could make changes to any part of the project to make it behave as we wanted it to.
3. Since they were running on virtual machines we could ”travel back” in time to an earlier working state (no need to clean up after dirty crashes)

The first advantage was of limited use because both OpenIMS and UCT IMS Client (especially the OpenIMS Core platform) are huge projects with many tens of thousands of lines of code. However we did make good use of the second point in the UCT IMS Client to make it work together with the AS (and MAS), to make it more suitable for testing and behave more like something in a mobile framework, changed include:

- Displaying of messages other than text/plain
- Adding of application ID to outgoing messages / responses.
- Stop the spam of status messages (this made the logs very hard to read)

Results The OpenIMS Core testing environment was our first real contact with an IMS environment hence it gave us a great deal of help both fixing bugs and (especially) in adding features that we noticed were missing or didn't think we would need earlier.

For just a taste of the bugs that was discovered (and fixed) due to the OpenIMS Core testing environment here is a short list.

- Lack of proxy behavior in the SIP/SIMPLE stack.
- How do discovered originating / terminating end of a session.
- Tons of minor bugs in various parts of the code.

5 Diameter / Sh

The Diameter is an Authentication, Authorization and Accounting (AAA) protocol which is used by HSS to interface between CSCFs and application servers. Sh is the application that is used to send the actual data built on Diameter protocol.

5.1 Requirements

Requirements for implementing Diameter protocol and Sh applications arose from the need of a secondary registration mechanism, when we weren't allowed to access the IMS registration facilities directly because of Ericsson's security policies. This mechanism allow us to implement presence based services in the MAS application, but it can be used by any applications on top of the AS.

Any agent who wants to use IMS platform must register itself using SIP register messages, which is basically a logon. These messages are handled and the data is stored by the HSS. Our MAS application also needs the registration data. Before implementing Diameter protocol and Sh applications we had the following possible scenarios to access the registration data:

1. Directly accessing the HSS using Sh client and Sh server applications.
2. Having our own HSS implementation and accessing it via Sh client and Sh server applications.
3. Implementing a not standardized and not specified database and be free to use any protocol or methodology to access and manipulate the data.

As stated earlier, we couldn't use the first option so we were left with second and third option. After analyzing the alternatives we found that the second option's protocol is well specified so implementation would be more straight forward. And when using an open source HSS implementation we did not need to implement an HSS. Another argument for the second option is that the data access protocol used in the first option, to access the HSS directly demands the use of Diameter and Sh applications. Diameter and Sh was needed for both first and second option and hence we decided to use the second approach.

5.2 Design / Architecture

Sh applications are implemented independent from any other module or application in the system. The specification of HSS says that communication between HSS and AS must be done using Diameter/Sh. Our implementation of Sh applications could be integrated in any module. We decided to have a design where the Sh server application integrated to the HSS and Sh client integrated to the AS. In the perfect case both ends have both client and server applications to support pull data, push data, update notifications and to subscribe to notification operations.

In our requirements, supporting notification system had a very low priority and our functional requirements are satisfied with only pull and push.

In any normal action, AS queries the HSS database using pull or push operation through the Sh Client. HSS is listening for queries using the Sh Server and handles both pull and push operations (retrieving and sending operations).

The diagram in figure 8 below shows the interaction between the HSS, the AS and the application (the MAS in the picture). It also shows where the Sh client and server are located and that the diameter protocol is used between them.

5.3 System descriptions

Diameter protocol is described in [1, RFC 3588] and Sh Applications are specified by 3GPP in the [3, 29.328] and [4, 29.329] documents. The HSS and the data stored is specified in part of [5, 3GPP 23.008] document.

Sh applications are running on diameter protocol and the heavy work is done by the protocol. Sh applications brings some additional data types to make the protocol support the HSS operations.

The protocol supports two transport layer protocols, SCTP and TCP. Both SCTP and TCP are implemented in our implementation. Handling the packages are done in the transaction layer and a finite state machine is used for this purpose. On the top of

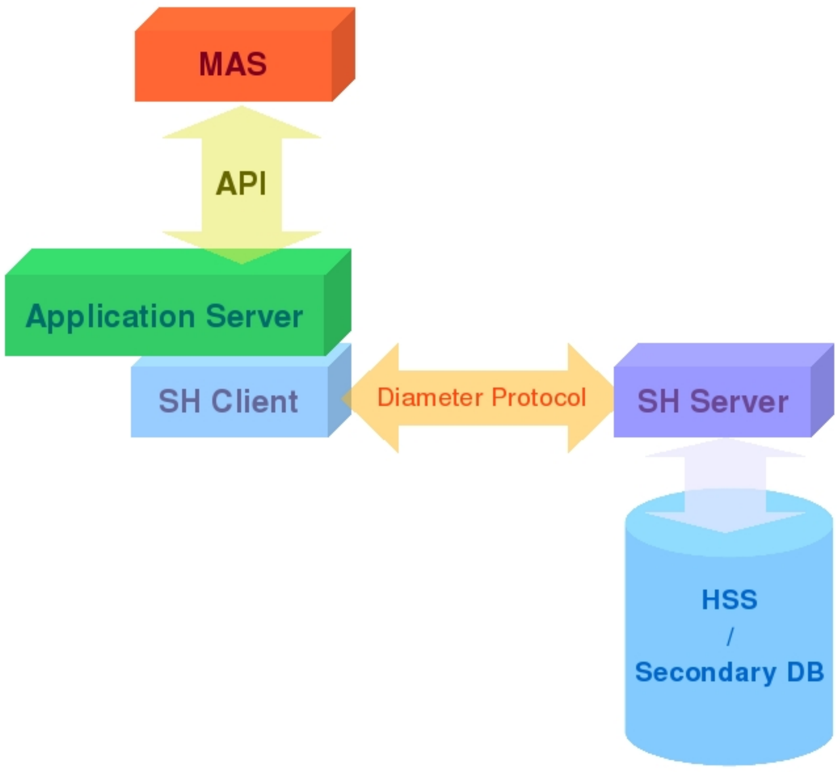


Figure 8: Architecture diagram

transaction layer SH applications are run and they communicate directly with the state machine below.

All the data are delivered using the Attribute Value Pairs (AVPs) defined in the Diameter protocol and the extra AVPs that required to support Sh applications.

5.4 Functionalities / Features

We only need a partial implementation of the Diameter protocol to meet our requirements, but it has enough infrastructure to support full functionality for a future development. The Sh application is also partially implemented for the same reason. In our version pull and push operations are supported by the system. So our MAS application is able to process a register message and register the user to the secondary registration database, which is functionally an HSS, and make queries and modifications on the data stored.

5.5 Implementation details

Diameter protocol and Sh applications are implemented on the Erlang/OTP platform. According to our system needs, and nature of the architecture, an implementation using all of the specifications isn't needed. The biggest difference between specification and implementation is that it's recommended to separate connection logic from session logic in the specification. We implemented sessions dependent on connection between pairs. This is because we don't need to place any redirect or proxy servers in the network.

Another issue is the logic of SCTP based sessions and TCP based sessions. Parallel processing of two session between the same pair doesn't need to be supported according to the specifications. We have support for parallel sessions in TCP connections but parallel sessions for SCTP connections aren't supported for now. Since this feature isn't required and would have taken a lot of time compared to its low priority, it was skipped.

Any session initialization is started with a trigger call to the corresponding module and starts a chain reaction in the architecture. When a connection is established, transport layer module initializes session data, stores needed information and triggers upper layer module packet handler. Packet handler does the same and triggers state machine in the upper layer. Diameter protocol logic runs in the state machine and triggers corresponding Sh application. During the session, data and information flow goes through the same way, see figure 9. If the session is initialized from the Sh client, the chain of flow starts from top and goes down.

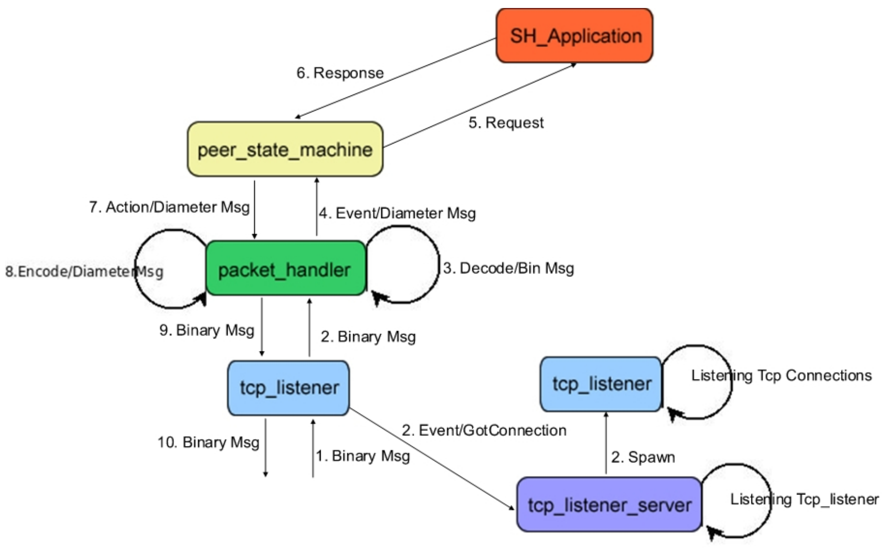


Figure 9: Message flow between layers

Any new application like CX or DX may be added in the future in the same way Sh is added today. The Diameter implementation is done independent from the Sh application implementation and it supports other applications as well.

6 Messaging Application Server (MAS)

The MAS is a messaging service available to the end-users of the Ericsson IMS framework, which allows them to deliver instant messages to other users of the network while each user, both at the sending and the receiving end, are able to maintain their own preferences for the application. Making the application suit the users need.

The Messaging Application Server or simply the MAS is an application, hosted over the Erlang application server provided by the AS team, which in turn is hosted over the IMS platform (the AS is described in detail in section 4). MAS processes messages sent between end users such as text message, delivery-receipt, auto-reply, group-invitations etc. The name of this application might be slightly misleading (it should have been something like IMA – Instant Messaging Application, but we couldn't agree on a name during the project). The MAS is not really an Application Server in itself but simply one of the applications hosted over the AS; however, we still keep the name to avoid any confusion regarding the CS project.

The MAS offers the following services to the end-users:

- Simple text messaging
- Offline messaging
- Delivery receipt
- Auto-reply
- Email copy
- Message forwarding
- Group messaging
- Blacklists
- Message logging
- Application level registration

We will describe each of them in detail in the following sections.

6.1 Requirements

The general requirements for the MAS application, as provided by the customer (Mobile Arts), is explained under the following points:

Simple text messaging
A user registered with the IMS network should be able to send instant text messages to other users who are registered with the IMS network.

Offline messaging
If a user is unavailable or offline (busy/out-for-lunch or not registered with the IMS network), the messages sent to the user should be stored in some intermediate storage and delivered to him/her as soon as he/she becomes available (comes online).

Email copy
Users sending and/or receiving instant text messages should be able to send them as email messages to specified email addresses.

Delivery receipt
Users should be able to receive delivery success notification in case of successful message delivery and delivery failure notification in case message delivery failure occurs due to any technical reason. Users should also be able to enable and disable such a service.

Auto-reply
Users using the services of the MAS should be able to specify auto-reply messages, independent of their status being online or offline. Auto-reply messages are sent back as an automated response from the receivers to those sending messages. Users should also be able to enable and disable such a service.

Message forwarding
Users should be able to enable receiver defined forwarding of received messages to another receiver, i.e. to forward all its received messages from one of his/hers addresses to another. This function must also come with loop detection and cancellation when messages are forwarded more than once.

Group messaging
Users should be able to enter into group conversations on invitation from an existing group member. Each group member should be able to join, leave and send/receive messages visible to the entire group i.e. all members of the group.

Blacklists
Users should be able to avoid receiving messages from other users by specifying them. The users sending messages will not be aware of their messages being ignored by the receiver. Auto replies will be skipped in this case.

Message logging
MAS should log all messages being received and sent by all the users. This is used for administrative purposes. However, it may be made available to users in a read-only mode through a web interface.

Application level registration management
Besides getting presence information from the IMS network, the MAS application should be able to maintain private application level registration information in order to leverage the MAS of the IMS presence information dependency. This is particularly useful in case of IMS “presence status” conflicts with other applications the user might be using.

User settings management (subscriber/configuration database)
Users should be able to save and change their settings and preferences in a subscriber’s repository to avoid specifying preferences every time they start using the application.

6.2 Design / Architecture

Like any typical telecom application, the MAS is a distributed application where multiple instances can be hosted over the IMS network keeping in mind load distribution and distributed service availability for the users of the entire IMS network.

The services of the MAS are made accessible by hosting it as an application over the AS, which in turn is deployed over the IMS platform. The AS subscribes to the S-CSCF to request forwarding of specific SIP messages, which are received from specific registered or recognized users. All the SIP messages received by the AS from the IMS network are forwarded to the originating or the terminating side of the MAS, according to the information provided in the relevant header of the SIP message. This however, is determined by the AS (by looking at the ACCEPT-CONTACT2 field of the SIP message) and is not needed to be taken care of by the MAS.

The MAS has two modes of operation, the originating and the terminating. The originating MAS processes requests on the sending users end, whereas the terminating MAS processes messages/requests at the receivers end. Messages/requests processed by the originating part of an MAS will be processed by the terminating part of the next MAS (which may be the same as where the originating is; this is very much possible in distributed applications), before the message or request is completed or delivered to the receiving user.

6.2.1 Originating MAS

The originating MAS is a part of the MAS application, which handles messages soon after it arrives at the home server of the sender. The message is first processed by the originating MAS where logging, message type checking and other relevant actions are taken depending on the type of the message request. If required the message is then forwarded to the AS, which sends it onwards to the MAS at the receiver's home server; there the message is processed by the terminating MAS.

The architectural flow of the Originating MAS as shown in the diagram 10 can be described in the following steps:

1. A SIP message received by the AS is forwarded to the "Read SIP" message module, which unpacks and parses the sip message.

2. The message and parsed information is logged (2B), and relevant information from the SIP message is passed to the Type Checker module (2A), concurrently. Message Type Checker detects the type of request and takes the following actions accordingly.

    1. If the request is of type group-message, the message information is sent to the Clone module which sends relevant messages to all the group members. Group members information is retrieved from the Clone DB.

    2. If the request is that for a simple instant message, the message information is forwarded to the "Check Config" module, which may add relevant information to the message depending upon the sender's settings as present in the Configuration DB. The message information which may now be modified is forwarded to the next modules. The following two operations may be performed in parallel.

        1. Message information is forwarded to the "Send SIP message module", which composes a new SIP request and sends it back to the AS, which then forwards this request to the AS-MAS bundle at the receiver's home server.

        2. If the Email copy has been enabled, the "Check Config" module invokes the Email-Copy module which then embeds the message body

in an email message and sends it to the email address specified in the Configuration DB.

3. If the request is a 200 OK, the message information is sent to the Check Response Message module, which takes relevant action on the request (if needed). In the current implementation, this module is never used since the AS handles SIP status messages internally and doesn't forward such requests to the MAS at all.
4. If the type of the request is a REGISTER, then the "Check Offline Messages" module retrieves all the offline messages addressed to the sender, embeds all the messages in new SIP messages addressed to the sender and forwards them to the AS.

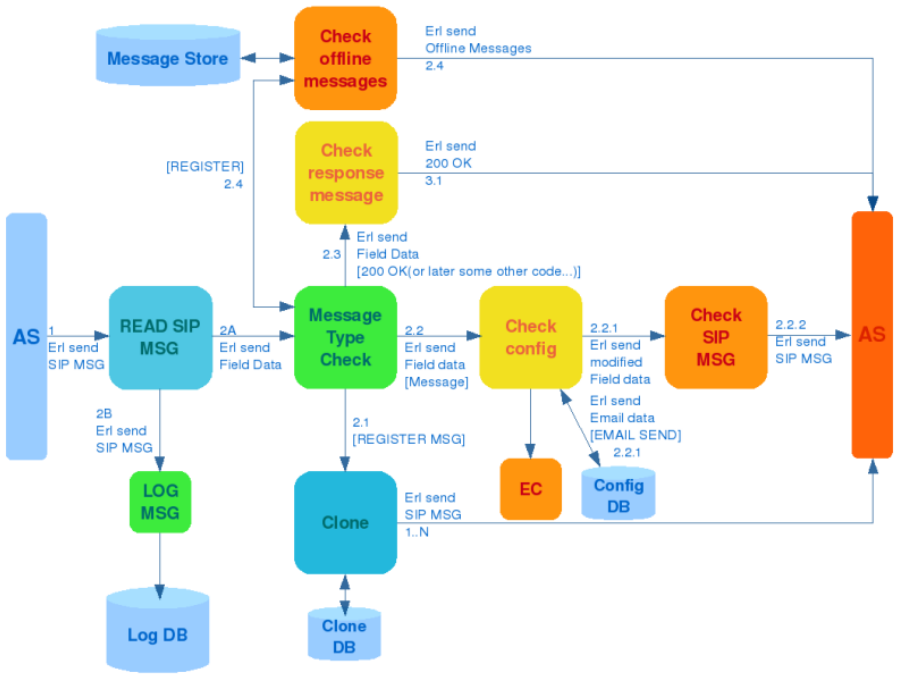


Figure 10: MAS originating side message flow

6.2.2 Terminating MAS

The Terminating MAS is the part of the MAS application which handles messages received on the receiver's home server, after it has been processed by the Originating MAS. Terminating MAS logs the incoming messages just like the Originating MAS. Besides logging, Terminating MAS takes necessary actions for cloning requests, offline messages, generating Auto-reply and Delivery-receipts. If required, the message is forwarded to the AS which in a final step delivers the message to the receiver.

The architectural flow of the Terminating MAS as shown in the diagram 11 can be described in the following steps.

1. Sip message received by the AS is forwarded to the "Read SIP message" module, which unpacks and parses the SIP messages.

2. The message and parsed information is logged (2B), and relevant information from the SIP message is passed to the Type Checker module (2A), concurrently. Message Type Checker detects the type of request and takes the following actions accordingly.

    1. If the type of the request is “instant message”, the message information is forwarded to the “Check Config” module, which may add relevant information to the message depending upon the receiver’s settings in the Configuration DB. The message information which may now be modified is forwarded to the next modules.

        1. If the Email copy has been enabled in the receiver’s settings, the “Check Config” module invokes the “Email-Copy” module which then embeds the message body in an email message and sends it to the email address specified in the Configuration DB (3.1A)
        2. If the receiver is found to be offline through the presence module, the “Check Config” module invokes the “Auto-Reply” module which may send an auto-reply to the sender if the receiver has enabled this service (3.1.1).

            1. If enabled, the “Auto-Reply” module sends the original message to the message store and forwards the auto-reply message to the AS, concurrently.

        3. If the receiver is online, as per information form the Presence module, “Check Config” module forwards the message to the “Deliver or Forward” module, which then forwards the message to the AS to be delivered to the receiver (3.1.2).

    2. If the request is a 200 OK (or with any other valid code), the message information is sent to the “Check Response Message” module, which takes relevant action on the request (if needed) and forwards the message to the AS. In the current implementation, this module is never used since the AS handles SIP status messages internally and doesn’t forward such requests to the MAS at all.

    3. If the request is an “offline message”, the message is forwarded to the “Check Config” module. From there the message is forwarded to the “Deliver or Forward” module, which in turn sends the offline message to the AS to be delivered to the receiver.

    4. If the request is a group message, it is forwarded to the “Clone” module which takes the following steps accordingly;

        1. In case of “Group-Message”, the message is simply forwarded to the “Deliver or Forward” module, provided the receiver is online. This information is fetched from the Presence module.
        2. If the receiver is offline, a “remove me from the group” group message is composed and sent to all the members of the group, which the receiver had been a part of. The group members information is retrieved from the Clone DB.

Presence Module is an information retrieval module, which informs about the online status of a user when invoked by any other module or process. Presence module gets the presence status information from the HSS, an essential part of the IMS platform.

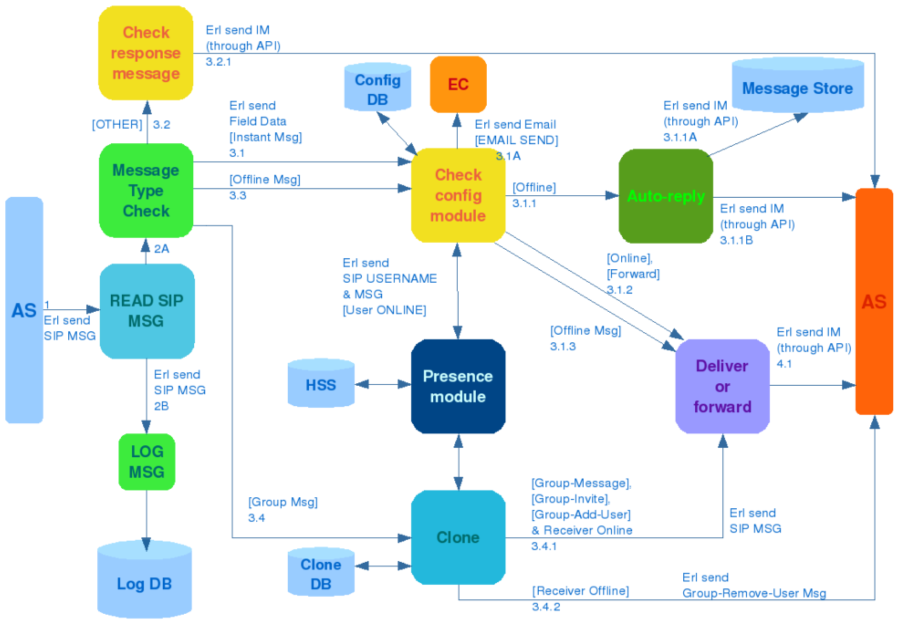

Check response message
Erl send IM (through API) 3.2.1
Config DB
EC
Message Store
[OTHER] 3.2
Erl send Field Data [Instant Msg] 3.1
Erl send Email [EMAIL SEND] 3.1A
Erl send IM (through API) 3.1.1A
Message Type Check
[Offline Msg] 3.3
Check config module
[Offline] 3.1.1
Auto-reply
Erl send IM (through API) 3.1.1B
AS
2A
1 Erl send SIP MSG
READ SIP MSG
Erl send SIP USERNAME & MSG [User ONLINE]
[Online], [Forward] 3.1.2
AS
[Offline Msg] 3.1.3
Erl send IM (through API) 4.1
Erl send SIP MSG 2B
HSS
Presence module
Deliver or forward
LOG MSG
[Group Msg] 3.4
[Group-Message], [Group-Invite], [Group-Add-User] & Receiver Online 3.4.1
Erl send SIP MSG
Clone
Log DB
Clone DB
[Receiver Offline] 3.4.2
Erl send Group-Remove-User Msg

Figure 11: MAS terminating side message flow

6.3 System descriptions

The MAS is simply a message processing application which performs certain user-defined actions on incoming messages. The MAS is an application registered and hosted by the AS through a well-defined TCP interaction layer. The AS is in turn hosted over the IMS platform. The AS receives user messages as SIP messages from the IMS, and forwards them to the MAS which has subscribed with the AS to forward different types of requests to its originating and terminating sides. Messages that have been previously processed by an Originating MAS are sent to the Terminating MAS by the AS. The MAS works with the incoming requests independent of the IMS platform details with the help of the abstraction provided by the AS.

The MAS message processing mechanism is specific to SIP 2.0 standard and is not capable of processing any other messaging protocol. This is because, the MAS was an application intended for use over the IMS platform, which itself extensively uses SIP 2.0 for its own internal messaging.

Besides the normal message headers, the MAS uses following custom headers;

- Group-ID
- Group-Members

MAS also makes use of the following self-defined custom SIP Content-Types;

- application/auto-reply
- application/delivery-receipt
- application/group-invite
- application/group-add-user

- application/group-remove-user
- application/group-message

6.4 Functionalities / Features

The MAS can fulfill the following functionalities

Simple text messaging
A user registered with the IMS network or the MAS application through the MAS registration management can send instant text messages to other users who are registered.

Offline messaging
If a user is not available or is not online, then the messages sent to him are delivered to him as soon as he comes online.

Email copy
Users sending and/or receiving instant text messages can also copy the instant messages as e-mails to email addresses specified by them.

Delivery receipt
Users having enabled the Delivery-Receipt option will receive delivery-receipts for each simple instant message they send. Please note that this service is not applied to group-messages, auto-replies, forwarded messages and delivery-receipts.

Auto-reply
Users having enabled the Auto-Reply option will cause the MAS to send automatically generated replies on that user's behalf to anyone sending messages to it. Please note that this service is not applied to group-messages, auto-replies, forwarded messages and delivery-receipts.

Message forwarding
Users can forward messages to other users by enabling this option and specifying the other user's SIP address. Forwarding is not applied to group-messages, auto-replies and delivery-receipts.

Group messaging
Users can initiate a group conversation and become a part of it on invitation from an existing group member. Each group member can join, leave and send/receive messages to the entire group. Groups are automatically destroyed when there are no more users in them.

Blacklists
Users can avoid receiving messages from other users by enabling this option and specifying target users. The users sending messages will not be aware of their messages being ignored by the receiver. Auto-replies, forwarding and delivery-receipts will not be applied in this case.

Message logging
MAS logs all messages being received and sent by all the users. This is done by separate logging logic's on both the terminating and originating sides.

Application level registration management

Besides getting presence information from the IMS network, the MAS application is able to maintain private application level registration information in order to leverage the MAS of the IMS presence information dependency. This is done by emulating the HSS and making it a part of the application itself.

6.5 Implementation details

The MAS application is implemented entirely in Erlang using the OTP standards. For the data layer requirements and repositories, Erlang environment's inherently available database MNESIA has been used. Separate databases are used for Logging, Clone/Group information, subscriber configuration and temporary offline message storage.

The internal logic of the MAS handles each request, that is received from the application server, by spawning a new process for each module that is going to process that message. Each module runs as a separate process and is responsible for processing just one request at that stage. We call this logic "Dynamically linked pipelined message processing", since a message at any given stage/module is independent of the previous stage worker thread which can work on any other request, if needed. An intermediate data buffer is used to pass a message between two modules. A load balancer keeps track of the number of working processes and waiting requests at an intermediate data buffer between every two modules. The load balancer kills or generates processes depending upon the number of available requests waiting in the buffer queue. A minimum pool of all processes is always kept alive, to handle sudden burst of requests efficiently. This can be changed with the help of the provided configuration files. This minimum pool is an efficiency improvement from the design point of view and may not be visibly efficient in the actual running environment, since in the Erlang environment killing and creating processes is extremely cheap. This will however affect efficiency in any other environment like the JRE (Java run-time environment), where frequent creation and killing of processes can be very expensive.

The MAS uses Erlang messaging for application's internal message passing and communication. For interaction with the application server, the MAS uses a well-defined protocol of registering, message passing and de-registering mechanisms through a TCP layer. All messages exchanged between the MAS and the AS are communicated through the API served by AS. Communication port for each application hosted by the AS is defined in AS configuration files and can be changed if required.

6.5.1 User Configuration

To be able to use the application, each user needs a user configuration stored in the configuration database. They only need to make one for each MAS they intend to use.

The database has multiple tables, with one super-table and several sub-tables detailing the behavior of the application.

user_config

The super table. Contains alias and password, the password is intended to be used with user configuration edit page.

delivery_receipt

This is a deprecated table that still remains. Its not used.

email_copy
Details the behavior for the email copy module.

presence
This is a deprecated table that still remains. Its not used.

forwarding
Details forwarding behavior.

auto_reply
Details auto reply behavior.

black_list
Details the blocking list.

When a user is deleted, all the corresponding rows in other tables should also be deleted. E.g. when user A is deleted in table “user_config”, user A’s information in “email_copy”, “delivery_receipt”, etc is also removed.

Looking at email copy we get a good idea on how it works.

email_copy
- Key: user_name
- org_bool: default value as **false**. Origin side, set as **true** when a user is online and uses email_copy
- term_online: default value as **false**. Terminating side: set as **true** when a user is online and uses email_copy
- term_offline: default value as **false**. Terminating side: set as **true** when a user is offline and uses email_copy
- email_copy_data: application data

6.5.2 Web front-end

To be able to maintain the application, do upgrades and error checks we used a copy of the AS web front-end see section 4.2.5. The only added functionality is to edit user configurations. This is also possible through the mas_config_db module’s exported get/set-functions.

6.5.3 Testing

During the MAS development lifecycle, development and testing has gone hand in hand. We thoroughly tested every functionality, once we had completed it. However, the development and testing environment has not been very straightforward. Development and testing in the early phases began over a simulated IMS environment called OpenIMS, since Ericsson hadn’t granted our application server access to the Ericsson IMS network. Once we gained access to the Ericsson IMS network, we had to retest and adjust our implementation to that environment. Our entire testing has been based on the actual running environment of the Ericsson IMS platform.

The first test runs on the OpenIMS was helpful because we could verify that the application started and stopped correctly. When this was done we could do the release upgrades and test the uploads in a controlled environment. We didn’t finalize the application in this stage, but we did get something that we knew we could upgrade and restart if some error surfaced. This way we were confident that we could solve most of the problems that could occur when the application was deployed, without having to involve Ericssons staff.

We used a distributed development model in a similar way as we did with the AS section 4.4.1. Each user maintained their own version with their local changes and we had a group which where responsible for adding all the changes into one main repository. This worked fairly well, the problem was once again lack of experience. We did faulty pushes to the repositories.

6.6 Mobile Client

6.6.1 Requirements & System descriptions

The whole purpose of having a mobile client in this project is to test the functionalities of our Application Server and Message Application Server which are above the Ericsson's IMS platform and Open IMS platform.

The focus of this mobile client is not about building a perfect mobile program but testing our Application Server, Message Application Server and the communication between IMS platform.

Our mobile client is implemented in a rather short time comparing to the AS and the MAS. We would say that it might not be a perfect mobile solution but we have got what we wanted.

Our development environment is Ericsson Service Development Studio – SDS 4.1 together with Sony Ericsson SDK and J2ME. We randomly choose type Sony Ericsson_JP8_240∗320 as our emulator.

6.6.2 Design

A class diagram of the design can be seen in figure 12 Here is the class diagram of our mobile client:

GUI Layer The GUI layer is responsible for displaying the pages (frames). It does not do much work, its only tasks is to display the information we received from the server as well as remembering the information that the user have inputted and pass the actions to the lower technical layer.

Technical Layer The technical layer is the event handler. There are two classes:

Commands
: Implements the interface *ActionListener* and is responsible for handling events from the GUI layer.

MyCoreServiceListener
: Implements the interface *CoreServiceListener* and is responsible for listening for SIP messages received from the server, and sending them to the GUI layer.

The API we used in this layer is [8, IMS Innovation].

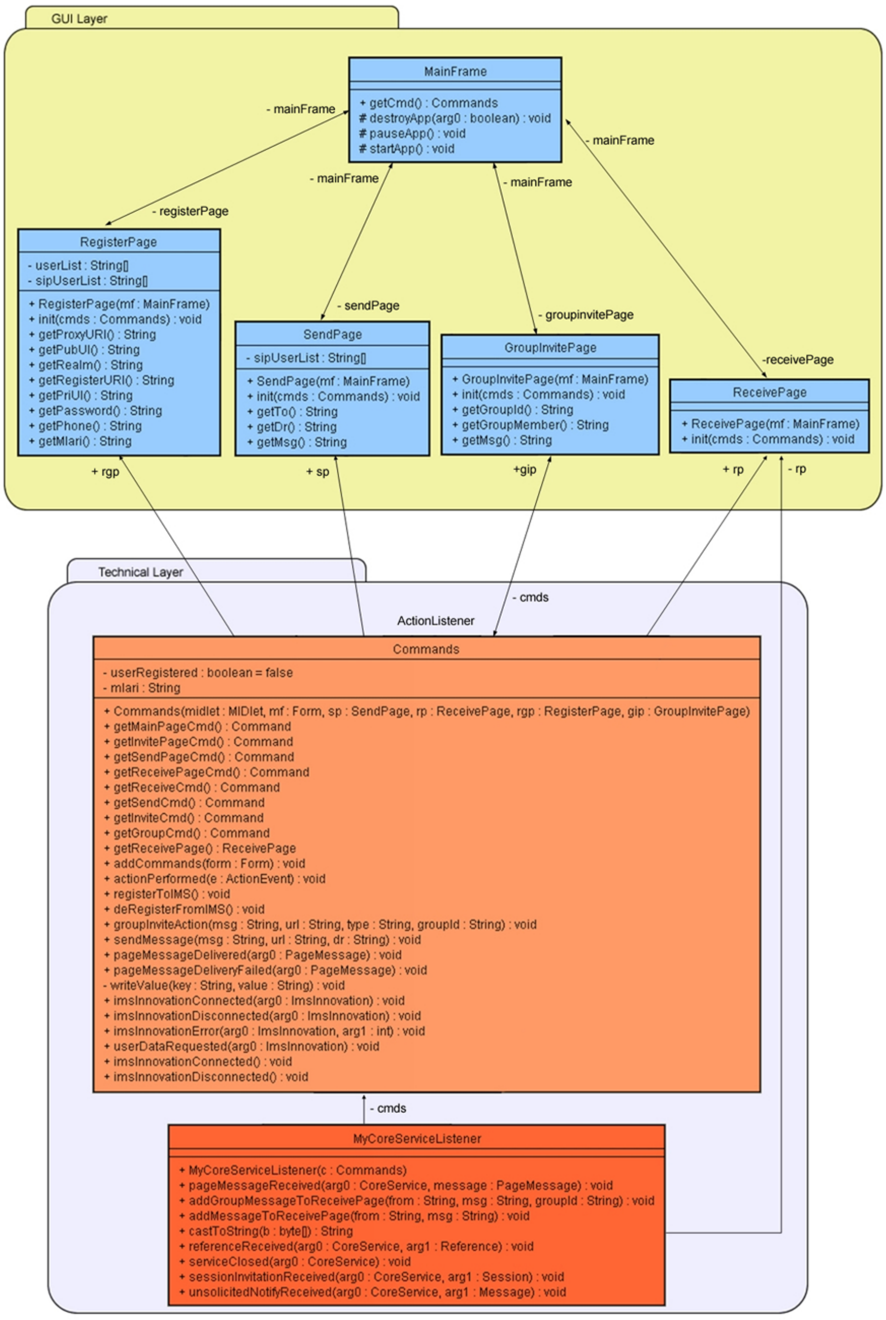

GUI Layer
MainFrame
+ getCmd() : Commands
# destroyApp(arg0 : boolean) : void
# pauseApp() : void
# startApp() : void
- mainFrame
- mainFrame
- mainFrame
- mainFrame
- registerPage
- sendPage
- groupinvitePage
-receivePage
RegisterPage
- userList : String[]
- sipUserList : String[]
+ RegisterPage(mf : MainFrame)
+ init(cmds : Commands) : void
+ getProxyURI() : String
+ getPubUI() : String
+ getRealm() : String
+ getRegisterURI() : String
+ getPriUI() : String
+ getPassword() : String
+ getPhone() : String
+ getMlari() : String
SendPage
- sipUserList : String[]
+ SendPage(mf : MainFrame)
+ init(cmds : Commands) : void
+ getTo() : String
+ getDr() : String
+ getMsg() : String
GroupInvitePage
+ GroupInvitePage(mf : MainFrame)
+ init(cmds : Commands) : void
+ getGroupId() : String
+ getGroupMember() : String
+ getMsg() : String
ReceivePage
+ ReceivePage(mf : MainFrame)
+ init(cmds : Commands) : void
+ rgp
+ sp
+gip
+ rp
- rp
Technical Layer
- cmds
ActionListener
Commands
- userRegistered : boolean = false
- mlari : String
+ Commands(midlet : MIDlet, mf : Form, sp : SendPage, rp : ReceivePage, rgp : RegisterPage, gip : GroupInvitePage)
+ getMainPageCmd() : Command
+ getInvitePageCmd() : Command
+ getSendPageCmd() : Command
+ getReceivePageCmd() : Command
+ getReceiveCmd() : Command
+ getSendCmd() : Command
+ getInviteCmd() : Command
+ getGroupCmd() : Command
+ getReceivePage() : ReceivePage
+ addCommands(form : Form) : void
+ actionPerformed(e : ActionEvent) : void
+ registerToIMS() : void
+ deRegisterFromIMS() : void
+ groupInviteAction(msg : String, url : String, type : String, groupId : String) : void
+ sendMessage(msg : String, url : String, dr : String) : void
+ pageMessageDelivered(arg0 : PageMessage) : void
+ pageMessageDeliveryFailed(arg0 : PageMessage) : void
- writeValue(key : String, value : String) : void
+ imsInnovationConnected(arg0 : ImsInnovation) : void
+ imsInnovationDisconnected(arg0 : ImsInnovation) : void
+ imsInnovationError(arg0 : ImsInnovation, arg1 : int) : void
+ userDataRequested(arg0 : ImsInnovation) : void
+ imsInnovationConnected() : void
+ imsInnovationDisconnected() : void
- cmds
MyCoreServiceListener
+ MyCoreServiceListener(c : Commands)
+ pageMessageReceived(arg0 : CoreService, message : PageMessage) : void
+ addGroupMessageToReceivePage(from : String, msg : String, groupId : String) : void
+ addMessageToReceivePage(from : String, msg : String) : void
+ castToString(b : byte[]) : String
+ referenceReceived(arg0 : CoreService, arg1 : Reference) : void
+ serviceClosed(arg0 : CoreService) : void
+ sessionInvitationReceived(arg0 : CoreService, arg1 : Session) : void
+ unsolicitedNotifyReceived(arg0 : CoreService, arg1 : Message) : void

Figure 12: Design of the mobile client.

6.6.3 Functionalities / Features

Register Page The register page is used to register to IMS platform before send, group invite and receive messages from mobile client. We are supposed to send the following items to the IMS:

- ProxyURI (sip:193.180.168.44:35060),
- PubUI (sip:Xiaoming.Yu1@imsinnovation.com),
- Realm (imsinnovation.com),
- RegisterURI (sip:imsinnovation.com),
- PriUI (Xiaoming.Yu1@imsinnovation.com),
- Password (UUErlang)
- Phone Number

Send Page The send page is used to send IM messages with a single recipient.

- Destination address (sip:Xiaoming.Yu2@imsinnovation.com),
- DR-enable (True/False), whether delivery receipt is enabled.
- Message body.

Group Invite Page The group invite page is used to fulfill the group invite functionalities which can be divided into two parts, one part is the sponsor of the group invite, it should contain:

- Friend address which is the SIP address of the friend that you want to invite.
- Group-Id, an unique identifier for the group.
- Message body.

The other part is the group send functionality. It contains the same information as above and is used for sending group messages to an already existing group. Of course clients are able to leave the group at any time.

Receive Page The Receive Page is used to display received messages.

6.6.4 Implementation details

Libraries Used

LWUIT.jar
Graphical framework for J2ME.

IMS_1.1.1
Framework provided by [8, IMS Innovation].

JUDE-1.2.1
Class diagram generation framework.

Problems In the beginning our Application Server and Message Application Server was hosted on the OpenIMS platform and we had problems with connecting our mobile client to the AS. We were stuck here for a while, but later we figured out that there is nothing we can do except shifting our AS and MAS to the Ericssons IMS platform which "solved" the problem.

We also encountered problems with our own custom headers in the SIP messages like `Group-ID` and `DR-enabled`. However, the [8, IMS Innovation] API does not allow us to do that. We post our problem on the portal and got a reply with a fixed API which allow us to edit our own custom headers.

Future Work It would be great if we can deploy our program on a real mobile in the future, moreover, fixing bugs within the program. Making some fancy GUI would also be nice.

Testing We have not done a lot on testing our mobile client and we did not have a formal testing procedure since we started with it rather late and it is not the main goal of our project. This is the part we could do better. Our emulator is Sony Ericsson_JP8_240∗320. We could test our mobile client on real mobile phones.

7 Problems Issues

7.1 Insufficient testing

Due to both lack of encouragement and general laziness we didn't acquire the level of automated testing that we wanted. This means that the many module in the source code still lacks unit tests and we do not have an automated system test.

This did not have any large impact on the end result as our application is rock solid anyway but we could probably have reached this goal a lot faster and with less struggle if we had done proper test driven development and had an automated system / integration test.

7.2 Delayed Access to IMS environment

Due to delays in acquiring access to the Ericsson IMS network, we had to emulate the IMS network with the help of OpenIMS, and run the AS and MAS over it. Although it provided a good enough development and testing environment for the time being, it was still a different flavor from the Ericsson version of it. Once we had shifted from the OpenIMS to Ericsson's IMS innovation environment, we had to retest the application from scratch and even fix a few bugs that never appeared in the OpenIMS. This probably happened due to the essential implementation differences in the two environments.

8 Known Unresolved Issues

8.1 Use defensive programming in the AS API

Currently the API does not check the input coming from the hosted applications. This have lead to some fairly hard to track down bugs (such as improper lists as message bodies, etc.). It would ease debugging if the API employed defensive programming techniques to ensure correct input.

8.2 Unfinished work in the MAS

- The load balancer described in previous section has not been fully tested / integrated into MAS.
- More testing for some very late discovered bugs in the message cloning functionality is still needed. Specifically a bug was found when a user leaves a group while other users still remains, resulting in the unwanted behavior that the remaining users can't get messages delivered to the group any more

8.3 Delivery Receipts

When the users send messages they can choose whether they want to get delivery receipts or not. This doesn't work right now. When the SIP-message is sent to the user a delivery-callback is set (mas_term_delivery_callback). This module creates the delivery receipt and sends it back to the sender. However something doesn't work when the delivery-callback is called.

9 Conclusion and future work

9.1 Extend the SIP/SIMPLE stack

The SIP/SIMPLE stack in the AS is very simple and only have support for a very limited part of the standard. Most interesting of these features are:

- `INVITE` support.

9.2 XCAP

Support for the XCAP protocol which would allow us to access the PGM server in the IMS service layer.

9.3 Spam filter for MAS

It would be interesting to develop a SPAM filter mechanism that would prevent unnecessary and unwanted messages being sent around to the users using services from the MAS.

9.4 User Configuration security and functionality

There is no security in the editor (userconfig.php, the "Edit User Config"-page) at the moment. The input should be checked to increase security, cleaned from any malicious input. There should be one "admin" page where the administrator(s) can change any user configuration and one user page where the users themselves can change the configuration. For both of these pages there has to be extra security through logins, no user should be able to change another users configuration, the easiest way might be to add this in the mas_config_db module. The user configuration should be expanded to be user-friendly, there is a lot of functions in the mas_config_db module that could be used to change one part of the configuration instead of changing the whole configuration at once each time you want to do an update.

9.5 Security

Many of the modules assume that the parameters and input has been checked for malicious code. This needs to be corrected if the application is to be deployed.

9.6 Removing unused tables from Configuration DB

When we started out, we stored presence and delivery receipt information in the configuration database. Later on we used HSS to get the presence and added delivery receipt information to the SIP-messages directly. However the presence and delivery receipt is still left in the configuration database. The use of these tables could be removed, it would affect the mas_config_db module, the user configuration editor and some of the test files.

A Installation / Upgrade Instructions

A.1 Erlang installation

We are using a special version of Erlang for our project since the vanilla version didn't contain everything we needed to get running. The difference between our flavor and the vanilla is that you can set permissions of the UNIX pipes generated when running Erlang in embedded mode.

A.1.1 Compiling from source code

To compile (and install) Erlang with our modifications you need to first download the source code, apply our patch and then compile it. That can be done with the following sequence of shell commands:

```
$ tar zxf otp_src_R12B-5.tar.gz # From CD
$ cd otp_src_R12B-5
$ patch -p1 < ../run_erl-pipeperms.patch # From CD
$ ./configure
$ make
```

And then the following as root:

```
# make install
```

Note that you will also need to have a number of dependencies installed (when compiling) the source code as we make use of these features in the AS and MAS.

- SSL
- ODBC
- JDK

A.2 AS installation and upgrading instructions

Download the AS Code:

You will follow the following instructions to get the system installed and real-time updated

System setup This is where we will install our final system:

```
export INSTALL_PATH=/usr/local/erl-target
groupadd erl
mkdir $INSTALL_PATH
chgrp erl $INSTALL_PATH
chmod 2775 $INSTALL_PATH
```

Now make sure that everyone that are supposed to install applications have are members of the 'erl' group. That includes the web server.

A.2.1 Compiling the AS

Before you do this you need to have done everything in the previous chapter.

```
 $ cd server
 $ make build
 $ cd ebin
 $ erl -pa .
 erl> target_system:create("app_server").
```

Installing the AS Before you do this you need to have done everything in the previous chapter.

```
$ cd server/ebin
$ umask 0002  # We need to be sure that the group has
              # write permissions.
$ erl -pa .
erl> target_system:install("app_server",
                           "/usr/local/erl-target/").

Edit /usr/local/erl-target/bin/start and add the
following lines somewhere before the last line:

editor> umask 0002
editor> export PIPE_PERMISSIONS=0660
editor> export PIPE_GROUP=erl

'erl' is the group that you created in section System
setup
```

Running the AS Before you do this you need to have done everything in the previous sections.

```
$ $INSTALL_PATH/bin/start
```

Attaching to the AS shell

```
$ $INSTALL_PATH/bin/to_erl
# From here you can run any normal erl commands.
```

Shutting down the AS

```
$ $INSTALL_PATH/bin/to_erl
erl> q().
```

Release upgrade Refer to the appup documentation on in the Erlang documentation for details. Now create a relup file:

```
$ cd /path/to/new_vsn/ebin
$ erl -pz /path/to/old_vsn/ebin -pz .
erl> systools:make_relup("app_server-<new_version>",
     ["app_server-<old_version>"],
     ["app_server-<old_version>"]).
```

Make sure both the new version and the old version of ".rel" files are placed in ebin. Create the boot script and package the release:

```
$ erl -pa .
erl> systools:make_script("app_server-<new_version>",
                          [no_module_tests]).
erl> systools:make_tar("app_server-<new_version>",
                       [no_module_tests]).
```

The .tar.gz file in ebin (named `app_server-<new_version>.tar.gz`) is ready to be uploaded to the PHP front-end. Upload the tar file to the web page, and copy the file to the release folder using the upload page. Go to the web console and unpack the release using the command:

```
web> release_handler:unpack_release(
         "app_server-<new_version>").
```

Install and run the new version of the release.

```
web> release_handler:install_release("<new_version>").
```

This will not make it permanent (the old version is still default when you restart the system). To make it permanent go to the web console and run:

```
web> release_handler:make_permanent("<new_version>").
```

Refer to the release_handler documentation for more information on what can be done with releases.

A.3 Web front-end for administration setup instructions

You will find the php files and docs at http://hg.sysrq.se/hg/www/ or you can run

```
hg clone http://hg.sysrq.se/hg/www/ tmp\begin{verbatim}
```

in the shell. Here is the instructions about how to do it:

```
Web system setup
================
# export INSTALL_PATH=/as/uploads
# mkdir -p $INSTALL_PATH
# chown lighttpd:erl
# chmod 2775 $INSTALL_PATH

Web server configuration
========================
Make sure that the user that the web server uses
(lighttpd in the above example) is a member of the
group that's used in the AS setup (erl in the above
example). To make it a member execute:

# usermod -a -G erl lighttpd

Make sure that PHP and PEAR is installed and that
upload_max_size and post_max_size in php.ini are large
enough for the files that you want to upload. You must
also turn off "magic_quotes_gpc".
```

A.4 MAS Installation and Upgrading

A.4.1 Installing

The installation of the MAS is very similar to the installation of the AS

```
$ make build
$ cd ebin
$ erl -pa . -pz <Path to AS ebin directory> \
      -sys config
# This starts an erl shell in which we create and
# install the application.
erl> target_system:create("mas_app").
erl> target_system:install("app_server",
                           "/usr/local/erl-target/").

Running the MAS
==============
$ $INSTALL_PATH/bin/start

Attaching to the MAS shell
=========================
$ $INSTALL_PATH/bin/to_erl
```

```
# From here you can run any normal erl commands.

Shutting down the MAS
====================
$ $INSTALL_PATH/bin/to_erl
erl> q().
```

A.4.2 Upgrading

The MAS upgrade follows the OTP Upgrade guidelines, to get more information on how it works please check the OTP Design Principles.

Before you can start an upgrade you need to have four files. These are

Release resource file
: The .rel file, contains information about which applications it relies on. Also version information. It is named mas-¡version¿.rel.

Application resource file
: The .app file and contains version, registered modules and information how to start/stop the application. It is named mas.app for all releases, but each release will have its own mas.app.

Application upgrade file
: The .appup file, this file describes how to upgrade/downgrade between versions.

System configuration
: The sys.config file, this is copied automatically when you do "make build"/"make", it contains information on what S-CSCF to use and where to listen for the app_application_api (the application server interface).

The normal MAS release upgrade allows upgrade and downgrade to/from only the earlier version and that is what we describe here.

This assumes a new .rel-file is created

```
$ cd /path/to/new_vsn/ebin

$ erl -pa . -pz <path to old versions ebin directory> \
      -pz <path to app_server_api ebin directory> \
      -sys config
erl> systools:make\_relup("mas-<new\_version>",
          ["mas-<old\_version>"],["mas-<old\_version>"]).
```

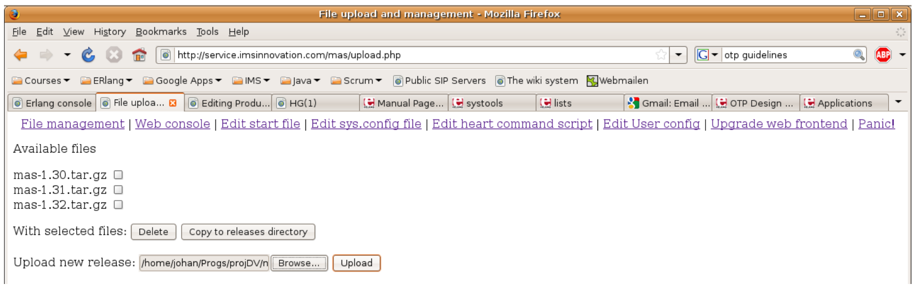
Create the boot script and package the release:

```
erl> systools:make_script("app_server-<new_version>",
                          [no_module_tests]).
erl> systools:make_tar("app_server-<new_version>",
                       [no_module_tests]).
```

The .tar.gz file in ebin (named `mas-<new_version>.tar.gz`) is ready to be uploaded to the PHP front-end. Upload the tar file to the web page, and copy the file to the release folder using the upload page in figure 13.

Don't forget to copy the package to the releases directory.

Go to the "Web Console" page and unpack the release using the command:


File upload and management - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://service.imsinnovation.com/mas/upload.php
otp guidelines
Courses ERlang Google Apps IMS Java Scrum Public SIP Servers The wiki system Webmailen
Erlang console | File uploa... | Editing Produ... | HG(1) | Manual Page... | systools | lists | Gmail: Email ... | OTP Design ... | Applications

File management | Web console | Edit start file | Edit sys.config file | Edit heart command script | Edit User config | Upgrade web frontend | Panic!

Available files

mas-1.30.tar.gz ☐
mas-1.31.tar.gz ☐
mas-1.32.tar.gz ☐

With selected files: Delete | Copy to releases directory

Upload new release: /home/johan/Progs/projDV/n | Browse... | Upload

Figure 13: The release upload feature in the MAS web front-end

```
web> release_handler:unpack_release("mas-<new_version>").
```

Install and run the new version of the release.

```
web> release_handler:install_release("<new_version>").
```

This will not make it permanent (the old version is still default when you restart the system). To make it permanent go to the web console and run:

```
web> release_handler:make_permanent("<new_version>").
```

Refer to the release_handler documentation for more information on what can be done with releases.

A.4.3 Configuration files

These are examples of configuration files.

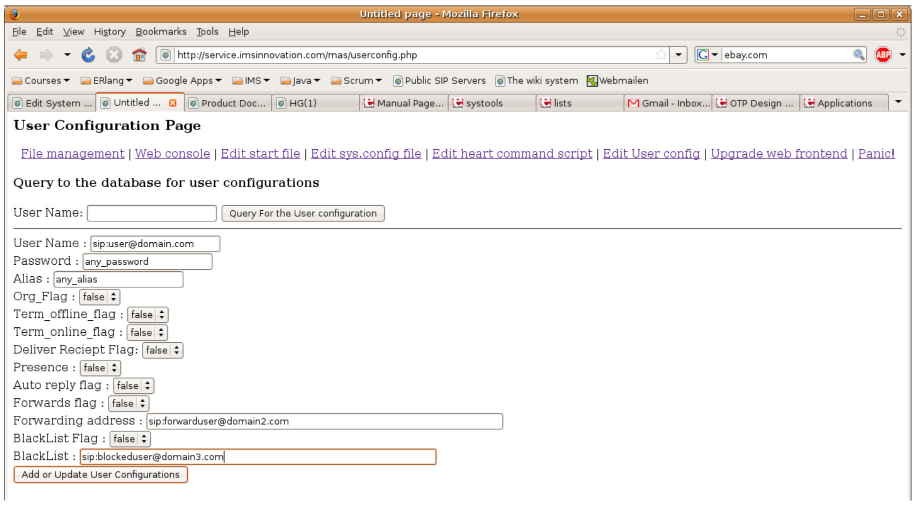
System Configuration Called sys.config, there should be a copy in the src directory

```
[ { app_server_api,
  [
  { host, "localhost" },
  { port, 8001 },
  { scscf, "sip:172.23.214.27:5063" }
  ]
    }
].
```

The app_server_api tuple contains vital information to access the app_server_api, namely the address and port the api listens to. The information should be mirrored from the information in the AS sys.config.

A.4.4 MAS Web Front-end

If you got the mas from the repository you should have a sub-directory called "web", in there you'll find the web pages. The front-end is installed by copying it into the web directory.


Untitled page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://service.imsinnovation.com/mas/userconfig.php

User Configuration Page

File management | Web console | Edit start file | Edit sys.config file | Edit heart command script | Edit User config | Upgrade web frontend | Panic!

Query to the database for user configurations

User Name: [ ] Query For the User configuration

User Name : sip:user@domain.com
Password : any_password
Alias : any_alias
Org_Flag : false
Term_offline_flag : false
Term_online_flag : false
Deliver Reciept Flag: false
Presence : false
Auto reply flag : false
Forwards flag : false
Forwarding address : sip:forwarduser@domain2.com
BlackList Flag : false
BlackList : sip:blockeduser@domain3.com

Add or Update User Configurations

Figure 14: User configuration page on the MAS web front-end.

A.4.5 User configuration Manual

For the users to be able to customize the application behavior there has to exist user configurations for each user.

If its the first time you use the application, you have to add the user to the configuration database. To do this, enter the user information into the fields on figure 14.

User name
The username for the user.

Password
This password is intended to be used when the user wants to update his/her user configuration, not used for now.

Alias
Unused for now.

Org_Flag
This boolean value states whether the user wants to have email_copy enabled on the originating side.

Term_offline_flag
This boolean value states whether the user wants to have email_copy enabled on the terminating side when the user is offline.

Term_online_flag
This boolean value states whether the user wants to have email_copy enabled on the terminating side when the user is online.

Deliver Receipt Flag
Deprecated, set in the client for now. The user has to be set all the same.

Presence
Also deprecated. The presence is gotten from the as module.

Auto reply flag
: States whether the user wants to have auto replies active. It is possible to set the body of the auto-reply, but this isn't supported through the user configuration page.

Forwards flag
: Boolean to show if forwarding is enabled or not.

Forwarding address
: Where to forward.

Blacklist Flag
: If blacklist is enabled.

Blacklist
: This list states which sip addresses who are blocked from sending messages to the users.

The simplest way to change a users configuration is to make a query on the user, if the user is available in the db the fields are populated and one can change the values and press ”Add or Update User Configuration”.

B User Manual

B.1 Application Server API

The AS offer an API to the hosted applications, this API can be found in the application app_server_api and it includes the following modules and behaviors.

`ims`
: The main module, it is here that each application register for use of the API and create instances of each other service.

`uac_message`
: Module for sending SIP page instant messages.

`uas_message`
: Module for retrieving information about a received message.

`uac_email`
: Module for sending e-mails.

`ims_application (behavior)`
: A behavior that each hosted application should implement, it define functions for receiving SIP messages.

`ims_instant_message (behavior)`
: A behavior for receiving status report of sent instant messages (whether they arrived or not).

Overall the API is designed to have an overall look and feel similar to the one provided by the Sailfin framework developed by Ericsson. Following this tenet the API make use of the observer pattern heavily. This means that almost all operations are asynchronous with an optional setting of a callback module that can be used to retrieve additional information.

B.1.1 ims

The main responsible module that takes care of creating instances of the other or registering applications.

`register_application(AppName, Callback, IARI)`
: Establish a connection to the AS and register an application named `AppName` which is identified by the IARI `IARI`. Any received messages is sent to the `Callback` module which should implement the *ims_application* behavior.

`unregister_application(AppName)`
: Close the connection to the AS for the application registered under the name `AppName`.

`is_registered(AppName, SipURI)`
: Check whether the user `SipURI` is online.

`create_instant_message(AppName)`
: Create and return an instance for the module **`uac_message`**.

`create_email(AppName)`
: Create and return an instance for the module **`uac_email`**.

B.1.2 uac_email

Module responsible for sending e-mail messages, the `ID` argument is acquired through the use of `ims:create_email/1`.

`set_sender(ID, Sender)`
: Set from whom the e-mail should be addressed.

`set_receiver(ID, Receiver)`
: Set to whom the e-mail should be addressed.

`set_subject(ID, Subject)`
: Set the subject of the e-mail.

`set_body(ID, Body)`
: Set the body of the e-mail.

`send(ID)`
: Send the e-mail.

B.1.3 uac_message

Module responsible for sending SIP instant page messages, the `ID` argument is acquired through the use of `ims:create_instant_message/1`.

`set_sender(ID, Sender)`
: Set from whom the message should appear to come from. (Since the AS is a trusted entity in the IMS network you can send from anyone, even none existing users).

`get_sender(ID)`
: Get from whom the message is addressed.

`set_receiver(ID, Receiver)`
: Set to whom the message should be delivered.

`get_receiver(ID)`
: Get to whom the message is addressed.

`set_body(ID, Content, ContentType)`
: Set what the body of the message should contain and what type it should be recorded as.

`add_header(ID, Key, Value)`
: Add an extra SIP header to the message.

`set_raw(ID, RawSIP)`
: Set the raw SIP message to send. Note that if you use this function you override any changes made using other methods.

`set_callback(ID, Callback)`
: Set the callback module of the message, this module will be called when we know the destiny of the message (e.g. `200 OK` or `404`).

`send(ID)`
: Send the message.

B.1.4 uas_message

Module responsible for receiving information about a received SIP message. The `ID` argument is acquired through the `ims_application` behavior.

`get_sender(ID)`
: Get from whom the message was addressed.

`get_receiver(ID)`
: Get to whom the message was addressed.

`get_session_case(ID)`
: Get the session-case of the message.

`get_expires(ID)`
: Get the expiring date of the message.

`get_body(ID)`
: Get the content of the message.

`get_request_uri(ID)`
: Get the Request-URI of the message.

`get_header(ID, Key)`
: Get the first matching header in the message.

`get_headers(ID, Key)`
: Get all matching headers in the message.

`get_raw(ID)`
: Get the raw SIP message.

`received(ID)`
: Send a `200 OK` back in response to the message.

`reject(ID)`
: Send a `400` back in response to the message.

`respond(ID, Code)`
: Send a `Code` response back.

`forward(ID)`
: Forward the message using SIP proxy behavior.

B.2 Source Code documentation

To generate the source code documentation for either of the projects type

```
make all_doc
```

in their source code directories. This will create the documentation in HTML format and store it in either the `doc/` or the `docs/` directory.

References

[1] http://tools.ietf.org/html/rfc3588

[2] http://tools.ietf.org/html/rfc3261

[3] http://www.3gpp.org/ftp/Specs/html-info/29328.htm

[4] http://www.3gpp.org/ftp/Specs/html-info/29329.htm

[5] http://www.3gpp.org/ftp/Specs/html-info/23008.htm

[6] http://www.selenic.com/mercurial/wiki/

[7] http://www.trapexit.org/

[8] http://www.imsinnovation.com/