NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR MATEMATIKK, INFORMASJONSTEKNOLOGI OG ELEKTROTEKNIKK

HOVEDOPPGAVE

Kandidatenes navn: Svein Løvland og Audun Mathisen

Fag: Datateknikk

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk): Mobile Instant Messaging – Extending Jabber to Support Mobility

Oppgavens tekst:

This work will engineer a software system prototype for instant message exchange. The system is designed and implemented using Jabber, extended to support mobility.

Several areas are to be investigated:

- handling of very thin clients, by reducing client processing and communication
- multiple simultaneously active clients for single person
- integration of location information in presence/awareness protocol
- support for both transparent and opaque messaging wrt. recipient, target device and location

The project will be based on exchange of XML messages, using the Jabber XML middleware and messaging protocols. This will ensure interoperability with existing Jabber clients, and the other IM systems that Jabber supports.

The thesis will include a study of the Jabber architecture leading to an improved system architecture and implementation, extended to support mobility.

The architecture of the system will be used as an example in the context of the software architecture course SIF8056. This case study will comprise at least an architecture assessment session to which both students and system architects will participate.

Oppgaven gitt:	20. januar 2002
Besvarelsen leveres innen:	17. juni 2002
Besvarelsen levert:	14. juni 2002
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veiledere:	Letizia Jaccheri og Hallvard Trætteberg

Trondheim, 14. juni 2002

Letizia Jaccheri
Faglærer

Preface

This master thesis was worked out at the Norwegian University of Science and Technology, Department of Computer and Information Science in the period January to June 2002. The background for the master thesis is the project ‘Java 2 Micro Edition – Technology Driven Architecture Work’, which we conducted during the fall 2001. Results of this thesis were an instant messaging (IM) system which included a client application for J2ME devices. However, we wanted to investigate what changes has to be done when making IM mobile beyond simply developing clients for mobile devices.

Results of this thesis include suggestions for extensions which can be implemented in an IM system to support mobility. As a proof of concept we developed the SIGN prototype. SIGN consists of a modified Jabber server along with client applications for mobile phones, PDAs and desktop PCs. Although SIGN is a prototype we believe it provides a platform on which further work can be done.

The architecture developed during the work was used in an assessment session in the software architecture course SIF8056, given by our advisor associate professor M. Letizia Jaccheri. Although the session took on a different form than intended, we presented our work to the students, resulting in a discussion of our proposed architecture and design. The session provided valuable suggestions for areas of improvement.

We wish to thank our advisors, associate professor M. Letizia Jaccheri for valuable insights and comments and PhD. student Hallvard Trætteberg for a seemingly endless repository of ideas for our work.

Trondheim, June 14. 2002

Svein Løvland
sveinlo@idi.ntnu.no

Audun Mathisen
audunma@idi.ntnu.no

Abstract

To extend instant messaging (IM) with support for mobility, merely providing clients for wireless devices is not sufficient. Characteristics of wireless devices and networks, and the context in which the system is used affects the users' ability to communicate. This suggests that new functionality should be introduced to make IM useful and efficient in a mobile setting.

In this master thesis we have studied how support for mobility can be introduced in IM systems. New features and functionality for this purpose is demonstrated in a prototype implementation of an IM system with mobility support – SIGN. The open-source software Jabber and its open XML message protocol is used as a framework for basic IM functionality. Upon this framework we added the functionality found important to support users on wireless devices. SIGN provides client applications for mobile phones, PDAs and desktop PCs, developed with Java 2 Micro Edition, PersonalJava and Java 2 Standard Edition, respectively.

Providing users with extended awareness information is a fundamental feature of the SIGN prototype. Mobile users can be in different environments and settings not captured by the awareness information available in existing systems. The awareness information in SIGN is structured along three independent axes – presence, device and context. Extended awareness information is important to improve communication between users in mobile environments.

In SIGN, control is pushed the towards the recipient rather than the sender. The user can decide *where* and *how* messages shall be delivered. Limited device capabilities and cumbersome input mechanisms led to a differentiation of the functionality among the client applications. The desktop client serves as an interface to setting preferences and controlling the system, whereas the wireless clients only offer basic awareness and IM functionality. Due to the low bandwidth and the payment model of wireless networks, the user can limit the bandwidth usage by applying restrictions on messages destined for the wireless clients.

We studied the Jabber software architecture and design before incorporating our changes and extensions to Jabber. The Jabber reference architecture, with emphasis on component-based design and external as well as internal XML routing, is meant to facilitate addition of functionality through components. However, our proposed extensions were not easily incorporated at an architectural or component level. We were thus faced with the option of either duplicating existing functionality in new components or make changes in the source code. We found the latter approach most suitable to implement the extensions within the scope of this thesis. The Java implementation of Jabber used in this master thesis proved to be immature and not true to the important Jabber concept of component based design.

The XML based Jabber message protocol was easily extended to our needs, but required low-level changes on the server. Parsing the messages on the wireless clients caused no problems, as the messages exchanged in an IM system can be characterized by small size, low complexity and low frequency.

Contents

List of Figures

List of Tables

Introduction

Since the late 1990's instant messaging (IM) services on the Internet have become increasingly popular. The allow people connected to the Internet to communicate with family, friends and colleagues all over the world and be notified of their current presence. One of the first of such services, ICQ, was launched in November 1996. It grew quickly to be a popular service and only six months later 850 000 users had registered. Today several similar services are available and used by millions of users all over the world.

IM systems provide awareness information and by checking the system one can determine the availability of other users. "Online", "Do not disturb" and "Away" are examples of presence states a user can set and thereby communicate to others. There has been huge interest in developing systems for collaborative work, providing people in business organisations with information about colleagues. Knowing if a colleague is busy in a meeting or available in the office can make it easier to know when to place a call or make a visit to the office. Although many systems have been developed aiming to support collaborative work, instant messaging services are often used instead or in addition.

In recent years the proliferation of powerful mobile devices has enabled the use of such systems while on the move. Several IM services now provide clients that can run on wireless devices. However, little has been done to make IM mobile beyond simply offering clients for wireless devices. We believe this is insufficient as the ability to communicate may be reduced due to characteristics of the wireless devices, such as cumbersome input methods and small screens. Further on, an IM system supporting mobility enables use in environments that affect the user's ability to read and write messages.

In our work we have identified several extensions which we believe will offer better support for mobility in IM systems. To demonstrate these concepts we have implemented a prototype IM system called SIGN. Using the open source software Jabber as basis we have implemented our suggested extensions. SIGN offers client applications for mobile phones, PDAs and desktop PCs.

This master thesis presents the results of our study of IM systems, suggestion for improved mobility support and how we proceeded with the prototype implementation. The latter includes a comparison of two available Jabber implementations and how we changed the architecture and design to realize the SIGN prototype.

A part of our work consisted of using our architecture and design in the software architecture course SIF8056, where we conducted an architecture evaluation session in collaboration with the students.

The following sections give a short description of the content of each of the chapters in this report.

Chapter 1 - Project Description
This chapter outlines the work to be performed in this master thesis. It gives a short introduction to the problem, our research questions and our work process.

Chapter 2 - Background
This chapter presents background information on four different topics which are important to the work of this master thesis.

Chapter 3 - The SIGN Prototype
This chapter presents the conceptual model of the SIGN prototype we have developed. Further on, the main areas of the model are discussed.

Chapter 4 - System Requirements
The chapter presents a discussion on the main functional requirements for the SIGN prototype. The requirements specification which is the basis for this discussion is presented in appendix A.

Chapter 5 - Architectural Choices
Two different Jabber implementations, *jabberd* and Jaba, are presented and compared in relation to architecture and design. Subsequently architectural choices for extending Jabber are discussed.

Chapter 6 - Design
In this chapter we present the architecture and design of the SIGN prototype, including both server and the different client applications.

Chapter 7 - Implementation and Deployment
A detailed description of server– and client implementations is presented in this chapter. In addition we give a description of how the system can be deployed.

Chapter 8 - Evaluation
In this chapter we evaluate the work done in this master thesis.

Chapter 9 - Related Work
This chapter presents four different efforts similar to the SIGN prototype.

Chapter 10 - Conclusion
The conclusion describes the important results from this thesis and suggestions for further work.

Chapter 1

Project Description

1.1 Introduction

Today some instant messaging (IM) services support mobility by providing IM clients on PDAs. Some services also offer support for Short Message Service (SMS) to enable users with mobile phones to interact with their IM service. In supporting mobility the IM services will be used in situations and environments were IM earlier has not been available. When sending messages or chatting with another user one could earlier presume that this user was by his desktop computer at the office or home. With the support for mobility this is no longer true, as the user now can be anywhere using a wireless device.

Many of the IM services now supporting mobility offer their IM clients on wireless devices but have done little about the functionality of the service. We believe that in order to provide true mobility to IM services merely offering clients for wireless devices is not enough. Making IM mobile will affect the communication among users on several areas, thus the IM service should be enhanced with functionality supporting the use of IM in a mobile environment. Below is a short summary of the key issues that should be addressed when making IM mobile.

Wireless devices and IM clients To support mobility the first problem to be solved is enabling access to IM from wireless devices. This is to some extent provided by some IM services today. However this often includes the use of technologies such as SMS, telephony and WAP.

Awareness information Enabling the user to use IM away from the desktop computer means that the user can be anywhere and still connected to the IM service. This should be reflected by the awareness information provided by the IM system.

Multiple devices Going from one device, the desktop PC, to many different devices introduces a range of new aspects. One of the major tasks is handling the interaction between the different devices and the transition when the user moves from one device to another. Due to the different nature of the wireless devices, functionality might also differ across devices.

1.2 Research questions

Our work is to engineer a software system prototype for instant messaging. This prototype shall provide users with access to IM from wireless devices and networks. Below we present the research questions we have stated for this project.

1. How can existing IM functionality be extended to support mobility and make IM useful in a mobile environment?
2. What new features should be introduced to support users in mobile environments and settings?
3. How can IM be implemented on wireless devices used in every day life?

In this project we shall use Jabber to support basic IM functionality. Jabber is an open source instant messaging system and it implements an open XML message protocol.

4. What changes are necessary to the architecture and design of an existing IM system, Jabber, to extend the system with support for mobility?
5. Is XML a suitable message format for communication using thin, wireless clients?
6. How can the existing Jabber XML message protocol be extended to support mobility?

During our work with Jabber we will in addition investigate if Jabber is a suitable framework for further work on IM and development of systems based on XML.

1.3 The SIGN prototype

The SIGN prototype that will be developed in this thesis is a system extending instant messaging to wireless devices and aims to solve the issues briefly discussed in previous sections. The system will provide basic IM functionality and clients for various wireless devices will be developed. More specific, IM clients will be developed for mobile phones, PDAs and desktop PCs. The awareness information in the system will be extended to reflect users in a mobile environment and the system will be enhanced with functionality supporting the use of wireless devices.

1.4 Prerequisites

Some prerequisites are given in this master thesis and they are presented below.

Jabber - Open source software

One of the prerequisites of this project is to use the open source software Jabber. Jabber is server software providing, among others, basic IM functionality.

One important aspect of Jabber is enabling Jabber users to communicate with users of other commercial IM services. In fact this was one of the main reasons for the initialisation of the Jabber project - “finding a solution to the soup of incompatible IM systems that meant that it was necessary to run a whole host of different programs just to talk with friends”. Jabber offers interfaces to the major IM services and translates the Jabber message protocol to the specific service and vice versa.

XML message protocol

The open Jabber XML message protocol shall be used. XML is the only information exchanged between the clients and the server in Jabber meaning that any device capable of making a connection over an IP network and processing XML can in principal be made into a Jabber client. Interoperability with existing clients shall be ensured, but extensions to the protocol can be made to implement new functionality and extend the awareness information.

For limited devices such as mobile phones it might be necessary to use a message protocol other than XML, one that is less resource demanding.

Wireless devices and J2ME

IM clients shall be developed for several wireless devices where one client shall be developed using the language J2ME. Other languages may be used for development at the server side and for desktop and PDA client development.

1.5 Main focus and evaluation

The focus in this master thesis will be on defining an IM system supporting awareness and mobility and implement a working prototype of the system. In addition to demonstrating IM mobility concepts it will be a system that can be used and extended in further studies on IM, mobility and awareness.

We will not conduct usability tests of large scale. During development of the prototype we will continuously test the system, but this can only establish that the system behaves according to the specification

To demonstrate the functionality and test that the system behaves as expected we will, upon prototype completion, run through the scenario described in the following section. This description is only a short summary of the scenario and the actual test, detailed description of different actions and the results will be presented in chapter 8.

1.5.1 User scenario

The scenario we intend to run through to test and demonstrate the system is depicted in figure 8.1.

Three devices will be used during the run of the scenario. This is a J2ME enabled mobile phone, a PDA and a desktop PC. The scenario will simulate a day

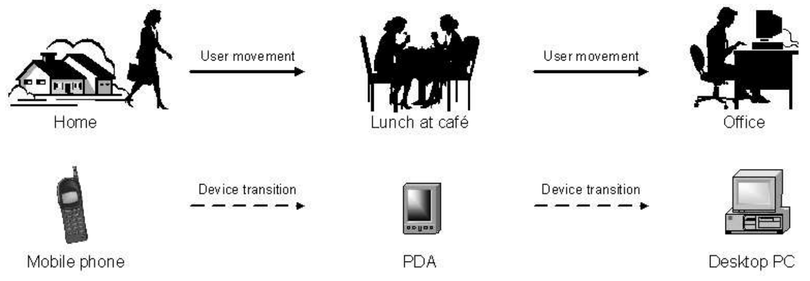


Figure 1.1: Scenario of use

where the user starts out at home in the morning connected with her mobile phone. She then leaves home for a lunch meeting at a café. At the café she uses her PDA. Later she arrives at the office and continues the day in front of her desktop PC.

During this time a series of events will occur, triggering different actions in the IM service. Messages will be sent and received and awareness information will change.

1.6 Process

This section defines how we intend to proceed in our work with this thesis. First we present the process of defining the functionality SIGN shall provide, then how the architecture of the system is developed and finally how we will implement the system.

The SIGN prototype

Before developing the SIGN prototype, we define a conceptual model for the system. The challenge is to identify the problem areas which need to be addressed when making an IM system mobile. Based on the conceptual model and proposed solutions to the problems, we arrive at the functional requirements to the system.

Some of the concepts which shall be investigated are stated in the description of the thesis. Others will result from a study of existing IM systems and our own experience with use of instant messaging. Further on a study of literature on the subjects of instant messaging, mobility and awareness will be conducted in order to serve as inputs to this task. The conceptual model for SIGN along with other key-points, is presented in chapter 3.

Based on this chapter, we will specify the requirements to the different parts of the SIGN prototype. Chapter 4 gives a motivation for these requirements and the full requirement specification is listed in appendix A.

SIGN architecture and design

As stated in the thesis description, the Jabber XML middleware shall be used in the system prototype. Once the requirements to the system are established, the system architecture shall be made such that the requirements can be incorporated into Jabber. In order to accomplish this, a thorough study of the Jabber architecture must be performed. However, as there are two available Jabber implementations – the reference implementation *jabberd* and a Java implementation Jaba – both of these will be studied in order to choose which one is best suited for the purpose. The results of this study are presented in chapter 5.

The architecture for the server-part of the system will depend on the architecture of the chosen Jabber version. Regardless of which we choose, focus will be on minimizing the changes to the existing architecture and design. As for the client applications, the key issue will be isolation of the Jabber-related parts of the design.

In addition to being presented in chapter 6 of this thesis, the resulting architecture shall be communicated to the students in the software architecture course SIF8056, given at NTNU. The slides from our presentation to the students can be found in appendix B.

Implementation

The SIGN system will consist of four different applications: the Jabber based server and client applications for mobile phones, PDAs and desktop PCs. We plan to have basic[1] client applications up and running at an early stage of our work, before the requirements to the system are settled and an architecture for the clients has been developed. In our J2ME project [17] we developed IM clients which exchanged XML with the server, and they are therefore a natural starting point for the client applications in SIGN.

By experimenting with the clients and the server software, we will investigate possible solutions for extending Jabber with the additional functionality needed by SIGN.

The basic client applications will be fully implemented when the client requirements and architecture is made.

Although we plan to base our development on prototyping, we do not intend to follow any formally defined software development processes. Chapter 7 describes details of the server and client implementation and deployment.

[1]With basic we mean applications that can connect to a Jabber server, but with very limited functionality and user interface.

Chapter 2

Background

2.1 Introduction

In this chapter we present background information on four different topics which will be important to the work of this thesis. We start off by giving a short description of a project which we conducted during the fall 2001 – ’J2ME - Technology Driven Architecture Work’ [17]. Java 2 Micro Edition (J2ME) was used in an XML-based instant messaging system and the lessons learned from the project will be applied in this thesis.

The following section provides an introduction to the Jabber middleware and the Jabber message protocols, which will be a fundamental part in our work.

Then follows an overview of the four most popular instant messaging services on the market – ICQ, AOL Instant Messenger, MSN Messenger and Yahoo! Messenger. The section gives a short description of the functionality of these services, their availability on wireless devices and finally the presence information offered. Knowledge of existing services is important when specifying the functionality of SIGN.

A section on wireless devices concludes this chapter. First we present characteristics of different categories of devices, then two different Java environments targeted at wireless devices and finally some properties of wireless networks which are fundamental for using wireless IM clients.

2.2 J2ME project

This master thesis is based upon the project ’J2ME - Technology Driven Architecture Work’ [17], which was written in the ninth semester course ’SIF8094P1 Prosjektarbeid’ at the Department of Computer and Information Science, NTNU. The goal for the project was to use a technology as a starting point and investigate what implications use of that specific technology had on a system’s architecture and development process. Our chosen technology was Java 2 Micro Edition (J2ME).

2.2.1 MobileIMS

In addition to an evaluation of J2ME, a part of the project consisted of using J2ME in an implementation of a software system, and designing the system to take into account the possible constraints use of J2ME had on the system. We developed an instant messaging system called MobileIMS, using J2ME to implement clients for Java enabled phones and PalmOS PDAs.

One of the goals for MobileIMS was to differentiate the client applications according to device characteristics and capabilities, with regards to screen-size, processing power and memory budget. In other words, the more capable devices should provide more functionality compared to less capable devices.

As the system itself was not the main focus of the project, the functionality was restricted to basic messaging, contact list management and distribution of presence information.

The server part of the system was written using Java 2 Standard Edition (J2SE). The non-functional requirements to the server were heavily relaxed, as the main goal was to pinpoint implications imposed by the J2ME clients. Further on, persistent data storage, such as user accounts and contact lists, was not implemented as we considered it to be beyond the scope of the project. As a consequence of this, the resulting system was nothing more than a rudimentary prototype of an instant messaging system.

2.2.2 XML in MobileIMS

Messages exchanged between the MobileIMS server and clients were XML formatted. Use of XML has several advantages, such as platform independence and ease of use, but comes at the cost of processing requirements and bandwidth usage. During our work we found three different XML parsers for J2ME. We ended up using a parser called kNanoXML, which is a small, non-validating DOM parser.

The MobileIMS message protocol was not written in accordance with any existing standards, as it was kept as simple as possible.

2.2.3 Project conclusions

The main conclusion of the project was that J2ME in itself, had less architectural implications on the system than we had expected. Rather, the limited capabilities of the J2ME devices constituted for most of the implications. However, as J2ME was designed to run on wireless devices with limited capabilities, one must take the implications into account when developing a system where J2ME is used.

At the start of the project we were uncertain if the XML-parsing would be too processor intensive for the relatively limited mobile phones. We concluded that this was not the case, however we found XML to be rather bandwidth-intensive when used over a GSM data connection, which is only capable of 9600 kbps. However, GPRS is gradually replacing GSM for data traffic in the mobile networks, and with the higher bandwidth in GPRS bandwidth usage becomes less of an issue. Further on, IM services are hardly usable in a circuit switched network like GSM anyway, as the cost of staying connected for longer period becomes too high. A packet switched network like GPRS is on the other hand a perfect match for IM systems, as the user pays for the amount of data transferred rather than the time he is connected.

2.3 Jabber

The Jabber Communications Platform is an open-source, XML-based platform which provides basic networking and communication services. One application available on the platform is the Jabber instant messaging service, which provides standard IM functionality, such as messaging, contact lists management and distribution of presence information. Jabber will be used as the core part of the server in SIGN.

There were several reason why Jabber was chosen for this project. The first and most obvious reason is the advantages associated with using existing components. We could have continued work on the server developed in the MobileIMS system, but using Jabber allows us to keep focus on the key areas as described in section 1.1.

The second reason is that Jabber is open-source, which allows us to extend the server's functionality to deal with multiple device categories and provide richer awareness information.

The fact that Jabber is based on XML is another reason as we had experience with XML from the MobilIMS system. In addition XML allows platform independence.

Finally, one reason for choosing Jabber was to get a better understanding of emerging XML based technologies.

The following subsections will present the Jabber message protocol and other parts of the Jabber platform.

2.3.1 Jabber message protocol

The Jabber message protocol is an open, XML based protocol managed by the Jabber Software Foundation (JSF).[1]. The fact that it is open enables third party developers to develop their own client applications for any platform capable of parsing XML messages. This is in contrast to the IM services [13, 2, 23, 31], which use periodically changed and proprietary message protocols to prevent unauthorized client applications to be developed.

The Jabber protocols have been submitted by the JSF to the Internet Engineering Task Force (IETF) for consideration as an Informational RFC [20]. Port 5222 is also registered by Internet Assigned Numbers Authority (IANA) as the standard port for Jabber.

The different entities of the Jabber architecture pass data to each other using XML streams, which is essentially the exchange of data in the form of XML fragments in "streaming" mode over a network connection. During the lifetime of a connection between two entities, a complete XML document will be sent from each entity to the other, one fragment at a time.

There are three top-level XML fragments exchanged between the entities: `<message/>`, `<presence/>` and `<iq/>`, each containing attributes and child nodes. These elements are described in then following subsections. For more details see [1].

Message element

The most fundamental part of an instant messaging system is the ability to exchange messages between the user, and the `<message/>` element provides this

[1]see http://www.jabber.org/jsf.html

facility. Each message has one or more attributes and sub elements, which are presented below.

The attributes `to` and `from` contain the sender and recipient addresses, respectively. `<message/>` can also contain a `type` attribute, which gives an indication to the recipient as to what sort of content the message contains. Table 2.1 presents the possible values and a short description of their meaning. An `id` attribute may

Table 2.1: Jabber message types

Value	**Description**
normal	The normal message type used for simple messages that are often one-time in nature. Similar to an e-mail message.
chat	Chat messages which usually carry a message that is part of a live conversation.
groupchat	The groupchat message type is used to notify the user that the message comes from a conference room.
headline	Headline messages are designed to carry news style information, often accompanied by a URL.
error	Error messages signifies that the message is conveying error information to the client.

optionally be included as well, and is used to uniquely identify a response to an outgoing message.

Sub elements of `<message/>` include `<subject/>` which is used to carry a message subject for `normal` type messages, but is usually not used for other message types. The `<body/>` sub element carries the body of the message. If the message type is `error`, the `<error/>` sub element contains the error message. Messages can optionally contain a `<thread/>` sub element, which contains an identification to group together chat messages belonging to one specific conversation.

Presence element

The `<presence/>` element is used to convey awareness of a user's presence state. Users can either be *available* or *unavailable*. In the first case, the user is currently connected and messages destined for him are delivered immediately. In the latter case, the user is not connected and messages are delivered the next time he connects to the server. The user himself controls the availability information through the client application. However, when the user disconnects, the server will communicate the user's unavailability if he has sent his availability beforehand.

The availability information for a particular user is distributed to other users that *subscribe* to that user's availability information. Each user can require to authorize other users before they can subscribe to his availability information.

In addition to the availability information, presence messages can also be qualified with more detail. The two sub elements `<show/>` and `<status/>` are used for this purpose. The `<show/>` tag can contain one of the values listed in table 2.2. The show value is often presented graphically using different icons in the client applications. The `<status/>` tag often contains a textual description

Table 2.2: Possible <show/> values

Value	**Description**
normal	The normal situation where the user is available. If the <show/> tag is not present, this is used as the default value.
chat	Similar to normal, but in addition indicates that the user is open to conversation.
dnd	Dnd stands for "Do not disturb" and means that the user is available but does not want to be disturbed. Messages are, however, immediately sent to the user.
away	The user is temporarily away from the client
xa	xa stands for "extended away" and indicates that the user will not return to the client for some time.

of the *show* mnemonic, for instance "Extended away" when show is *xa*. However it can also contain a value set by the user such as "Out to lunch!" when the *show* value is set to *away*.

IQ element

The IQ element is the third of top-level XML elements in the Jabber message protocol. IQ stands for "info/query" and is used for sending and retrieving information between entities. IQ messages are exchanged as a request/response mechanism similar to the GET and POST mechanism in HTTP.

An IQ element can be in one of the four states described in table 2.3. The IQ element is used for various purposes and to distinguish between them, namespaces are used in each IQ message. There are too many namespaces to list here, but the most commonly used are namespaces for registration, authentication and contact list management. For a detailed description of the various uses of the `<iq/>` element and the different namespaces see [1].

Table 2.3: IQ States

State	Description
get	Request information.
set	Set information
result	Show the result if the *get* or *set* was successful.
error	Specify the error if the *get* or *set* was not successful.

2.3.2 Resources and priorities

In Jabber each online client is attached to a resource and each resource has an associated priority. The resource is a string appended to the *jid*, for instance `peer@vinstra.net/mobile`, used by the server to distinguish between the connections. The priority is a positive integer used by the server to decide which resource, or client, a message shall be sent to. Larger numbers have higher priority, which means that a resource with priority 5 outranks a resource with priority 1. In the event of a tie between priorities, messages are sent to the most recent connection.

2.3.3 The *jabberd* server

The *jabberd* server is the original server implementation of the Jabber protocols for instant messaging and XML routing. *jabberd* is written in C for Linux and Unix, and is both open-source and free software. The project was started by Jeremie Miller in 1998, and has since then been worked on by the Jabber community to the current version, which at the time of writing, is 1.4.2.

Distributed architecture

The Jabber network architecture is modelled after the e-mail system where users have *one* server – the home server – that they connect to. User accounts and user data, such as contact lists and preferences, are stored on the home server. User identities are unique for each server, and by appending '@' and the server address, globally unique user identities are created, ex. *peer@vinstra.net* where *peer* is the local unique name and *vinstra.net* is the server address. These identities are called Jabber IDs or *jids*.

An instance of *jabberd* is running on the server jabber.org, where anybody can sign up for a user account. However, anybody can set up their own Jabber server that exchanges information with other servers over the Internet through a component called *Etherx*. This component communicates with the local Jabber server and remote Etherx components. When a user sends a message to a user on a different home server, the message is forwarded as shown in picture 2.1. In this

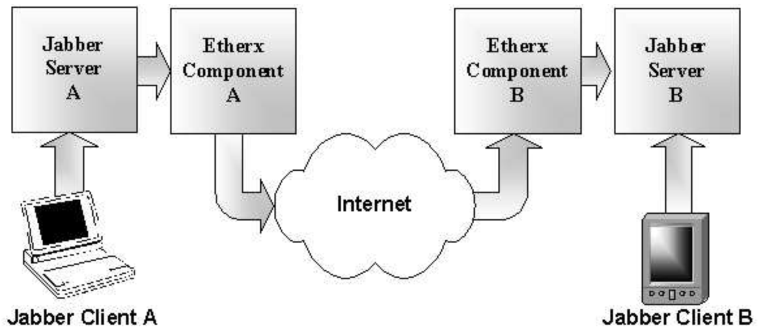


Figure 2.1: Distributed Jabber network

way a decentralized, distributed network of servers can be established.

Transports

A *transport* is a program running on the server that acts as a gateway between the Jabber server and other, non-Jabber instant messaging systems. The transports allow Jabber users to connect to other IM systems to send and receive messages. Translation between the Jabber message protocol and the other IM message protocol must be done in each transport. Although work is being done on standardizing instant messaging message protocols [7, 6], no standard has yet to be embraced by the industry. As a result, message protocols are changed frequently to monopolize the user accounts. This means that the transports must be updated every time a change has occurred.

2.3.4 *JabaServer*

JabaServer (Jaba) is an open-source Java implementation of Jabber, distributed under the BSD licence . It is being developed as a project under SourceForge[2], and is at the time of writing at version 0.6. Jaba uses Apache's Jakarta Avalon[3], which is a framework for server development using Java. The framework contains tools for among other things connection handling, thread pooling and logging.

Jaba is at the time of writing quite basic compared to the original `jabberd` implementation. Transports are for instance not implemented, neither is online user account registration, which must be done directly in the database. However, the handling of core protocol elements `<message/>`, `<iq/>` and `<presence/>` are implemented.

[2]see http://sourceforge.net/projects/jabaserver/
[3]see http://jakarta.apache.org/avalon/

2.3.5 Jabber clients

Jabber client applications must be written in accordance with the Jabber message protocol, but does not necessarily have to implement every feature of it. Clients can be written for any platform capable of parsing XML, and implementations are available for most operating systems. This includes μMessenger for J2ME and JabberCE (alpha version) for Windows CE or PocketPC devices. The clients do not have to be neither open source nor free software, but many clients can be downloaded from [14].

The clients communicate with the users home server over a TCP socket connection, and uses XML for all communication.

2.4 Instant messaging services

Instant messaging services on the Internet have become extremely popular. The awareness information provided by these services enables users to check the status of people in their contact list, check if they are available, busy or offline. IM systems can span the entire Internet such that users all over the world can communicate or they can be used internally in an organization between colleagues. Either way they provide the users with information about other users and a wide variety of ways of communication.

Four services are established as market leaders among the IM services, with user bases in the millions. These IM services were originally developed for desktop PCs, but as wireless Internet access is becoming increasingly common, versions for wireless devices have been released.

Each of these services are presented in the following subsections, with a description of each system's:

- functionality
- availability on wireless devices
- presence

2.4.1 ICQ

The first, and perhaps most well-known, IM service is ICQ [13] – pronounced "I seek you". The first version of ICQ was released late 1996 and instantly became tremendously popular, with 850000 users registering during the first half-year of existence. Since then, the user-base has grown consistently, exceeding 50 million registrations by the end of 1999.

From being a way of exchanging short text messages between computers, the service has been extended to include functionality such as file transfer, SMS messaging, tele-conferencing and multi-user gaming. ICQ was bought by America Online for $400 million in 1998 [29].

Table 2.4: ICQ presence values

Presence	**Icon**
Available	✿
Free For Chat	✿
Away	✿
N/A (Extended Away)	N/A
Occupied (Urgent Messages)	✿
DND (Do Not Disturb)	✿
Privacy (Invisible)	✿
Offline	✿

Wireless ICQ

Beta versions of ICQ are available for both PocketPC and PalmOS. Both versions are stripped down versions of the desktop client.

In ICQ the contact lists are stored locally on each device or PC. When using ICQ from more than one machine, contacts must thus be added to each machine. Although this can be bothersome, it allows a user to keep a smaller contact list on the wireless device.

Presence

The presence values, and the corresponding icons, available in ICQ are listed in table 2.4. The system does not allow the user to define other presence values but ICQ users can write a status message to describe the reason for a presence value when selecting it. This message is not automatically distributed to contacts, but can be read by other users on request. In ICQ there is a one-to-one mapping between a presence value and an icon.

Free for chat is not a presence value per se, but is used to set up *chat room* to which user other users can join. This initiates a different form of messaging, where messages are broadcast to all users currently “in the room”.

2.4.2 AOL Instant Messenger

AOL Instant Messenger [2] (AIM) is America Online Time Warners instant messaging client. In addition to messaging, AIM also provides content such as stock-quotes and news headlines.

With the acquisition of ICQ, AOL has more than 130 million users worldwide [29], a domination that led the American Federal Communications Commission to force AOL to open up its technology to other companies [9].

Wireless AIM

Versions of AIM are available for PalmOS and PocketPC devices. As with ICQ, these are stripped down versions of the desktop application.

AIM also provides an interface for mobile phone users called AIM Wireless [3], however it is available in USA only. Users with operators AT&T, Sprint or Voice-Stream can use AIM by sending SMS messages, using a WAP-enabled phone or using a Nokia 3390. When a user is connected by a mobile phone, other users are made aware of this by displaying a mobile phone icon in the buddy lists. Co-operation with the network providers is required for this to work, and there is no information available on the web-site regarding support for this in countries outside the United States.

Presence

AIM has three presence values: *Available*, *Away* and *Offline*. *Away* can be qualified with an auto-response text message i.e. a message which is automatically sent as a reply to other users when they send a message. The icon in table 2.5 displayed in other users contact lists when the presence is set to *Away*.

No icons are used for the *Available* and *Offline* states. Instead, offline contacts are displayed in a separate group.

Table 2.5: AIM presence values

Presence	**Icon**
Available	N/A
Away	🗐
Offline	N/A

2.4.3 MSN Messenger

MSN Messenger [23] is Microsoft's instant messaging client, and is, according to data released by Media Matrix Inc., the most popular instant messaging service, exceeding 29.5 million unique users primo 2001 [24]. With Microsoft's domination on the personal computer market, the popularity of MSN Messenger should come as no surprise. Further on, the tight integration between Hotmail, .NET Passport and MSN Messenger accounts, serve to consolidate MSN Messengers position in the IM market.

MSN Messenger also serves as a distribution channel for content. Further on, MSN Messenger support SMS messaging, e-mail messaging, tele-conferencing and file-transfer.

Wireless MSN Messenger

MSN Messenger is included in PocketPC 2002, which is Microsoft's latest version of the Windows CE operating system. There are no versions available for neither PocketPC 2000 nor PalmOS.

Presence

The presence values which can be set in MSN Messenger are listed in table 2.6. The user must choose one of these values, i.e. no customized presence value can be set. Neither is it possible to add a presence message – a message explaining the reason for the current presence value.

Some of the presence values share an icon, as is the case with the values *Right back*, *Away* and *Out to lunch*. In each case the icon symbolizes that the user is (presumably) away from the computer, and the text is used to give an indication of the duration of the absence. Similarly, *Busy* and *Telephone* share an icon, where the latter is a special case of the first.

Show as offline makes the user appear as if he were *offline* to other users, but he will continue to receive messages and presence information.

2.4.4 Yahoo! Messenger

Yahoo! Messenger [31] was released June 1999 and is similar to MSN Messenger in that it, in addition to instant messaging, serves as a distribution channel for content: 20 minutes delayed stock-quotes from European and US stock markets can be viewed free of charge. Domestic news, sport results and the weather forecast for the users country is available. Further on, airplane tickets can be booked and the Yahoo! search engine can be accessed.

Yahoo! Messenger has a Personal Information Management (PIM) system as well. Data is stored serve-side, and can be accessed from a web-browser or the Yahoo! Messenger client.

Table 2.6: MSN Messenger presence values

Presence	Icon
Online	👤
Offline	👤
Busy	⊖👤
Right back	🕒👤
Away	🕒👤
Telephone	⊖👤
Out to lunch	🕒👤
Show as offline	👤

Wireless Yahoo! Messenger

Yahoo! Messenger versions are available for both PocketPC and PalmOS. As with ICQ, these versions are stripped down in terms of functionality.

Further on, the customization of the content to be accessed with Yahoo! Messenger must be done from the desktop PC client. The customization can not be done on a per-device basis, which means that the same amount of data is sent regardless of which devices a user is connected with. The lower bandwidth in the wireless networks makes this scheme undesirable, as the user is faced with the option of either overloading the wireless device with data or removing content he would like to receive to the desktop PC. With the payment model of GPRS networks, cost becomes an issue as well.

As in ICQ, the contacts are stored locally.

Presence

Presence values in Yahoo! Messenger are similar to MSN Messenger, but with some additional values. Table 2.7 shows some of the presence values. Yahoo! Messenger allows users to write their own presence values, which are automatically displayed in other users contact list when selected.

Available and *Unavailable* have a unique icon each, whereas the same icon is shared among all the other values. There are two values – (*Busy* and *On the Phone* – indicating that a user is currently by the computer/device, but not interested in receiving messages. The remaining values can be set to indicate absence, with *Be*

Right Back and *On Vacation* at the far ends of the time scale. The user can choose whether to show the online– or away/busy icon for the user-defined presence messages.

Table 2.7: Yahoo! Messenger presence values

Presence	**Icon**
Available	☺
Unavailable	☺
Busy	☺
Not at Home	☺
Not in the Office	☺
On Vacation	☺
Out to Lunch	☺
Invisible	☺

2.5 Wireless devices

Wireless Internet access is now offered by mobile network providers, enabling the use of Internet services such as e-mail and Web browsing on wireless devices, in addition to ordinary voice calls. SMS provides functionality similar to the message exchange in instant messaging services. SMS has proved extremely popular although no awareness information is provided in this service.

For a desktop client connecting through a LAN, processing power and network capacity is more than sufficient. Users connecting with a mobile phone through a wireless network have far less recourses available, including limited bandwidth.

2.5.1 Device categories

Wireless devices have become increasingly popular over the last 5-10 years, and range from pagers and mobile phones to PDAs as powerful as desktop PCs only few years old. The characteristics of the devices vary in terms of screen-size, input mechanisms, memory budget, processing power and operating system. Devices can be categorized into groups of devices with similar characteristics:

Mobile phones Primarily used for voice-calls and SMS messaging, but many phones include additional software such as PIM applications, games and WAP-browsers.

Figure 2.2: The J2ME enabled Siemens SL45i

The most common input-mechanism is a 0-9 keypad, and the screen is small (usually 3–7 lines of text).

Smart phones This category consists of mobile phones with a richer operating system and more applications than regular mobile phones. These devices feature larger screens, often with a touch-screen complementing the keypad for input.

Palm PDAs PDAs running PalmOS, with a rich software environment. These devices feature larger screens, more memory and faster processors than devices in the previous categories. A touch-screen and buttons are used for input, and an external mini-keyboard can be connected as well.

PocketPCs These devices run Microsoft's Windows CE operating system, and are the most powerful in terms of processor speed and memory. Input is handled through touch-screens and buttons. Some devices in this category have built in GSM and/or GPRS mobile phones.

2.5.2 Wireless Java in SIGN

There are obvious advantages of using the same programming language when developing software for different platforms. With cross-platform compatibility as one of the main goals, the Java programming language is a natural choice. However, due to the varying capabilities of the wireless devices, different Java environments must be used. The next two subsections present two such versions which will be used in SIGN.

Figure 2.3: The Handspring Treo 180

J2ME

Java 2 Micro Edition (J2ME) is Sun Microsystems' latest addition to the Java family, and addresses the vast consumer space, which covers the range of extremely tiny commodities such as smart cards or a pager all the way up to the set-top box, an appliance almost as powerful as a computer. J2ME provides building blocks called *profiles* and *configurations*, which target categories of devices with similar capabilities in terms of processing power, memory budget, screen-size and input mechanisms. One such category is called Mobile Information Devices, which consists of devices such as mobile phones and PDAs. The Mobile Information Device Profile (MIDP) is a set of Java APIs which, together with the Connected, Limited Device Configuration (CLDC), provides a complete J2ME application runtime environment targeted at this category.

Up until the introduction of J2ME and MIDP/CLDC, third party software development for mobile phones was not possible. Since then, J2ME has been embraced by all the major mobile phone manufacturers, such as Nokia, Siemens, Motorola and Sony Ericsson. Examples of existing J2ME phones are: Siemens SL45i (see figure 2.2), Motorola Accompli 008 and Nokia 7650.

J2ME applications can also be run on PalmOS PDAs, which are available from a number of different manufacturers. These devices can be connected to the Internet using a separate mobile phone or a device such as the Handspring Treo (figure 2.3), which has a built-in mobile phone module. J2ME will be used for developing software for mobile phones, smart phones and PalmOS devices.

PersonalJava Application Environment

The PersonalJava Application Environment (PJAE) consists of a virtual machine and an API optimized for consumer electronics devices. Network connectable devices for home, office, and mobile use are the targeted applications for the Person-

Figure 2.4: The Siemens SX45 PocketPC

alJava AE. Examples of devices implementing the PersonalJava AE are Internet telephones, personal electronic organizers, and set-top boxes.

The PersonalJava runtime environment is based on JDK 1.1.x, and contains a lot more features than J2ME. This enables more complex programs, but it requires more powerful hardware than J2ME. There is no compiler included in the PJAE, rather it relies on the standard compiler distributed with Java 2 Standard Edition (J2SE).

PersonalJava will be used for developing software for PocketPC devices. The PersonalJava runtime environment for Windows CE 2.11–3.0 can be downloaded from [27], and runs on devices with MIPS or SH3 processors. Examples of devices are: Casio Cassiopeia E-115, HP Journada 540 and Siemens SX45 (see figure 2.4).

2.5.3 Wireless networks

Bandwidth has increased with the transition from the GSM to the GPRS carrier for data traffic in wireless networks. And it will increase even further when UMTS is implemented. However, the transition from a line-switched to a packet-switched network probably has greater impact on popularity of wireless Internet access. Instant messaging for instance is characterized by a irregular transfer of small amounts of data over long time intervals, which makes use expensive in a circuit switched network.

Chapter 3

The SIGN Prototype

3.1 Introduction

In this chapter we present the conceptual model and discuss key areas of the SIGN prototype – awareness and personalization. The conceptual model is shown in figure 3.1 and the following section describes each of the entities in the model. The next section discusses the awareness model used in the SIGN prototype. Finally we present the users possibilities to personalize the system.

Before this discussion we briefly present some key concepts and terminology concerning mobility.

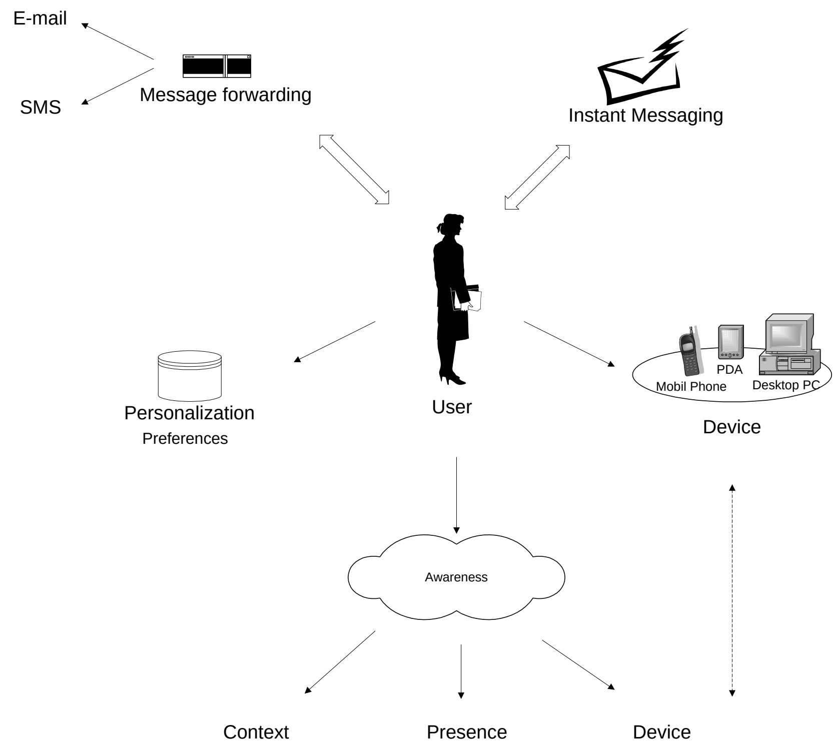


Figure 3.1: SIGN conceptual model

Mobility concepts

In the telecommunications domain, three mobility concepts have been defined, quoted in [28]. These three are personal mobility, service mobility and terminal mobility. *Personal mobility* - concerns possibility to redirect communication across heterogeneous user devices. *Service mobility* - or terminal adaptation concerns the user endpoint and the ability to access services independent of these. *Terminal mobility* - allows users to move from one physical location to another and still having the same set of services available. More specific it is the capability of mobile devices to change its attachment point to the network and retain the same network address and maintain active connections. The wireless networks today provide terminal mobility.

3.2 Conceptual model

3.2.1 User

The user interacts with the system using one or more of the available clients. Independent of device the user sees an interface for instant message and awareness exchange, although clients may differ in functionality dependent on device capabilities. Each user is represented in the system by a unique user name.

In today's telecommunication the caller controls how to reach the callee. In the SIGN prototype the recipient shall be able to decide how incoming messages are handled, thus being in control of how he can be reached. In the telecommunications terminology this refers to pushing control towards the callee or recipient [28]. Based on the users preferences incoming messages are sent to the preferred device or service at the current time.

3.2.2 Instant Messaging

Instant messaging is one of the main functions of the SIGN prototype with focus on enabling the use of wireless devices and handling the users' transition between different devices. Concerning instant messaging the main functions in the prototype are:

Create contact lists A user can create contact lists containing other users. Users added to the list receive a message asking them to approve to be added to the users list. The contact lists can be changed continuously by adding and removing users.

In the SIGN prototype the user can store separate contact lists for each device type. This enable the users to have a long contact list on the desktop PC, while keeping the list on a wireless device short.

Message exchange The user can send and receive messages to and from users in the contact list. A user is notified when new messages have arrived and can choose to read and reply. A history of sent and received messages is also available.

3.2.3 Awareness information

In addition to instant messaging, most IM systems provide awareness of *presence*. With the SIGN prototype users can be connected with a variety of *devices*. In addition, aspects of the user's *context*, or environment, may affect his ability to communicate with others.

The extended awareness model of the SIGN prototype is presented and discussed in detail in section 3.3, but we briefly present the main aspects of the model here.

Presence The system must be able to reflect and communicate the presence of each user. In the simplest sense this means the information concerning if the user is online or not.

Context The context is information that reflects the current situation of a user and properties of the surrounding environment.

Device The device the user is connected with affects his ability to communicate with others. For example, the characteristics of a mobile phone and wireless network restricts the user's ability to receive, read and write messages.

3.2.4 Devices

With the growing number of mobile terminals it is important to deliver services independent of terminal and access network limitations, an aspect described earlier as service mobility. However, differences in device capabilities regarding screen size, input method, processing power etc. makes it evident that functionality and presentation should be dependent on device. In order to meet the user expectations and provide a functional user interface the service must be adapted to the device being used.

Devices supported by the SIGN prototype are desktop PCs, PDAs and mobile phones as shown in figure 3.1. This enables the users to access the service using a wide range of devices and networks. Functionality will be differentiated based on device because of the variation in the device capabilities. The clients on the mobile devices will be adapted to meet the characteristics of the device in order to provide the user with a functional and neat user interface. In the SIGN prototype we will also try to maintain a certain consistency of the user interface across the different platforms. Below is a short description of the devices and functionality provided.

Desktop PC client The desktop client will be similar to clients provided in existing IM system. In addition the desktop client will be the interface to controlling the preferences the user chooses to set. While working at the desktop the user will probably choose to use this client.

PDA client PDAs has in most cases a touch screen and the use of a stylus as input method. It is natural to assume a PDA will be utilized with a wireless network although LAN interfaces are available. Basic IM functionality will be available on the PDA client but it will not be possible to control the preferences.

Mobile phone client The mobile phone is the device with the most cumbersome input mechanism. A user connected with a mobile phone with limited input methods will tend to write shorter messages and use abbreviations compared to a user on a desktop PC. Standard IM functionality will be available on the mobile phone client.

3.2.5 Personalization

In order to meet user demands a service should be highly configurable, especially in a mobile environment were the user can be connected with different devices through different networks.

In the SIGN prototype control is pushed towards the recipient rather than the sender, in contrast to most existing systems. The user can set preferences controlling the following areas:

Opaque or transparent communication This enables the user to choose what awareness information shall be communicated to others. Opaque communication signifies that the information communicated is minimal, while transparent reveals all available information about the user.

How to be reached The user shall be able to specify the preferred way to be reached at the current time. When connected with several devices, for example a desktop client and a mobile phone the user can specify which device incoming messages should be sent to.

Who can make contact The user shall be able decide who shall be allowed to make contact. For example, a user may specify only to accept messages from users on the contact list.

Auto reply messages Users can specify auto reply messages to be sent in specific situations. While busy, incoming messages can be answered with a self-composed automatic reply message.

Forwarding of messages In the SIGN prototype users can set the option to forward messages to other services. This is described further in the next section.

3.2.6 Forwarding

In a service supporting mobility, were users can be reached anywhere using any device, it should be possible to redirect message and information to alternative devices or services. The SIGN prototype supports many device types, nevertheless it can be situations where the user wishes to forward messages to other services. Messages can be forwarded to the following services:

E-mail User may choose to forward messages to their e-mail.

SMS Users may choose to forward messages to a mobile phone using SMS available in the GSM networks.

3.3 Awareness in SIGN

Much research is done on the area of how to provide awareness information and what information is relevant, especially in the area of collaborative work. In [4] awareness is defined as information that is highly relevant to a specific role and situation of a process participant. Further it is argued that awareness information must be digested into a useful form and delivered to exactly the user who needs it. If given to little information users will act in properly or be less effective. With too much information, users must deal with an information overload that adds to their work and masks important information.

Studies conducted in [26, 21] shows that IM often is used to negotiate the availability of others to initiate a conversation or a meeting. By consulting the contact list and the awareness information users can determine whether the recipient are available or not. Providing awareness cues to help people find opportune times to initiate contact is in [30] defined as a key design goal in a workplace communication tool.

In [8] a definition of awareness is given as: "an understanding of the activities of others which provides a context for your own activity". In the case of an IM system, the activities of a user's contact will set a context in which he will decide whether or not to initiate a message exchange with that contact. This will depend on the contact's *willingness* and/or *ability* to write and read messages, which we will call his *communication level*. The communication level must be inferred by the user based on the available information about the contact.

The contact's communication level is dependent on several factors, one of which is his *presence* state. Most existing IM systems provide users with awareness of the presence of their contacts. However, other factors, in addition to the presence state of a contact, affects the communication level. The SIGN prototype supports wireless devices in addition to desktop PC's, and which type of *device* a user is connected with may affect his ability to read and write messages. Further on, the *context* or *location* in which a contact is using the IM system will affect his communication level.

The next section will discuss the awareness information available in the existing IM systems which have been presented earlier. We will then present the awareness model of the SIGN prototype.

3.3.1 Existing systems

The awareness information available in ICQ, AIM, MSN Messenger and Yahoo! Messenger was presented in sections 2.4.1– 2.4.4, and relates solely to awareness of presence. These IM systems were originally developed for desktop computers and the wireless device clients have come along at later stages. However, none of the systems have included awareness information about which type of device a user is currently connected with. Users are thus left with incomplete information about the state of his contacts, which makes it difficult to infer the contact's communication level.

3.3.2 Awareness model

Figure 3.2 show the model for awareness in SIGN. In addition to awareness of presence, awareness of device and context is communicated by the system. The axes are orthogonal to reflect that values along the axes are non-related and that values can be selected independently for each axis. With this extended awareness model, SIGN provides the users better understanding of their contacts' communication level.

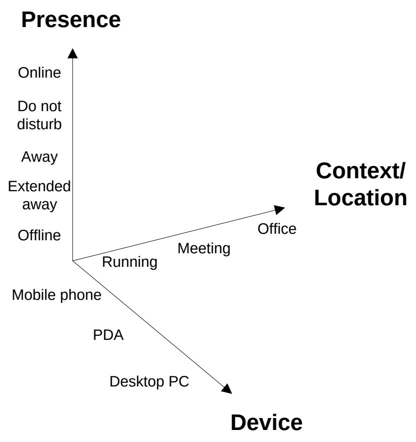


Figure 3.2: Awareness along three axis

The next three subsections will cover awareness of presence, device and context/location as provided in SIGN.

3.3.3 Presence

RFC 2778[7] defines presence in instant messaging systems as something that "allows users to subscribe to each other and be notified of changes in state." The state refers to the users *availability*.

Upon connection to the server, the client application sends an *Online* message to the server, which distributes the message to all subscribers. *Online* is the normal presence state for a user and indicates that he is ready to write and read messages. When the user disconnects from the server, an *Offline* presence message is sent to the server. In between the available and unavailable states there can be several states which reflect the users ability or willingness to communicate. These are:

- Online
- Do not disturb
- Away
- Extended away

The presence state can be set manually by the user or automatically according to some criteria, such as if the connection to the server fails (*Offline*) or if the device is idle for a certain period of time (*Away*).

Depending on which device a user is connected with, different presence values are available. If a user is connected with a mobile phone, he is not likely to be *Extended away* or *Away*, because a mobile phone is usually kept in close range to the user. During phone calls the presence is automatically changed to *Do not disturb*.

On a PDA *Away* should be available, as a user is more likely to leave the device unattended for shorter periods of time.

On a desktop PC *Extended away* is frequently used instead of logging off the system. If the PC is unused for a certain period of time, the presence can automatically be set to *Away*.

3.3.4 Device

SIGN offers awareness of device by keeping users informed of what type of device available contacts are currently connected with. This is important because the characteristic of the various devices will affect a user's communication level. In SIGN users can connect to the service using the following devices:

- Mobile phones
- PDAs or handheld PCs
- Desktop PCs

A discussion of the implications the different device characteristics have on a user's communication level, thus motivating the need for awareness of device, follows in the next subsection.

Input mechanism

The input mechanism varies between devices, and are typically as listed in table 3.1. The input mechanism affects the *speed* at which a message can be written, with a full-size keyboard being several times faster than a 0-9 keypad.

Table 3.1: Device input mechanisms

Device	**Input mechanism**
Mobile phones	0-9 keypad
PDAs	Touch screen or mini-keyboard
Desktop PCs	Full-size keyboard

Message *length* is also affected because a user is not likely to write messages exceeding 200 characters on a mobile phone.[1] Input mechanism affects the *language* used in messages as well. Mobile device users are more likely to use incomplete sentences and non-standard abbreviations, which have become widely used in SMS messages.

Screen size

Screen-sizes vary dramatically, with a mobile phone and a desktop PC at the far ends of the scale, and has implications on the *length* of a message a user can read efficiently and how *fast* it can be read. Further on, the sender may adapt his language according to the screen-size of the receiver, for instance by using "SMS language" and reducing the use of blank lines to partition messages.

Bandwidth

Mobile phones typically communicate over a network with bandwidth between 9.6 kbps and 38.4 kbps, whereas a desktop PC connected to a LAN can have up to 100 Mbps. The bandwidth affects how *fast* a message can be transmitted and indirectly the message *length*.

Further on, the payment models of mobile networks also affect the usage characteristics for users connected with a mobile phone. In packet switched networks the user is charged based on bandwidth usage, and the user may thus not want to receive messages exceeding a certain size. The user may also want to receive messages from fewer users when connected over a wireless network, in order to minimize network traffic.

3.3.5 Location and context

Location and context say something about *where* the user is or in what *surroundings*, respectively. However, context is also used for extending the concept of mobility beyond the sheer geographical meaning.

[1]SMS has maximum 160 characters per message

Information about the context can be collected from different sources. One source of information is what the user enters. In addition there can be other sources of information to capture the user's context. [18] defines the terms *sensor* and *context deductors*. A sensor can for instance detect the number of persons in a room, temperature or motion. A context deducer can examine a user's calendar to find out if a meeting is going on.

SIGN is capable of *distributing* context information, however neither sensors nor deducers are available to obtain this information. Users can, however, enter this information themselves.

Different contexts or locations indirectly tell something about the users ability to communicate. For instance, if a contact's context is set to *driving*, he may be able to *read* a message but not *reply* to it. Examples of other contexts and locations are:

- office
- noisy environment
- car
- meeting
- airport

3.4 Personalization

In this section we present important concepts and functionality introduced in SIGN to support mobility. These concepts can be controlled by the user.

Push control towards recipient

One fundamental property of SIGN is that control is pushed towards the recipient. This is in contrast to how most commercial IM systems work today. Pushing control towards the recipient will help reduce interruptiveness of the IM service, a phenomenon described in [26]. In SIGN the user can control where and how messages shall be delivered and which awareness information is visible to other users.

Control of message delivery

SING supports multiple simultaneous active clients for one user. This means that the user can be connected to the system with multiple devices at the same time.

Providing support for multiple simultaneous active clients will render the service more user friendly. The user is relieved of connecting and disconnecting to the service while moving from one device to another. However this introduces a new problem of how to route messages to users that are connected with several clients. This will in the SIGN prototype be controlled by the user. By assigning priorities to the connected clients the user chooses which one that should receive the message. In addition the user can set up different contact lists for each device and optionally select to not receive message from other than his contacts.

To reduce the amount of data transferred to wireless devices, it is possible to set a maximum size on messages destined for these devices.

In addition, the user can decide that messages should be forwarded to other services, e-mail or SMS.

Control of awareness

Support for transparent or opaque communication is provided by some commercial IM services. However in the SIGN prototype we will provide support for this on all levels of our awareness model. Regarding the awareness model presented in section3.3 the user can decide which of this information should be opaque or transparent independent of each other. This means that the user can hide the device he is connected with, but still communicate the presence and context.

Degrees of personalization

The concept of control can render the service quite different from user to user. A user very restrictive about how he can be reached can virtually block all messages and for example forward these to e-mail. This will of course remove some of the advantages of IM such as quick communication and fast responses. Another user can choose to be easy to reach accepting virtually all incoming attempts for communication.

Chapter 4

System Requirements

4.1 Introduction

In this chapter we present the requirements related to the main functional areas of the SIGN prototype. These requirements are extracted based on the prerequisites for the system and the discussion from the previous chapters. We will divide and present the requirements according to these main functional areas:

- Awareness
- Device differences
- Personalization
- Message delivery

The complete requirements specification is provided in appendix A.

Before presenting the requirements we give a short presentation of the components of the system. Figure 4.1 shows a high level view of the system with possible clients and other services which the system can interact with.

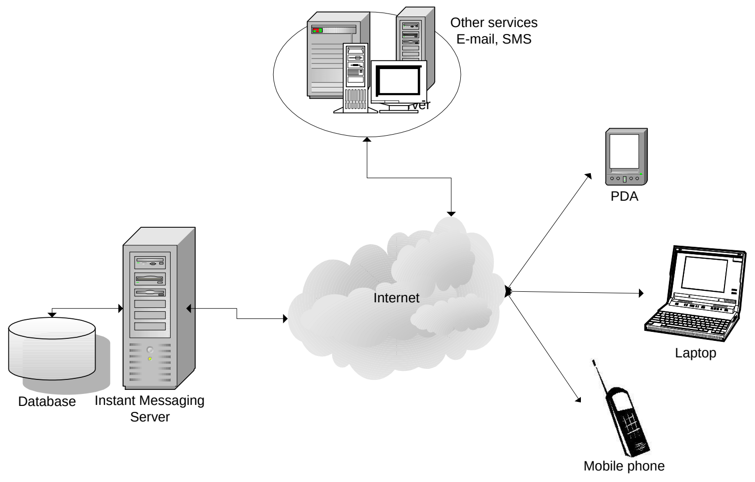


Figure 4.1: High level system overview

4.1.1 Clients

The Instant Messaging system will provide clients for the different devices depicted in the figure and listed below.

- Desktop PCs – J2SE
- PDAs – PersonalJava

- Mobile phones – J2ME

The different devices will connect to the SIGN prototype server through the Internet using standard sockets.

4.1.2 Server

The Jabber Communications Platform, based on XML, will be the core part of the server in the SIGN prototype. Jabber will provide basic services such as network communication, authentication and message routing. Components providing extended functionality such as preference storing and processing, extended awareness handling, interface to other services etc will be developed.

Database

The database will store user information, preferences and offline messages.

4.1.3 Other services

Depicted in figure 4.1 are also the different services the system can interact with. The SIGN prototype will have interfaces to two other services, SMS and e-mail.

4.2 Awareness requirements

The awareness information in the SIGN prototype is comprised of information from three sources, namely device, presence and context/location.

The presence value is set by the user from a predefined list and distributed to other users. The presence behaviour is equal on all the clients except from the values the user can choose from. When a user is connected with a mobile phone or PDA it is unlikely that the user is far away from, not using or inattentive of this device. This is not true for a desktop PC, were the user can have left the office, but still be connected.

The presence values that can be set for the different devices are shown in the table 4.1 below that is taken from requirement FC7 - Set presence, in the requirements specification.

In addition to setting the presence the use can chose to hide the presence value from other users. This functionality relates to the possibility to allow both opaque and transparent communication. If choosing to hide the presence values, the user will be displayed as offline to others (see FC8 - Show/hide presence).

In addition to presence, information regarding device can also be hidden. See requirement FCW2 - Show/hide device.

Context or location information, the last of the three sources for awareness information, must be written by the user and cannot be pre selected from any source

Table 4.1: Presence states

	Desktop PC	**PDA**	**Mobile phone**
Online	X	X	X
Free for chat	X	X	X
Do not disturb	X	X	X
Away	X	X	
Extended away	X		

(see FC11 - Set context/location). This feature is built into the system mainly for expansion reasons.

The awareness information concerning device is sent automatically by the application (see FCW1 - Device type). This is to relieve the user of as much input as possible, making the system easy to use. The information regarding device is sent immediately after logon and is communicated to other users. Another feature concerning awareness is the automatic presence change on mobile phones. The presence information is changed to busy if a user is connected with a mobile phone a engages in a telephone call (see FCW5 - Automatic presence change).

4.3 Device requirements

In the SIGN prototype there are differences in the functionality of the clients. This differentiation is done because of the characteristics of the client devices and the networks they connect through.

4.3.1 Thin clients

The wireless devices have limited resources and the client application that shall run on such devices should be lightweight. This is not entirely true for devices such as PDAs since these devices have powerful processors and more memory. However, limited input capabilities reduces the amount of tasks it is practical to perform on such devices. In the SIGN prototype we have chosen to put most of the functionality other than messaging and awareness to the desktop client. It will serve as an interface to setting most of the preferences in the system and administration of the contact lists. The following requirements are functionality that are only available on the desktop client:

- FCD5 - Add Contact
- FCD6 - Remove Contact

- FCD7 - Edit contact lists

4.3.2 Reduce amount of information exchanged

The wireless devices will usually be connected to the service using networks with limited bandwidth and high latency making it important to reduce the amount of information exchanged. In the SIGN prototype preferences and contact lists can be set for each device (see FS5 - Device dependent contact lists and FS14 - Device dependent preferences).

Contact lists dependent on device will make it possible for users to reduce the amount of contacts when using a wireless device. This will reduce the amount of data downloaded at logon and awareness information regarding other users. Preferences dependent on device makes it possible to limit the amount of information received on a wireless device without reducing the functionality of the desktop client.

4.4 Personalization requirements

Preferences can be set by the user to control various functionality in the SIGN prototype. This enables the user to adapt the service to his needs, control information flow and information amount. All the preferences set are stored on the server.(see FS13 - Preferences stored on the server).

Most of the preferences are controlled from the desktop client. Specifying the maximum size of a message sent to a wireless client and setting forwarding rules for SMS or e-mail are examples (see FCD10 - Set forwarding and FCD9 - Set maximum message length).

An auto reply message can be set for wireless clients. The reason is that user connected with wireless devices might be in situations where it is difficult or impractical to answer messages. Setting an auto reply message will inform the sender that the messages is received and any other information the recipient wants to communicate. An example of an auto reply message can be: ”The message is read, but it will take a while before I can get back to you.” The user can change the auto reply message whenever he pleases (see FCW3 - Set auto reply message).

Users of clients on wireless devices can also choose to forward messages to SMS. This could be the case if a user is connected with a PDA or mobile phone but for some reason must disconnect. The user might want to receive incoming messages and can do that by activating forwarding to the mobile phone (see FCW4 - Set forwarding).

The option to block messages from users not in the contact list can be set for all devices. This can relieve the user of interruptive messages from unknown users (see FC6 - Block messages).

4.5 Message routing requirements

When the server receives a message from a user, a number of different factors influence how and where the message will be delivered to the recipient. These factors are:

- Forwarding rules
- Simultaneous clients
- Filtering rules

The first factor relates to the forwarding rules which can be set by the user (see FCD10/FCW4 - Set forwarding). SIGN allows the user to have messages forwarded by SMS or e-mail regardless of any currently online clients.

The second factor manifests itself because users are allowed to have multiple clients logged on to the system at the same time. However, messages shall not be delivered to more than one client. In order to allow the user to decide which client messages shall be sent to, each client will have a *priority*. The currently logged on client with the highest priority receives messages.

Finally, the filtering rules set by the user affect the message delivery. The filtering can be set up independently for the desktop client and the mobile clients.

The following subsections will describe these factors in detail and give a description of how messages are delivered based on these factors.

4.5.1 Forwarding

SIGN supports forwarding of messages to SMS or e-mail (see FS10 - E-mail message forwarding and FS11 - SMS message forwarding). This allows users to receive messages even if they are not logged on to the system. Forwarding to SMS is particularly useful for user connecting to the system from a mobile phone, as limitations in the J2ME-phones prevent the client application from running during phone calls. Further on, if the user do not have a J2ME mobile phone, a regular mobile phone will at least allow to *receive* message from the SIGN system.

SMS messages cannot be longer than 160 characters. The complete messages will therefore be sent to the desktop client in order to avoid loss of data.

The forwarding rules will be applied regardless of which, if any, clients the user is currently logged on with. This allows the user to be online with a desktop client in the office, but still receive notifications on his mobile phone when new messages arrive.

Figure 4.2 shows how the forwarding rules are applied in SIGN. When a message is received on the server, the forwarding rules for the recipient are retrieved. If no forwarding rules exist, the message is delivered in accordance with the process `Message delivery`, which is described later in this section. If the user has set a forwarding rule, the message is sent using the chosen delivery method.

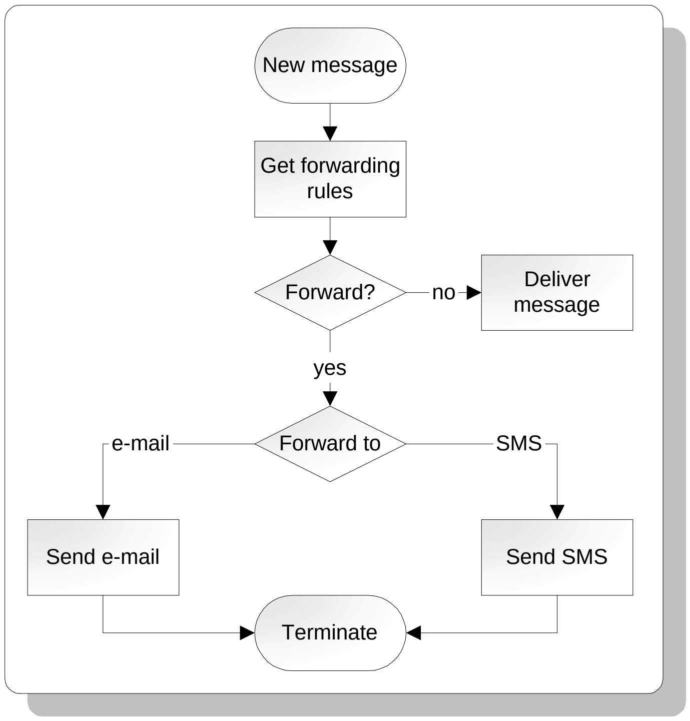


Figure 4.2: 'Forward message' flow chart

4.5.2 Simultaneous clients

One important feature of SIGN is that a user can be logged on with multiple clients at the same time (see FS16 - Multiple active clients). If a user is logged on from his desktop PC, he can log on using a mobile phone during lunch, without having to log off the desktop client. However, when one user has multiple clients online, some mechanism is needed to decide where messages shall be delivered.

Jabber provides this mechanism using the concepts of *resources* and *priorities*.

The resource names will be set automatically for the wireless devices – `'pda'` for PersonalJava applications and `'mobile'` for J2ME applications. This implies that a user cannot be logged on with two mobile phones or two PDA's at the same time, but we consider this to be acceptable. For the desktop application, the user can choose the resource name.

4.5.3 Filtering

The filtering options affect the message delivery in SIGN. Figure 4.3 shows the decomposition of the process `'Deliver message'` from figure 4.2.

Filtering is done one the server (see FS8 - Message filtering), and the user can set it up according to the following criteria:

- Block based on sender
- Maximum message size

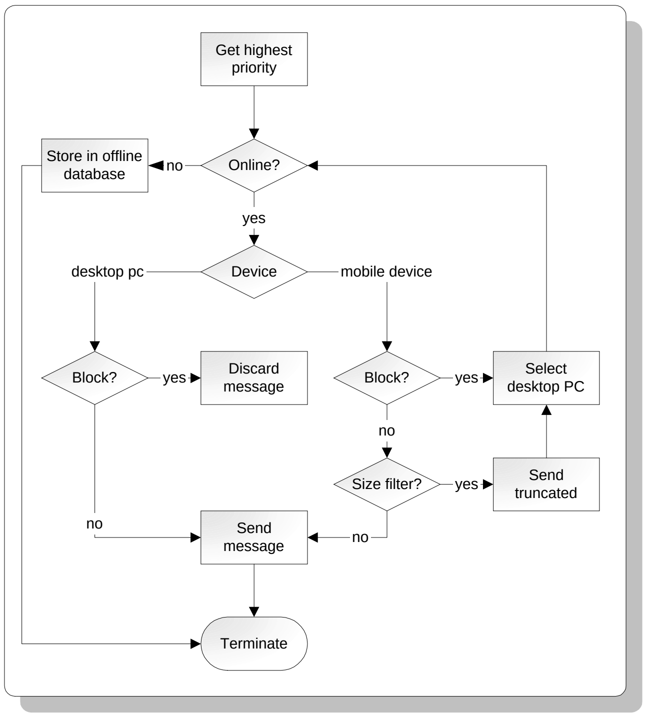


Figure 4.3: 'Deliver message' flow chart

Block based on sender

The user can set the option to block messages (see FC 6-Block messages). If this option is set the user will only receive messages from users on the contact list. This option can be set up independently for the desktop client and the wireless clients. If a message destined for a mobile device is blocked it will be sent to the desktop PC and either delivered or discarded. This filtering mechanism serves two purposes.

For one, access to a packet-switched network, for instance GPRS, is considered to be a condition for using the wireless clients. The payment model in these networks is based on the amount of data transferred rather than connection time. In order to limit the expenses, the user is therefore allowed to put restrictions on the message length. Secondly, it can be used to prevent spam messages.

Maximum message size

The second criteria provides filtering based on the size of a message and can only be set for wireless clients (FCD9 - Set maximum message length). Maximum message length is specified by the user and messages exceeding this length will be truncated before they are sent. The motivation for this filtering option is to allow the user to reduce data traffic. Further on, the wireless devices generally, and mobile phones in particular, have small screens. Reading long messages will therefore be inconvenient on these devices.

In order to prevent information from being lost, the complete messages are sent to the desktop client as well. If the user is not currently logged on with a desktop client, the message is stored on the server and sent to the user on next logon from a desktop client.

Chapter 5

Architectural Choices

5.1 Introduction

One of the prerequisites for this master thesis is to use Jabber to support basic IM functionality. The two different versions of Jabber are listed below.

- *jabberd* - the main implementation of Jabber implemented using C
- Jaba - an ongoing project implementing Jabber in Java

We have studied and tested both of these two implementations to be able to decide upon which one to use in this project. We compared the two versions according to criteria important in this project. These criteria are presented below. Some of them are general criteria that are interesting using any COTS [22] software and some are special for this thesis.

Table 5.1: Selection criteria

Criteria	**Description**
Development language	Knowledge and prior experience
Code duplication	Keeping duplication of existing code to a minimum
Changes in existing code	Keeping changes in existing code minimal
Maturity	Stability of the server software and its compliance to Jabber specification
Work effort	Effort needed to implement changes
NTNU student sessions	Parts of our work shall be used at NTNU in student sessions on architecture and design

At the end of this chapter we discuss possible approaches for extending Jabber with the functionality of the SIGN prototype. We present the implementation we chose, with a motivation for our choice based on the aforementioned criteria. Before presenting the results we start out by presenting the two Jabber versions.

We first introduce the main components of the *jabberd*. This is the version that is in accordance with the documentation and written material about Jabber. Thus this will give insights on how Jabber works and important Jabber concepts. Subsequently we present the Jaba version, the Java implementation of Jabber.

The documentation of the Jabber project is very limited concerning the architecture and design of the different server versions. For *jabberd*, some component diagrams are available. For Jaba, no documentation exists. The documentation presented in this chapter is thus mainly based on our study of the source code.

5.2 *jabberd* server

Jabber is based on a client–server architecture. All data exchanged between clients is routed through the server. Although client-to-client connections are possible, dependent on the client implementation, these are initially negotiated through the server. Jabber has a strong foundation in XML where streams are used for both client - server and server - server communication.

The *jabberd* server is designed to be highly modular where different components handle logically different functionality. Also internally in the *jabberd* server components communicate using XML, thus it is a composition of various XML processing components. All components serialize or deserialize XML for use by other components. Some of the most important tasks handled by the *jabberd* components are listed in below:

- Session management
- Client–server communication
- Server– server communication
- User authentication
- User registration
- Database lookups

5.2.1 Base components

The *jabberd* and the components collectively form a Jabber server. Figure 5.1 shows a high level view of the base components forming the server and the relationship with the *jabberd*. The *jabberd* forms the central hub for all the components. It acts as a central coordinator by routing and delivering XML packets that the different components exchange. The different components handle different types of packets where each packet is in the form of a distinct, fully formed XML fragment. It is identified by the outermost element name in the XML fragment.

Below is a short description of the components depicted in figure 5.1.

Session management

Handles all aspects of a user session on the Jabber server. Examples are IM features such as sending messages, presence, updating the user's contact list and basic session management. The component is called Jabber Session Manager (JSM).

Client to Server connections

This component manages connections between clients and the server.

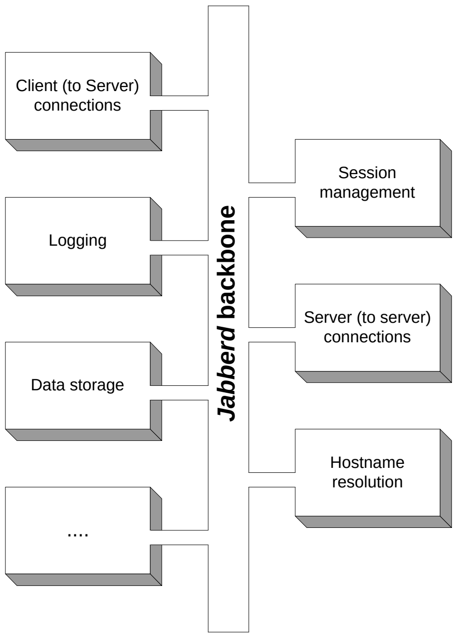


Figure 5.1: *jabberd* and base components

Server to server connections

To manage sending messages from a client on *one* Jabber server to a client on another Jabber server, the servers need a way to communicate with each other. This task is handled by the component called s2s, which establishes and manages server-to-server connections.

Logging

The logging component enables logging of error messages, alerts, notices etc.

Data storage

XDB is an acronym for XML Data Base. It is a component of the Jabber server that provides an interface to any data source used for the storage and retrieval of server-side data.

Hostname resolution

The *dnsrv* component provides a way to resolve names of host that the Jabber server does not recognize as local, as in the server to server context.

5.2.2 Component connection methods

An important concept of *jabberd* is the use of components and the way they can be plugged into the server to add or change functionality. The *jabberd* server's internal use of XML makes it possible to route messages based on XML tags. A component registers what types of tags it wants to receive. Based on the return value of a component's method to handle the message or tag, it is decided if the message shall be routed further to other components or if the processing of this message is complete. An example is to plug in a component that handles login differently than the standard component. This new component could be plugged into the server and register that XML messages of a specific type, e.g. login, shall be routed to this component.Components can connect to the *jabberd* backbone by the methods specified in table 5.2.

Table 5.2: Component connection methods

Method	**Description**
TCP sockets	Components connect through a TCP socket and run as a separate entity. The component can reside on a different server than the one running *jabberd*.
Standard I/O(STDIO)	STDIO method is a mechanism where the *jabberd* process starts the external component itself. The XML documents are exchanged through standard I/O
Shared objects/libraries	Component sources are compiled into shared objects (.so) and loaded into the main *jabberd* process. The component must be written specially with the *jabberd* backbone in mind and contain certain routines.

The method for component plug in is significant for the development of the component. The library load method demands that the component can be compiled into shared object libraries and that specific routines are present in the component. The STDIO method imposes less restrictions on the component, only that it can be started and stopped by *jabberd*.

The TCP socket method imposes virtually no restrictions on the component. Only the exchange of XML messages binds the *jabberd* and the component together. This means that components can be distributed and run on different servers.

The connection method also affects at what detail level the component can be integrated into the server. As the TCP socket method imposes no restrictions on the component it also reduces the control this component has on the message flow in the server. In fact, messages that shall be routed to components connected through TCP sockets must be specifically addressed to this component by the clients. In addition this component can not control the flow of this message further through the server.

Components connected as shared objects can register to receive messages on and XML tag level. These components can also specify if the messages should be processed further by other components or not.

5.3 Jaba server

In this section we present an overview of the design of the Jaba server. The Jaba server design is strongly affected by the main task of the server, namely the processing of XML messages. However the server is not designed to be a generic framework for processing XML elements, but are on the contrary heavily focused towards the Jabber XML message protocol and the processing of the three top-level XML elements `<presence/>`, `<message/>` and `<iq/>`.

5.3.1 Jaba components

Before presenting the design we present the main functional areas for the server. As a starting point we use the package structure of the Jaba server, treating each package as a component or main functional area. Figure 5.2 depicts the different packages where the top-level package is *org.novadeck.jabaserver*. Each package will be referred to as a component. The components are described below and figure 5.3 shows the relations between the most interesting Jaba components.

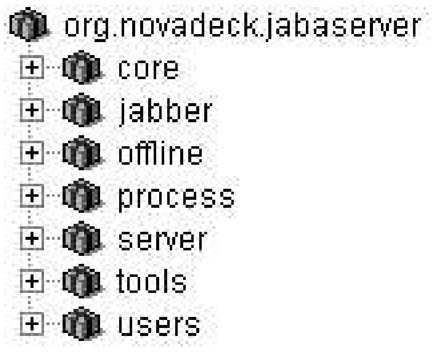


Figure 5.2: Jaba packages

Core

The core component consists of the basic functionality of the server such as starting and initializing the server in addition to client connection handling. The class containing the reference to the XML parser is also contained in this component. The

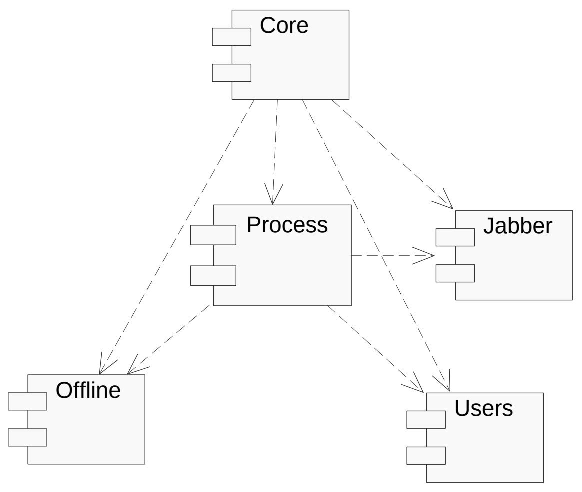


Figure 5.3: Relations between Jaba components

parser used in Jaba is a SAX parser and each new client connection is associated to a parser responsible for parsing the XML from the particular client.

Jabber

The jabber component contains the realization of the Jabber XML message protocol. This component has functionality deserialize XML messages specified in the message protocol. Examples of classes in this component are `Message`, `Iq` and `Presence`.

On this area Jaba differs significantly from the *jabberd* implementation. Whereas *jabberd* uses XML internally in the server where the components interchange XML the Jaba implementation converts the XML messages into Java objects. Thus in Jaba XML is first converted into objects and then the processing of this message, e.g. object, can start. In Jaba objects representing the XML messages are passed to different components or classes for processing.

Offline

The offline component provides functionality for storing and retrieving offline messages.

Process

This component provides the functionality for the processing of an XML message, e.g. message object. This component receives messages that have been deserialized by the Jabber component and performs the tasks associated with the XML message that is received.

Server

This component manages server-to-server connections, enabling the exchange of messages with users connected to other servers.

Tools

This component provides basic tools for TCP socket communication that the server uses for both client-to-server and server-to-server communication.

Users

The server needs to manage user sessions while they are connected to the server. This includes users data management, roster management and resources or active client connections. This component is responsible for storage and retrieval of data from a database or other persistent storage.

5.3.2 Jaba design overview

Figure 5.4 shows an overview of the main classes in the Jaba server. The main elements are the class containing the reference to a XML parser, `Streamparser`, the classes realizing the message protocol, `Element` and its subclasses and the classes managing the client sessions where `User, UserHomeDB and Userstream` are the significant classes. One of the classes responsible for processing XML messages is also shown in the diagram, the class `ProcessIq`. In the following we will discuss the classes in figure 5.4 in more detail, splitting the diagram up in smaller diagrams.

XML message processing

In figure 5.5 main classes for the XML processing is depicted. The classes discussed in this section reside in the components `Core, Jabber, User and Process` described earlier.

The class `StreamParser` contains the reference to the XML SAX parser. For each new client connection an instance of `StreamParser` is created. The XML SAX parser is registered to receive the data from the client socket and `StreamParser` register it self to receive the parsed XML from the parser.

When data is received from the XML parser `StreamParser` checks the type of the message, e.g. `<presence/>`, `<message/>` and `<iq/>`. An object corresponding to the type of message is created, enabling further parsing of the specific message type. In the end, a sub class of `Element` contains all the data from the XML message.

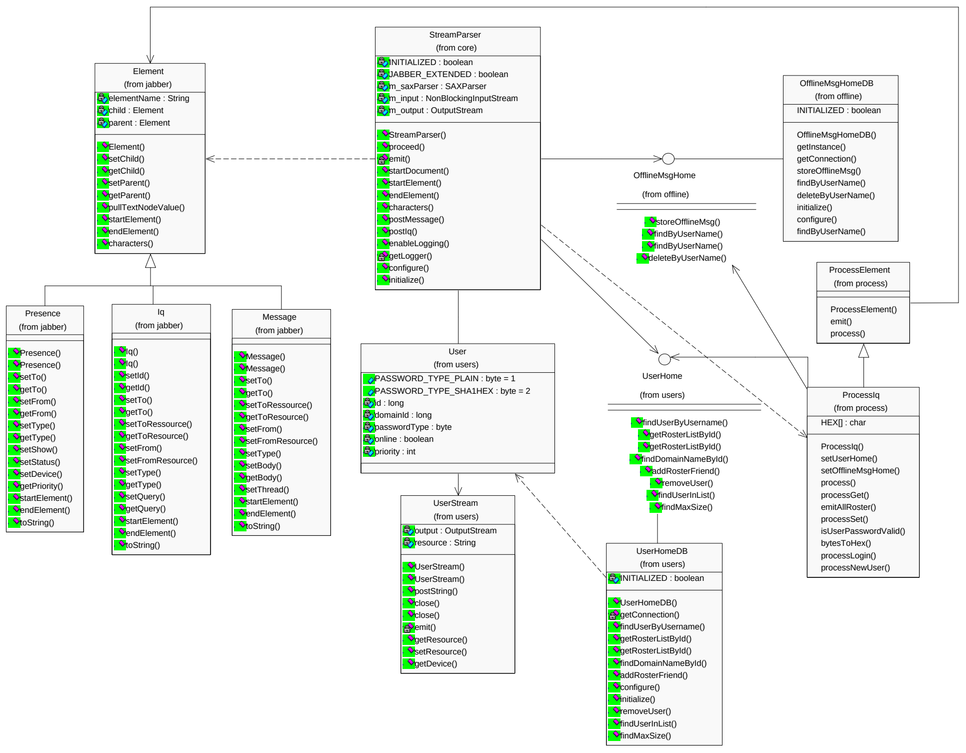


Figure 5.4: Jaba class diagram overview

When a complete XML message is received `StreamParser` either does the actions associated with the specific message type, or it forwards the message object to another class handling the processing. The class `ProcessIq`, depicted in the figure, contains the actions associated with `<iq/>` messages.

A user is realized by the class `User`. A user who is online has an outgoing socket connection and this is realized by the class `UserStream`.

The sequence of actions and method calls on arrival of an `<iq/>` message are described in the sequence diagram in figure 5.6. The generation of the `Iq` object based on the parsed XML is somewhat simplified in the figure since this is done by subsequent calls to the methods `startelement()` and `endElement()` in the `StreamParser` and `Iq` classes. Further on `process()` is called to process the `Iq` message object when it is fully generated. In this example the Iq message triggers an action to send a message to another user. The outgoing message is generated in the `ProcessIq` class and sent through the `User` class and its `UserStream`.

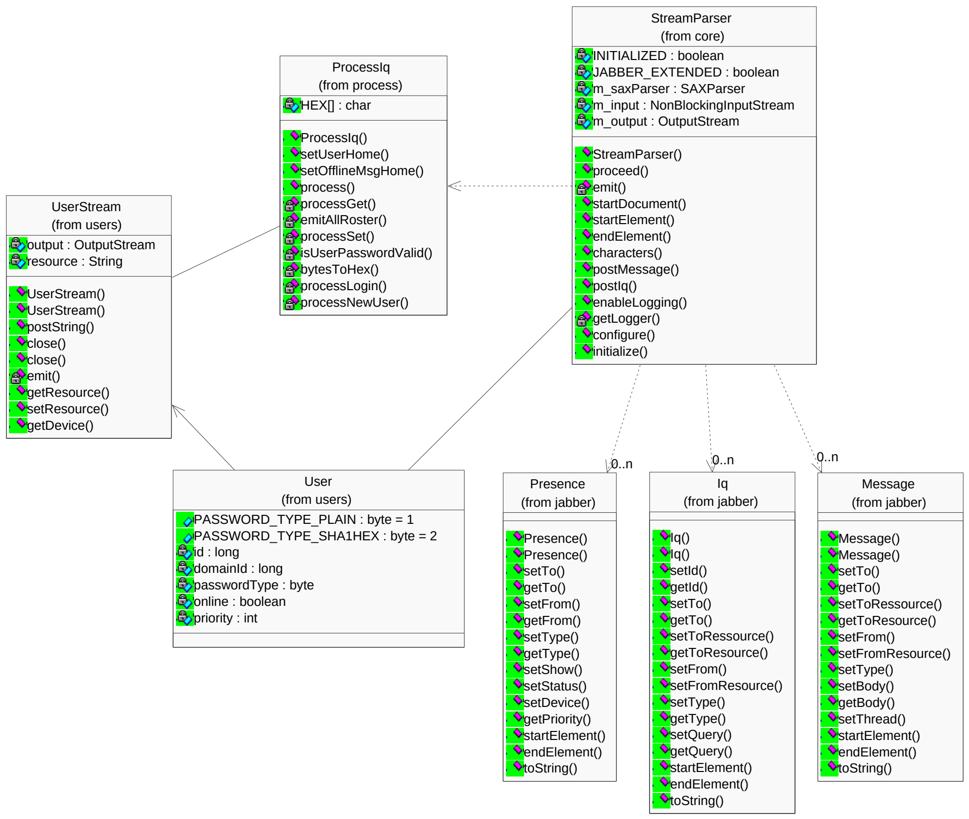


Figure 5.5: Jaba class diagram for XML processing

Message protocol realization

Figure 5.7 shows a detailed view of the realization of the XML message protocol, classes contained in the `Jabber` component.

At the top level of the inheritance hierarchy is the class `Element`. There are eight subclasses of this class where the three top-level elements of the Jabber message protocol are recognized, namely `<presence/>`, `<message/>` and `<iq/>`. The `Iq` class utilizes the class `Query` to generate sub queries which again can be of the types depicted in figure 5.7. In this area Jaba differs from *jabberd*. Instead of keeping the XML and exchange XML internally in the server Jaba creates objects of the incoming XML messages and uses these objects for processing internally. These objects are instances of the classes depicted in the figure.

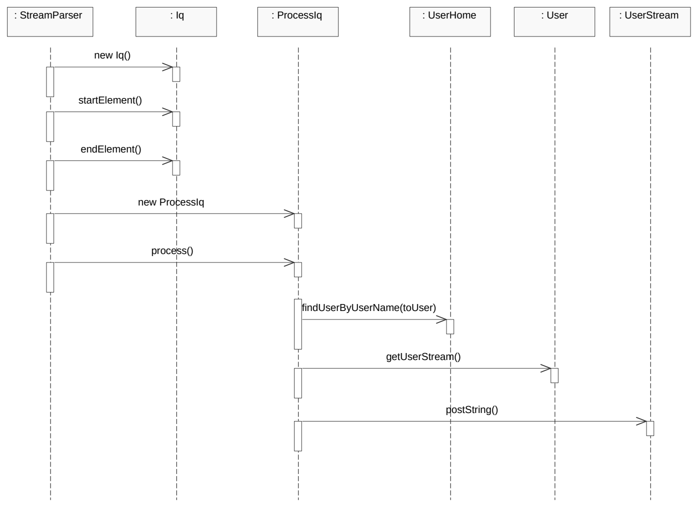


Figure 5.6: Jaba sequence diagram for message processing

Jaba user management

Figure 5.8 shows classes realizing the storage and retrieval of user data. These classes reside in the `User` component.

`UserHomeDB` realizes the interface `UserHome` implementing methods for access to a database. This class contains the connection to a database and the different SQL queries. Other classes access these methods through the interface, such as shown in the figure by the class ProcessIq. The `User` object is instantiated by `UserHomeDB` triggered by a method call from `ProcessIq` when a new logon message, e.g. `Iq` message, is received.

5.4 Architectural choices

In this section we discuss alternative ways of incorporating the new functionality of SIGN into to the Jabber server. We present two approaches, a layered design and the use of internal components.

5.4.1 Layered design

One approach would be to use a layered design (figure 5.9) by introducing a service layer between the Jabber server and the clients, where extensions to the server could be implemented. In this case the clients do not communicate directly with the Jabber server, which means that changes in the message protocol can be implemented if desired. Translation between message protocols would then have to

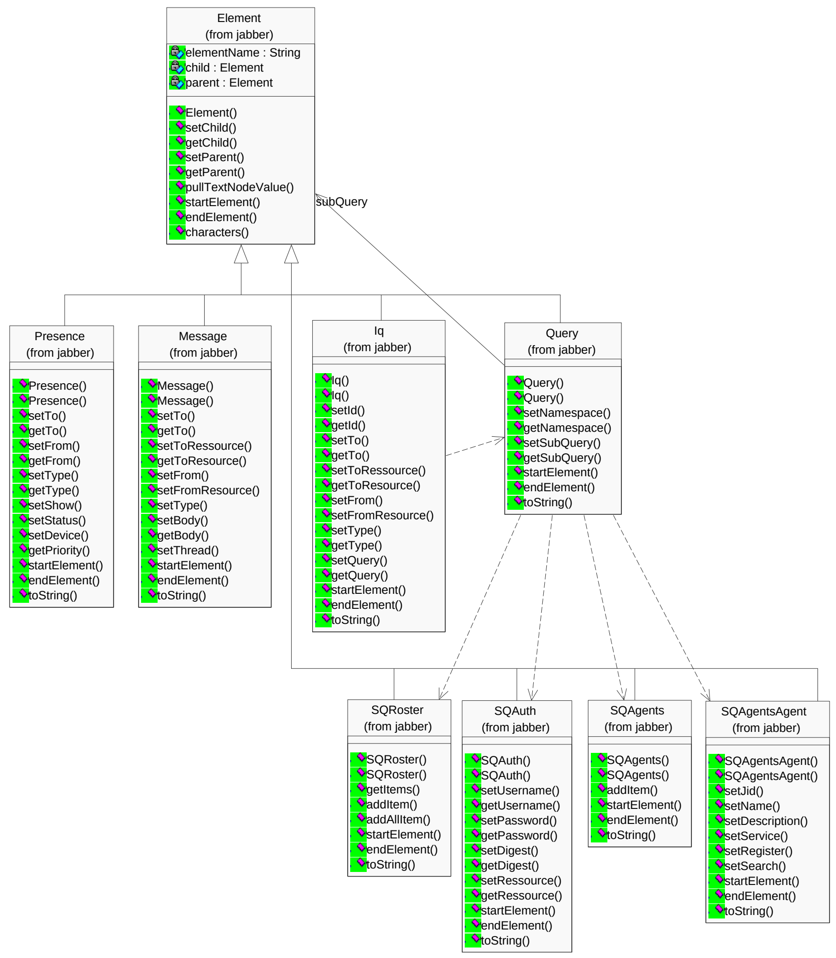


Figure 5.7: Jaba class diagram for XML elements

be implemented in the service layer. Language independence is ensured as the service layer would exchange XML messages with the Jabber server over socket connections.

One of the drawbacks of this approach is that functionality provided by the Jabber server, such as connection handling and XML parsing, would have to be duplicated in the service layer. The work effort to achieve this would be high, and much of the motivation for using existing components would disappear.

As Jabber does not distinguish between devices, the service layer would also in some cases have to “undo” actions performed by Jabber. For instance, if a message should not be delivered to a wireless client due to the message size, the service

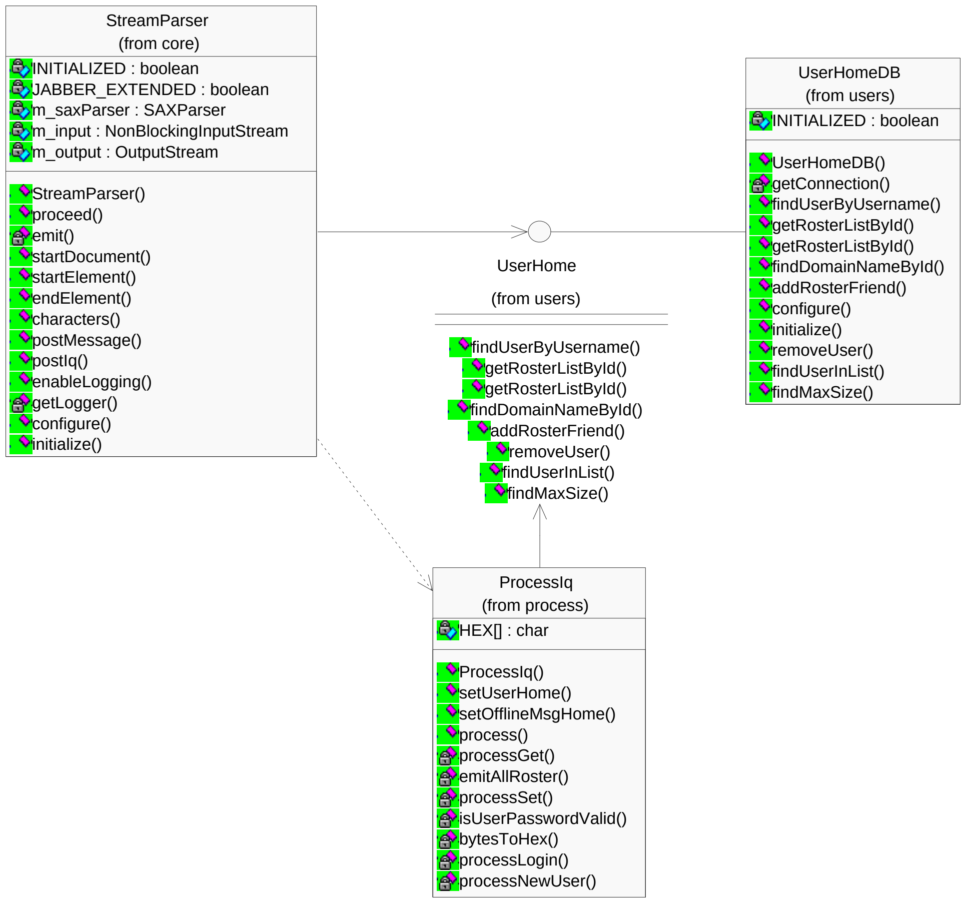


Figure 5.8: Jaba user class diagram

layer would have to intercept the message. Then it would have to send it back to Jabber, addressing the message to the desktop resource or to offline storage.

User information would also have to be stored in the service layer, as the user can have separate contact lists and preferences for different devices.

One advantage of this approach is that no changes to the Jabber server is necessary. The system will thus be easier to maintain as new versions of the Jabber software appear, as only changes to the service layer are (possibly) needed.

Assuring that existing Jabber clients can use the system is easy as well, as they can bypass the service layer entirely.

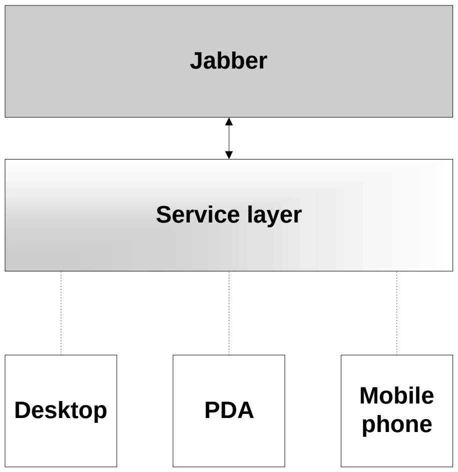


Figure 5.9: Layered server

5.4.2 Internal components

Using internal components (see figure 5.10) is another way of implementing the changes to the Jabber server. In this case the clients communicate directly with the Jabber server, same as the normal use of Jabber.

Internal communication between the new components and the server is done by method calls, which means that programming language independence is not achieved. The components must in some way be inserted into the message processing "loop".

In *jabberd*, components can be loaded at runtime based on a startup script. When loaded, components register callback functions for various events, such as the arrival of messages or presence information. The further internal routing of data is based on the result of each component in the loop.

Jaba on the other hand does not explicitly consist of components. However, as mentioned in the previous section some groups of classes can be regarded as logical components. New components can not be added at runtime as in *jabberd*, rather they must be added at compile-time. In addition, it is not possible to set callback-functions. As a consequence, changes in the source code is necessary to implement the extensions to the Jaba server.

Using components will give a higher degree of control than the layered approach. Components that are added can shortcut the use of standard components replacing functionality. On the other hand components can provide additional functionality by including the component in the message processing sequence without replacing others. In this way functionality provided by Jabber could be reused and minimal duplication of functionality would be necessary.

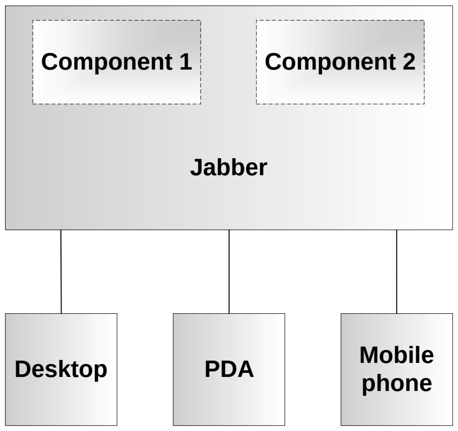


Figure 5.10: Internal server components

5.5 Jaba vs. *jabberd*

After an initial study of the two available server versions, our preferred choice of server was *jabberd*. But after the detailed study, in the end we chose to use the Jaba. In this section we present the pros and cons for each implementation starting with the one with did not choose, *jabberd* We base our discussion around the selection criteria we presented in the introduction of this chapter.

5.5.1 *jabberd*

One of the main reasons for initially wanting to user *jabberd* is that it complies to the documentation and written material about Jabber. It implements the message protocol completely and it is currently in use on the many Jabber server available on the Internet today.

In addition it realizes the important concepts of Jabber, namely the strong foundation on XML. *jabberd* uses XML also for internal communication between components acting as a framework for routing XML. Changes in the message protocol can be implemented with little effort adding a new component handling new XML elements or tags.

However, when studying *jabberd* we found that it would be difficult to implement many of the changes needed for the SIGN prototype because of the detailed level at which changes needed to be made. Some of the new functionality demanded either changes in current components or extensive duplication of functionality into new components. We concluded that implementing the new functionality would require extensive reworking of existing *jabberd* source code.

jabberd is implemented using C, a language in which we have little experience. Implementing the extended functionality of the SIGN would demand a greater effort than the case in a programming language in which we have experience. Adding components however could be done using STD I/O or TCP sockets enabling the use of other programming languages, but again reducing the detail level of how we could change *jabberd*.

Java is the main programming language taught at NTNU and since this project should be used in sessions with students using at least an object oriented language would be preferable.

5.5.2 Jaba

Our choice to use Jaba was very much based on Java being our preferred programming language and the fact that using *jabberd* would still require extensive duplication, or reworking, of the existing source code.

Java is a programming language we are familiar with and have experience from in other development projects. This should reduce the work effort compared to C. In addition using Java would make it more interesting to use the project in student sessions at NTNU, one of the prerequisites of this project. Further, it will probably make the work done in this project more accessible to other students wanting to continue on the work we have started.

However, there are several aspect about Jaba that makes it far from ideal to use. Jaba does not comply with the main concept of Jabber, namely a generic framework for routing XML. The server is highly focused towards the Jabber message protocol. In addition the server does not implement the message protocol fully, lacking a great deal of the functionality provided by *jabberd*. Thus, using Jaba makes it necessary to make changes in the source code.

Jaba is an ongoing project very much in it's infancy. This accounts for the lack of functionality and not complying to Jabber specification, but it also implies that many updates of the server software will be released before it could be called a complete Jabber server.

Further specification of how we utilize Jaba and make changes to implement SIGN, is given in the next chapter.

Chapter 6

Design

6.1 Introduction

In the previous chapter we presented the decision to use Jaba as the Jabber server in the SIGN prototype. In this chapter the focus will be on *how* we will use and extend Jaba to incorporate the SIGN requirements from chapter 4. An obvious goal is to design the system such that the existing Jaba functionality can be used and at the same time isolating our changes.

The chapter starts off by presenting the SIGN system architecture. When the decision to use Jaba was made, restrictions on the system architecture were imposed, as Jaba is less flexible than *jabberd* in terms of how the server interacts with other components and services.

Then we present the design for the extended Jaba server. In section 5.3 the various Jaba components were introduced and this section will focus on changes made to these components and the interaction between them.

The final section of this chapter presents Jabber related parts of the client design. More specifically, we will show the parts of the client design which are related to *parsing* and *processing* of the Jabber message protocol elements. This is shared among the three versions of the client application.

6.2 SIGN system architecture

Figure 6.1 shows the SIGN system architecture, a three-tiered architecture with a client–, server– and data-tier. Many of the server requirements from chapter 4, particularly the basic IM requirements, are already fulfilled by the Jaba server. The main decision concerning the system architecture thus relates to where the remaining server requirements shall be implemented. As the figure shows, these requirements are implemented in the server-tier, *inside* Jaba. From this follows that the SIGN requirements not necessarily lead to changes to the system architecture as a whole. A short description of the three tiers concludes this section.

Client-tier

The client-tier is comprised of the three versions of the client application. Each client exchanges XML-messages with the server-tier over TCP/IP socket connections. Instances of the client application will run on PC's, PocketPC's and J2ME enabled devices such as mobile phones and PalmOS PDAs.

In addition to the SIGN client applications, existing Jabber clients can connect to the server.

Server-tier

The server-tier consists of the Jaba server, modified to fulfill the SIGN requirements which Jaba do not. As shown in figure 6.1, we have chosen to implement the extended functionality internally in the Jaba server. The changes made to the

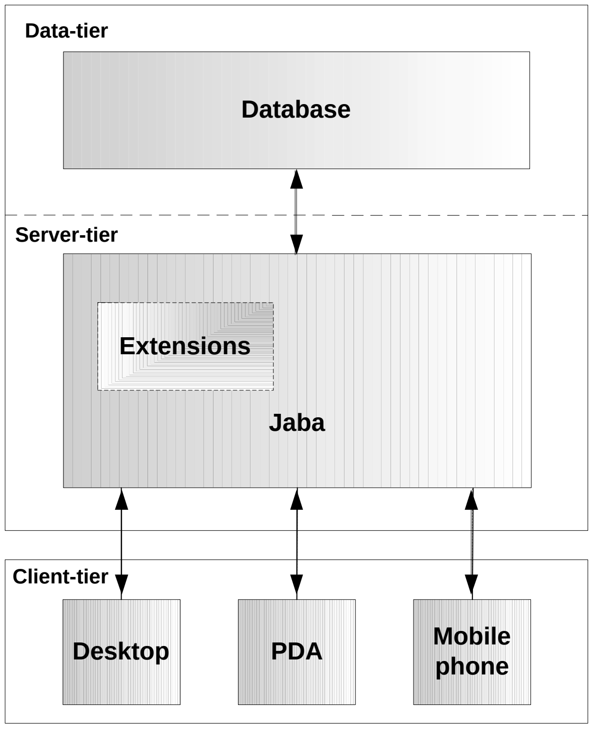


Figure 6.1: SIGN high-level system architecture

server do not affect the use of existing Jabber clients as the new functionality must explicitly be addressed by the client in the content of the `<iq/>` and `<presence/>` messages.

Data-tier

The data-tier is responsible for storage of persistent data. In SIGN additional data is stored on the server compared to Jaba. For instance contact lists for the different devices and user preferences.

How the data is stored is up to the data-tier, but in Jaba a database is used and we see no reason to change this. The data-tier provides an interface to the server for storage and retrieval of data, which will be presented in the next section.

6.3 Server design

The Jaba architecture was presented in section 5.3. In this section we will give an account of how we intend to extend Jaba into a SIGN server.

One of the main goals for this process is to modify *existing* functionality and incorporate *new* functionality, but at the same time minimize the changes to Jaba.

We have thus opted to leave the architecture fundamentally unchanged, as we believe that this will facilitate the process of converting future releases of Jaba into a SIGN server. When modifications are necessary, we will seek to isolate this from the existing classes.

Jaba already provides most of the basic IM functionality in SIGN, so the necessary changes by and large consist of adding functionality related to the following properties of SIGN:

- Extended awareness model
- Use of different devices
- User preferences

Awareness model

To implement the SIGN awareness model from section 3.3, device and context information, in addition to the users presence, must be sent from the client to the server and distributed by the server to subscribing clients. This information will be communicated using the existing Jabber `<presence/>` message, which implies that the parsing and processing of these messages must be changed on the server.

Different devices

A fundamental requirement to SIGN is that it shall allow users to connect with different client devices. The service can be customized by the user depending on which type of device he is using, for instance by using a smaller contact list when connected from a mobile phone compared to the contact list on a desktop PC. This implies that SIGN has different requirements regarding storage of persistent data.

Further on, contact lists and preferences must be set up on the client applications and sent to the server. For this purpose the ”workhorse” of the Jabber message protocol – the `<iq/>` message – will be used. The parsing and processing of these messages must thus be augmented.

User preferences

Section 4.5 explains how a user can set various preferences determining how and where messages shall be delivered according to which device(s) he is connected with. The SIGN server must thus obey different rules for delivering messages than Jaba.

In addition, SIGN shall allow the user to have messages forwarded to either SMS or e-mail. This functionality does not exist in Jaba, and we must thus find a proper place to implement it.

Design changes

Rather than changing the *architecture*, we intend to implement the new functionality by modifying the existing *design*.

Section 5.3.1 described the various components in Jaba, and of these the following components will be modified:

- `Jabber`
- `Process`
- `Users`

In addition to these changes, a new component called `Services`, used for sending SMS and e-mail messages will be added.

The following sections describe the changes to each of these components as they appear in SIGN.

6.3.1 `Jabber` component

The `Jabber` component is responsible for the process of converting XML messages into object representations, which is the unit of exchange between the Jaba components. Figure 6.2 shows the class diagram for this component in SIGN.

From the figure it can be seen that the `Message` and `Presence` classes do not rely on other classes for converting their corresponding message, which is the same as in Jaba. `Iq` on the other hand uses various classes when parsing messages. This is due to the fact that `<iq/>` messages can contain various data depending on the namespace set for the `<query/>` sub-element. In SIGN two new classes are used for this purpose in addition to those from Jaba.

Message

No new attributes or sub-elements to `<message/>` are introduced in SIGN, so the `Message` class is consequently left unchanged.

Presence

Due to the extended awareness model in SIGN, `<presence/>` messages from mobile phone and PDA client applications may contain a `<device/>` sub-element. Jaba discards tags which are not explicitly recognized (in contradiction to *jabberd*), therefore the `Presence` class must be modified to read this tag from the presence messages.

The same applies to the `<priority/>` sub-element in the presence messages, which are discarded by Jaba as well. This is not in accordance with the Jabber message protocol [20], and must be implemented as SIGN relies on the use of priorities.

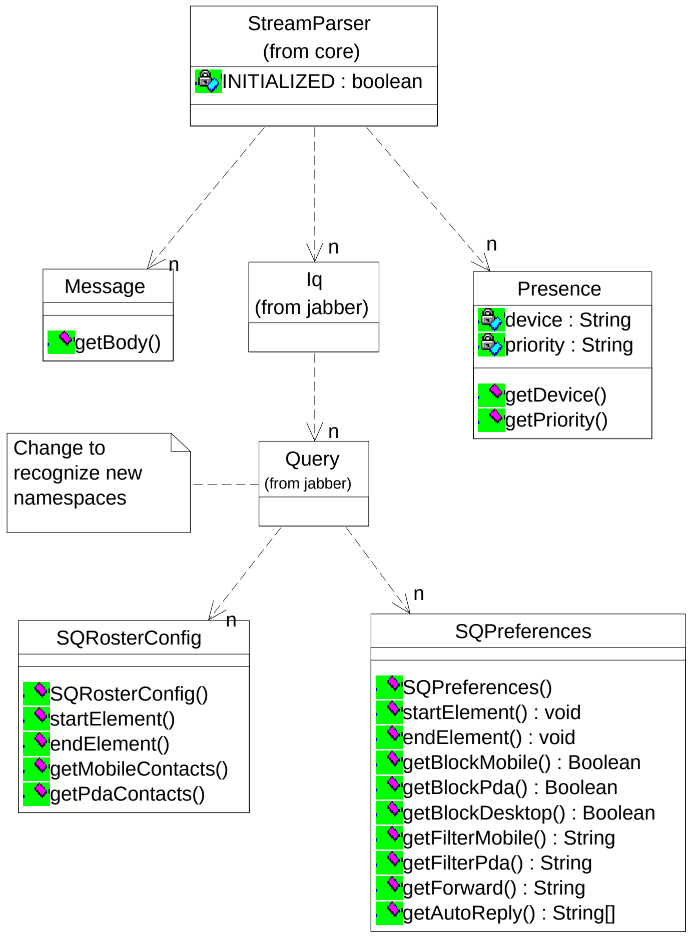


Figure 6.2: Class diagram of `Jabber` in SIGN

In figure 6.2 it can be seen that methods to retrieve these values have been added to the `Presence` class.

Iq

User preferences and device dependent contact lists are in SIGN sent from clients to the server contained in `<iq/>` messages. The Jabber message protocol utilizes namespaces to distinguish between the various uses of the `<iq/>` message, so two new namespaces are introduced in SIGN for user preferences and device dependent contact list. These are `jabber:ext:iq:preferences` and `jabber:ext:-iq:rosterconfig`, respectively.

The top-level `<iq/>` element itself is not changed, as the namespaces are set for the `<query/>` sub elements of the messages. Figure 6.2 shows that the `Iq` class is consequently left unchanged.

The `Query` class must, however, be aware of the two new namespaces above. In order to create object representations of messages under these namespaces, two new classes are introduced as shown in the figure:

SQPreferences This class contains methods to get the state of the different preferences which can be set up by the user.

SQRosterConfig This class contains methods to get the contact list for PDAs and mobile phones.

Instances of these classes are created when the respective namespaces are detected in the `<query>` element. In figure 6.2 it can be seen that the instances of `SQPreferences` and `SQRosterConfig` are created by the `Query` class.

6.3.2 `Process` component

In Jaba the `Process` component is used for processing `Iq` objects generated by the `Jabber` component. Processing of `Message` and `Presence` objects however, is done by the `Core` component. We believe that it is better to leave this responsibility to the `Process` component for the two latter message types as well, in order to maintain a clear distinction between the functionality of the different components. This is particularly important in SIGN, as the message delivery rules and the awareness information in `<presence/>` messages are more complex than in Jaba.

Figure 6.3 show the class diagram of the `Process` component in SIGN. All processing classes extend the class `ProcessElement`. The two classes at the top of the diagram are the existing Jaba classes, whereas the classes at the bottom – `ProcessIqExt`, `ProcessMessageExt` and `ProcessPresenceExt` are classes specific for SIGN.

The following subsections present each of the latter classes.

`ProcessIqExt`

As we stated in the previous section, two new namespaces are used in `<iq/>` messages. The `Process` component must thus be changed to handle these namespaces. Instead of changing or replacing the existing class used for processing `Iq` objects (`ProcessIq`), we introduce the class `ProcessIqExt` which will handle `<iq/>` messages in which one of the new namespaces is set. The two classes will be used in parallel, thereby evading the need to duplicate functionality existent in the first in the latter.

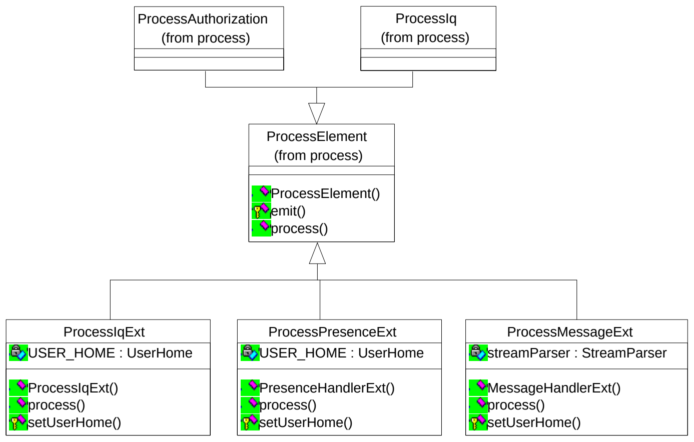


Figure 6.3: Class diagram of `Process` in SIGN

ProcessMessageExt

Message delivery is more complex in SIGN than Jaba, as the user can set up various rules as to where and how messages shall be delivered. Therefore we decided that the `Process` components shall be responsible for this in SIGN. The class `ProcessMessageExt` seen in figure 6.3 is used for this purpose and will be responsible for implementing message delivery according to the scheme which was described in chapter 4.

This class will replace the message processing which in Jaba is done by the `Core` component. Although this violates the goal of avoiding duplication of functionality, the existing message processing would have to modified anyway. We belive that the advantage of separating the changes in new classes in this case outweighs the disadvantages associated with duplication of functionality.

ProcessPresenceExt

The extended awareness in SIGN requires `<presence/>` messages to be handled differently than in Jaba.

Analogous to message processing, the responsibility of processing `Presence` objects is moved from the `Core` to the `Process` component by adding the class `ProcessPresenceExt`.

The same policy is applied as with message processing, where the new class will replace the existing functionality for this in the `Core` component.

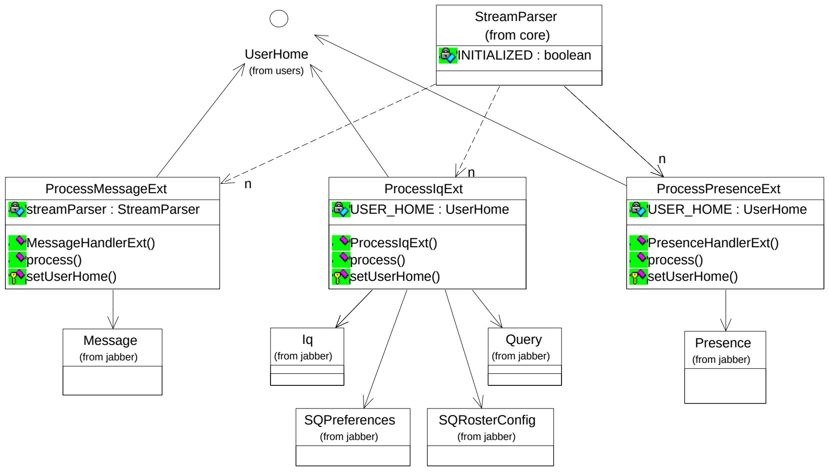


Figure 6.4: Interaction between `Process` and other components

Interaction with other components

Figure 6.4 show the relationship between classes from `Process` and classes from other components. Most classes from the `Jabber` component are used, with each processing class working with different classes from `Jabber`. `ProcessPresenceExt` and `ProcessMessageExt` uses the `Presence` and `Message` classes, respectively. `ProcessIqExt` uses the various classes representing different parts of `<iq/>` messages.

`StreamParser` from `Core` creates instances of the different processing classes for each message received.

The `User` component is used to store and retrieve various user data such as, offline messages, user accounts, user preferences and contact lists.

6.3.3 `Users` component

The `Users` component is among other things responsible for storage of persistent data, and is thus the interface to the data-tier. One of the requirements to SIGN is that users shall be able to set up different contact lists for the various devices. This implies that the `Users` component must be changed. This is also the case for the preferences that users can set up regarding where and how messages shall be delivered.

Other components access the persistent data through the interface `UserHome`. In Jaba, the implementation of this interface is done by the class `UserHomeDB`.

We have decided not to introduce any new classes in the `Users` component, instead including the necessary functionality by augmenting the `UserHome` interface with the additional methods needed by other components in SIGN. In this way the all access to the data-tier is done trough one interface.

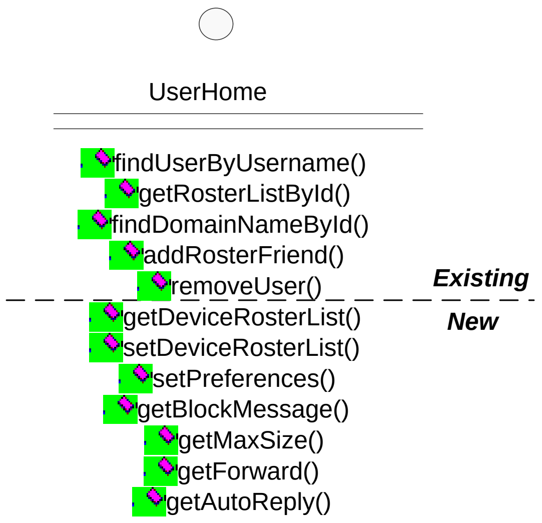


Figure 6.5: `UserHome` interface

Figure 6.5 shows the extended `UserHome` interface for SIGN. The methods above the dotted line are the existing Jaba methods, whereas those below the line are added for use in SIGN. Table 6.1 summarizes the purpose for each of the new methods.

6.3.4 `Services` component

SIGN allows users to have messages forwarded to SMS or e-mail. In contradiction to *jabberd*, Jaba does not provide a means to communicate with other processes or services, so the functionality to send e-mail and SMS must be included in the modified Jaba server.

We have designed a component called `Services` for this purpose, shown in figure 6.6. The idea is that various services can be added to this component. The services shall be loaded when the server is started according to configuration files.

Other components access the `Service` component through an interface called `ServicesHome`, which is similar to how the data-tier in Jaba is accessed. This interface provides methods to add and remove services and methods to give other components access to the existing services.

Classes are provided for each specific service: (`SMSService` and `MailService` in figure 6.6). All service classes must extends the abstract base class `Service`, and must hence implement the methods `sendMessage()` and `getConstraints()`. The first method is self-explanatory and the latter is used to get

Table 6.1: Additional `UserHome` methods

Method	Description
`getDeviceRosterList()`	This methods returns the roster list for the PDA or mobile phone client application. The existing method `getRosterListById()` will be used to get the roster list for other client applications.
`setDeviceRosterList()`	This method is used to set the roster list for the PDA and mobile phone client application.
`setPreferences()`	This method is used to store the various preferences which can be set up by the user.
`getBlockMessage`	This method is used to check if a message should be delivered to a user. This will depend on the device the user is connected with and the sender.
`getMaxSize()`	This method is used to check if the user has set a maximum size for messages destined for a PDA or mobile phone.
`getForward()`	This method is used to find out if the user has set the option to forward a message to either e-mail or SMS.
`getAutoReply()`	This method is used to find out if the user has set an auto-reply message, which will automatically be sent as a reply to incoming messages.

potential constraints, for instance maximum message length for the SMS service which is 160 characters.

6.3.5 Component dependency

Figure 6.7 shows the dependency between the main components in SIGN. From the figure it can be seen that the `Core` component is not directly dependent on the `Offline` and `Users` components as it was in Jaba (see figure 5.3). This is a result of moving processing of `<message/>` and `<iq/>` messages from `Core` to `Process`.

The `Service` component is used solely by the `Process` component.

Sequence diagram

Figure 6.8 show a sequence diagram which gives an example of how the components interact. The sequence diagram represents a case where a message shall be

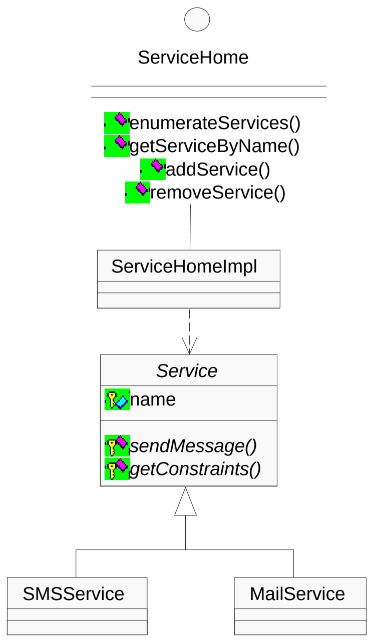


Figure 6.6: Class diagram of `Service` in SIGN

delivered to a client connected with a mobile phone.

When `StreamParser` detects a `<message/>` element it creates a new instance of `Message`. Through subsequent calls to `startElement()` and `characters()`, the message object encapsulate the XML message. `StreamParser` then creates an instance of `ProcessMessageExt` and invokes the method `process()`.

In `process()` the `User` object for the recipient is retrieved from `UserHome` and the device type, which in this case is a mobile phone, is retrieved. `getBlockMessage()` is invoked to determine if the user has set any preferences which would cause the message to be blocked. In this diagram this is not the case and `getMaxSize()` is invoked to get the maximum allowed size of messages destined for a mobile phone. In this case the user has set a limit which is less than the messages size and the message is consequently truncated.

The `UserStream` for the mobile phone is retrieved and the message sent to user.

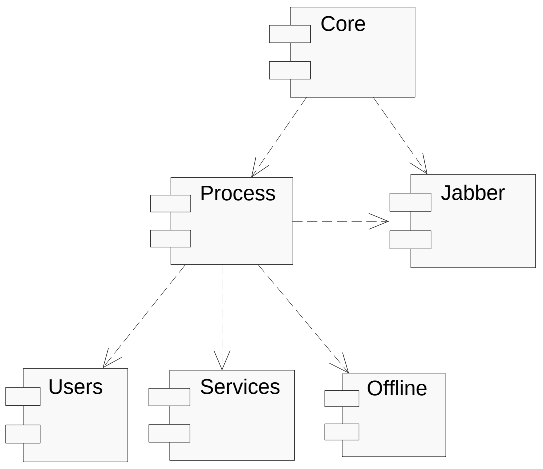


Figure 6.7: Dependency between SIGN components

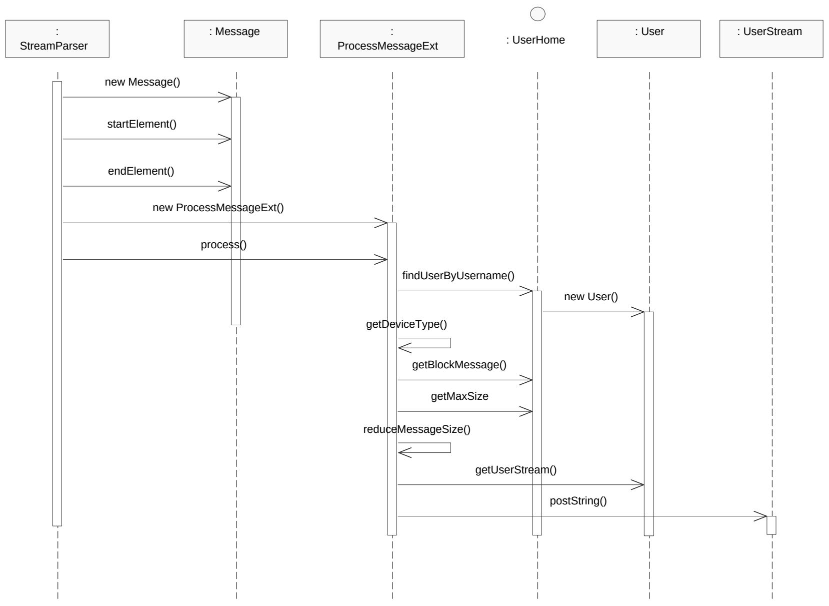


Figure 6.8: Sequence diagram of message delivery

6.4 Client design

Jabber's client–server model is heavily weighted to favour the creation of simple clients. Most of the processing and IM logic is carried out on the server and the responsibilities for the client are minimal. To create a Jabber IM client the following main tasks must be addressed:

- Managing the connection to a Jabber server
- Implementing the Jabber XML message protocol
- Providing the IM user interface

Providing means to connect to a Jabber server and handle the XML message protocol are fundamentals that must be in place for every Jabber client.

In the SIGN prototype we have developed three different clients, namely desktop client, PDA client and mobile phone client. In this section we present the central parts of the client design. We will not distinguish between the three clients since the central parts presented here apply to them all.

6.4.1 Design overview

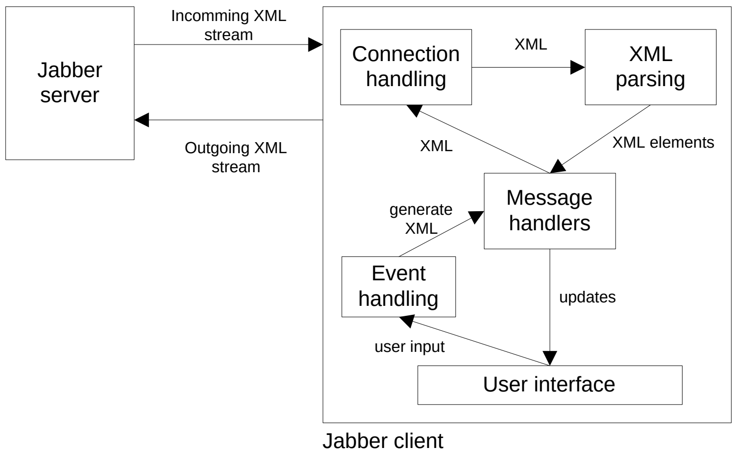


Figure 6.9: Client design overview

Figure 6.9 depicts an overview of the main elements of the client design.

The clients must create and manage a TCP socket connection to the Jabber server. XML is exchanged with the server over this socket. An XML parser parses incoming XML messages. Message handlers then handle parsed XML, there are one handler for each top-level XML element, e.g. `<message/>`, `<presence/>` and `<iq/>`. These handlers contain the functionality associated to a particular type of message. Usually the arrival of a message will trigger an update of the user interface.

The user interacts with the application through the user interface, causing different events. Events can either be handled by the user interface or sent to an event handler. If an event triggers the sending of an XML message, the message handlers generate the XML.

6.4.2 Message exchange and processing

The client design is the similar for all clients, except for the user interface. The different devices and programming languages used require different realization of the user interface. There are also some differences in the implementations because of the difference in the functionality on the clients. For example, the desktop client contains more functionality than the PDA and mobile phone clients.

Figure 6.10 present the class diagram realizing the communication, XML parsing and message processing on the clients.

The class `JabberComm` receives the data read from the TCP socket and parses the XML message. The XML parser is realized by the class `XMLElement`. A complete XML message is parsed before it is sent to processing. This is fundamentally different to the XML parsing on the server where data is sent for processing as soon as only a complete tag is received and not a complete message. Thus on the clients an XML tree, e.g. a tree of `XMLElement` objects, containing all attributes and tags of complete message is generated before further processing is done.

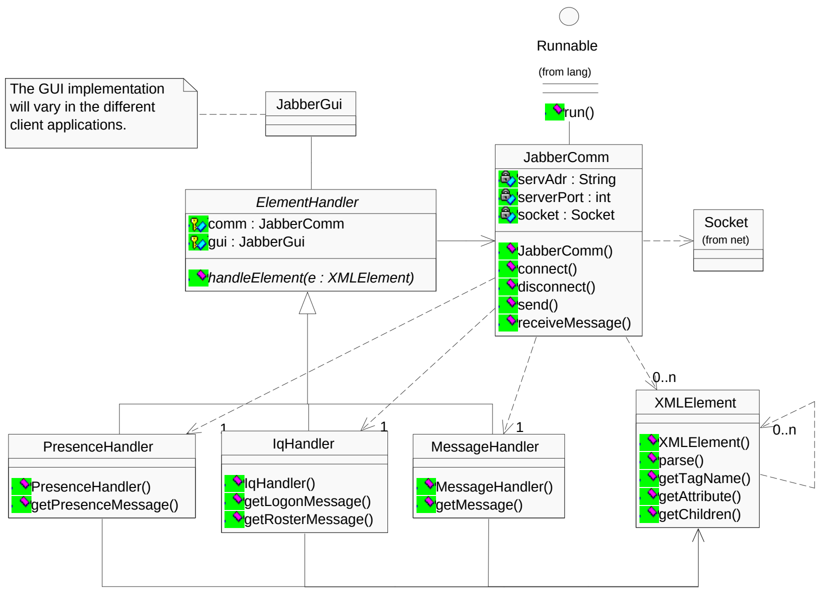


Figure 6.10: Client message handling

When parsing is complete, `JabberComm` checks the type of the XML message, e.g. `<presence/>`, `<message/>` or `<iq/>`. The message is forwarded to the corresponding handler class for that message type. These classes are `MessageHandler`, `PresenceHandler` and `IqHandler`. This design is similar to the design on the server.

Each of these classes contains functionality for extracting the elements and attributes of the particular type of message. In addition, these classes perform the actions associated with the arrival of the particular message type. Examples are updating the user interface or prompting the user for input.

The message handler classes also contain the methods for generating outgoing XML messages.

Chapter 7

Implementation and Deployment

7.1 Introduction

In this chapter we present details on the implementation of the SIGN prototype and provide instructions for deploying the server– and client applications.

We start of by presenting the server implementation, with focus on changes to the different Jaba components. Further on, Jabber related parts of the client implementations are explained. We conclude this chapter by describing how to install and run the SIGN server and the client applications.

The complete SIGN installation with source code is provided on a CD.

7.2 Server

In this section we present the changes and additions which we have made to the various Jaba components. Components not mentioned in this section are left unchanged. For each component we specify the number of lines of code which were added to existing classes and the number of lines of the new classes.

7.2.1 `Users` component

Figure 7.1 shows an ER diagram of the database used in Jaba. `Users` stores the basic information about a user – screen name[1], username and password. Although usernames are unique for each server instance, an integer id-number is given to each user. `Domains` stores the server's domain name, i.e. the name which is appended to the username to form a *jid*. Messages which cannot be sent to a user are stored in `Offline messages`.

To implement the data storage requirements of SIGN this data model has been extended and is shown in figure 7.2. As can be seen in the figure, `Device_id` has been added as an attribute to the contact-relation between users. This is to implement specific contact lists for each device. The device id is an integer with the representation listed in table 7.1.

Table 7.1: `Device_id` values

Value	**Device**
0	PC
1	Mobile phone
2	PDA

Further on, the table `Preferences` has been added to store various preferences set up by the user. `Users_id` is a foreign key to the *Users* table and

[1]The name that appears in other users' contact lists

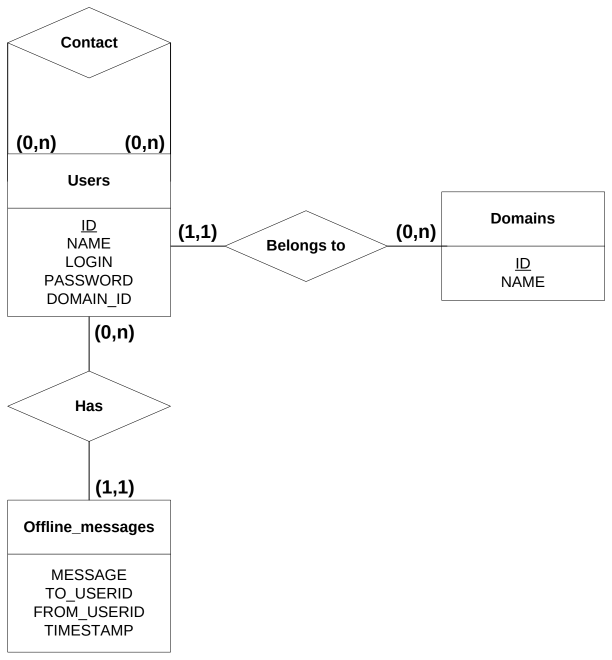


Figure 7.1: Jaba ER diagram

`Device_id` is one of the values listed in table 7.1. The `Item` field contains an option set by the user. Some of the options contain a parameter, which is stored in the `Parameter` field. Table 7.2 lists the possible `Item` values with corresponding `Parameter` values, and gives a description of the interpretation of these values.

In chapter 6 the extensions to the data-tier interface was described. Implementation of these methods is done in the `UserHomeDB` class, where existing methods for database access reside. Figure 7.2.1 shows the implementation of the method `getMaxSize()` in `UserHomeDB` and provides an example of how the `Preferences` table is accessed. The method retrieves the maximum size, set by the user, of messages destined to a mobile phone or PDA.

Implementing the changes to the `Users` component required about 200 lines of code.

7.2.2 `Jabber` component

In Jaba, XML element names must explicitly be recognized in the XML messages, otherwise they are discarded. This is a consequence of Jaba's policy of converting XML messages to object representation. As the `<presence/>` element was extended to include device information, the following changes was made to the `Presence` class:

[2] Interpreted as an e-mail address if the string contains the '@' character, phone number otherwise.

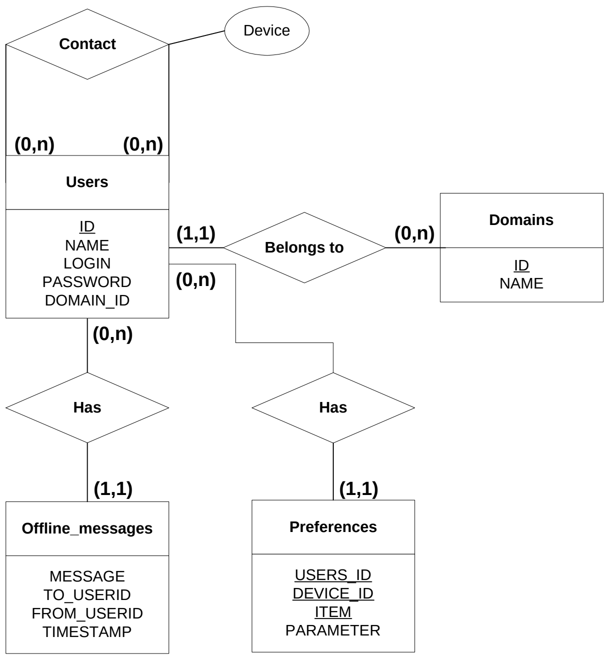


Figure 7.2: SIGN ER diagram

1. Added a string constant with the new element name: ”device”
2. Added a member variable, `device` to hold the value
3. Changed the endElement() method to look for the `<device/>` tag
4. Included the value of `device` in the `toString()` method
5. Added get and set method for `device`

According to the Jabber message protocol [20], presence messages may contain a `<priority/>` tag. This tag is not handled by Jaba, and as SIGN uses priorities, we implemented the changes above for the `<priority/>` tag as for `<presence/>`.

The two namespaces used in `<iq/>` messages led to the addition of two new classes to the `Jabber` component. These are `SQRosterConfig` and `SQPreferences`. These classes were implemented as specified in the design chapter. The `Query` class was modified to instantiate either of these classes upon detection of the corresponding namespace in the `startElement()` method.

7.2.3 `Process` component

The main change to this component was the addition of three classes, `ProcessMessage`, `ProcessPresence`, `ProcessIqExt`, responsible for processing `<mes-`

```
public int getMaxSize(long userId, int deviceId){
  Connection conn;
  int result = -1;
  try{
    conn = getConnection();
    String query = "SELECT parameter FROM " +
                    TABLE_PREFERENCES + " WHERE " +
                    FIELD_PREF_USERS_ID +"=? AND " +
                    FIELD_PREF_DEVICE_ID +
                    "=? AND " + FIELD_PREF_RULE + "=?";
    PreparedStatement pst = conn.prepareStatement(query);
    pst.setLong(1, userId);
    pst.setInt(2, deviceId);
    pst.setString(3, "size");

    ResultSet rs = pst.executeQuery();
    if(rs.next()){
      result = rs.getInt("parameter");
    }
    rs.close();
    pst.close();
  } catch (Exception e){
    System.out.println("Exception in getMaxSize()\n");
    System.out.println(e.getMessage());
  }
return result;
}
```

Figure 7.3: `getMaxSize()` implementation

Table 7.2: Preferences and parameters

Item	Parameter	Description
’block’	N/A	Block messages from people not in the contact list.
’size’	Max. message size	Truncate messages down to the specified value.
’auto’	Message	Send the specified message as auto-reply.
’forward’	Phone number or e-mail address[2]	Forward messages to SMS or e-mail.

`sage/>`, `<presence/>` and `<iq/>` messages, respectively. The first class implements the message delivery algorithm which was presented in chapter 4. The second implements routing of presence messages, which was not done correctly in Jaba. The latter is responsible for handling `<iq/>` messages for setting user preferences and device dependent contact lists. Together, the classes consist of about 250 lines of code.

Two methods were changed in `ProcessIq`, the original class for processing `<iq/>` messages:

processSet() This method is among other things responsible for handling logon. The method was changed to check which type of device the user connects with, and store this for use by other components. Four lines of code were required to implement the change.

emitAllRoster() This method is called succeeding a successful logon and sends a `<presence/>` message to users subscribing to presence information from the user who has just logged on. In addition it sends the presence state of the contacts to the user logging on. Changes to this method consisted of adding device information to the presence messages and retrieving the contact list according to which device the user is logging on with. Changes to the method amounted to about fifteen lines of code.

7.2.4 `Core` component

No new classes were introduced in this component. The changes which were made consisted of delegating the processing of messages to the three new classes in the `Process` component. For Iq messages, this requires checking if the namespace is one of those introduced in SIGN. About 30 lines of code were needed to implement the changes.

7.3 Client implementations

In this section we describe important aspect of the different client implementations, e.g. mobile phone client, PDA client and the desktop client.

7.3.1 Mobile phone - J2ME client

The client is implemented as described in chapter 6 where the functionality with regards to the handling of the Jabber XML message protocol is as show in figure 6.10.

The class `JabberComm` handles XML parsing and routing, whereas `MessageHandler, PresenceHandler` and `IqHandler` contains the logic for how to handle messages.

The J2ME client is implemented using the Model-View-Controller(MVC) design pattern [10]. The class `Model` interacts with the different message handlers on arrival and for sending of messages. The View is realized through the different `Screen` classes. Different screens provide the user with the interface for the different tasks. The main screen is the class `Contactsscreen` that displays the user's contact list and their awareness information. The class `Controller` controls what screens that are to be displayed based on different events, e.g. the arrival of a message or a user event. The screens retrieve the data to be displayed from the `Model`.

The source code of the J2ME client application is found in appendix C.1. It consists of 19 classes and comprises approximately 1500 LOC.

7.3.2 PDA and desktop clients

The PDA client was developed using PersonalJava and the desktop client was developed with J2SE. Both clients share the same basic design which can be divided into two logical components – one handles communication with the SIGN server (`Jabber` component) and the other implements the user interface (`GUI` component).

`Jabber` component

The `Jabber` component is based on the design which was presented in figure 6.10, with a separate class for processing each of the three top-level Jabber XML messages. However, the `IqHandler` for the desktop client has two additional methods for generating `<iq/>` messages with device dependent contact lists and user preferences.

In contradiction to the `Jabber` component in Jaba, this component does not create object representation of the XML messages. Instead it extracts the essential content of the message, which is passed as parameters to methods in the `GUI` component. Table 7.3 shows the methods in `GUI` which are called by the different message handlers in `Jabber`.

Table 7.3: `GUI`-component methods

Method	Invoked by	Trigger
addContact()	IqHandler	Invoked when the contact list is received from the server.
newMessage()	MessageHandler	Invoked when a message is received.
setPresence()	PresenceHandler	Invoked when awareness information is received.

`GUI` component

The GUI implementations varies substantially between the two different version. This is due to the following factors:

1. Screen-size varies significantly
2. Desktop client contains more functionality
3. Different GUI libraries are used

In the PDA client, the user interface has two main screens – one shows the contact list and displays awareness information for each of the contacts. From the other screen, the user can write messages to a contact and view an ongoing message dialog.

In the desktop client, there is a main window which displays the contact list and awareness information for each of the contacts. Separate windows are used for message dialogs with the contacts. However, both client implement the same interface for access by the `Jabber` component.

Table 7.4 shows the methods in `Jabber` used by `GUI` and gives a description of each method.

Classes and lines of code

A total of fourteen classes constitute the PDA client, where five belong to `Jabber` and five to `GUI`. Approximately one thousand lines of code is split equally between the two components.

The desktop client consists of fifteen classes – five in the `Jabber` component and ten in the `GUI` component. The lines of code count for `Jabber` component is about the same as for the PDA client – a little short of 500. With about a thousand lines of code in the `GUI` component, the total reaches nearly 1500 lines of code.

Table 7.4: Desktop– and PDA client `Jabber`-component methods.

Method	**Description**
connect()	Connects to the SIGN server on the specified address and initiates the XML conversation with the server by sending the `<stream:stream>` tag.
disconnect()	Ends the XML conversation with the `</stream:stream>` tag and closes the connection.
sendMessage()	Sends a message to a user.
sendPresence()	Sends a presence message to the server with device (PDA only), presence and context information.

7.4 XML

7.4.1 Parsers

The NanoXML parser is use in the server and desktop applications. NanoXML is a small non-validating XML parser for J2SE/J2EE, developed by Marc De Scheemaecker.

kNanoXML is a port of NanoXML, intended to run on J2ME enabled devices. This parser is used in the mobile phone and PDA client applications. kNanoXML was ported by Eric Giguere.

Both parsers can be downloaded from [25].

7.4.2 Jabber XML message protocol

The top level XML elements of the Jabber XML message protocol are `<presence/>, <iq/>` and `<message/>`. To provide the extended functionality in the SIGN prototype additions are made in the `<presence/>` and `<iq/>` element. They are described below.

`<presence/>` element

In this element we have added the sub-element `<device/>`. This element is used to carry information about the user's device. An example is given below.

```
<presence/>
  <show>away</show>
```

```
    <status>I am in a meeting.</status>
    <device>mobile</device>
  </presence>
```

In addition, the SIGN prototype uses the element `</status>` to carry information about the context.

`<iq/>` element

We have introduced two new namespaces for use in the `<iq/>` message sent from the desktop client – `jabber:ext:iq:rosterconfig` and `jabber:ext:iq:preferences`. The first is used for sending contact lists for the mobile phone and PDA client and the latter is used for sending various preferences set up by the user.

An example message for the preferences namespace is shown below.

```
<iq type="set">
  <query xmlns="jabber:ext:iq:preferences">
    <mobile>
      <block/>
      <size max="100"/>
      <forward type="SMS" address="+4791709927"/>
    </mobile>
    <pda>
      <block/>
      <size max="200"/>
    </pda>
    <desktop>
      <block/>
    </desktop>
  </query>
</iq>
```

The allowed sub elements of the `<query/>` element are `<mobile/>`, `<pda/>` and `<desktop/>`. In turn these can contain different sub elements which specify options for the different devices:

`<block/>` Sets the option to block messages from users not in the contact list for the a specific device.

`<size/>` Sets a maximum size for messages delivered to mobile phones or PDAs. The size is specified by the `max` attribute.

`<forward/>` Sets the option to forward messages to either SMS or e-mail according to the value of the `type` attribute. `address` contains an e-mail address or a phone number depending on the value of `type`.

An example contact list message is:

```
<iq type="set">
  <query xmlns="jabber:ext:iq:rosterconfig">
    <mobile>
      <item jid="lisa@hauk02.idi.ntnu.no"/>
      <item jid="albert@hauk02.idi.ntnu.no"/>
    </mobile>
    <pda>
      <item jid="betty@hauk02.idi.ntnu.no"/>
      <item jid="lisa@hauk02.idi.ntnu.no"/>
      <item jid="albert@hauk02.idi.ntnu.no"/>
    </pda>
  </query>
</iq>
```

For this message, the `<query/>` element contains the sub elements `<mobile/>` and `<pda/>`, which in turn contain `<item/>` sub elements with the *jids* of the selected contacts as an attribute.

7.5 Deployment

In this section we present how to install and run the different applications in the SIGN prototype system. Source code and necessary setup files are all provided on a CD. The directory structure is depicted in figure 7.4 with a short description of each top-level directory in table 7.5.

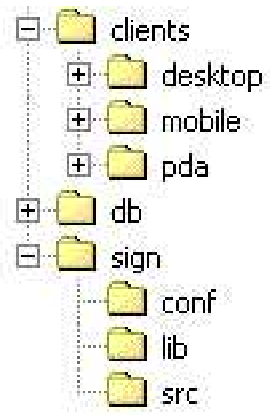


Figure 7.4: CD directory structure

In the next sections we will describe the main contents of each of the directories and describe how to install and run the different applications.

Table 7.5: Directory description

Directory	**Description**
`sign`	Contains files for the SIGN prototype server application
`db`	Contains the HSQLDB database
`clients`	Contains the different client applications, both source– and binary files

7.5.1 SIGN server application

The files concerning the server application are placed in the directory `sign`. The source code is placed in the directory `src` and necessary libraries are placed in `lib`.

The `conf` directory contains XML files for configuring different parameters of the SIGN server. The following files and parameters should be changed:

config.xml

The element `<port>` defines which port the server listens to for incoming connections. This is set to 5222, which is defined as the standard Jabber port and implemented by all Jabber clients. This should not be changed unless for testing or other special reasons.

The element `<bind>` defines the name or IP address of the machine running the SIGN prototype server and should be set correctly.

userDB.xml

This file defines the parameters for the server's access to the database. The elements `<driver>` and `<url>` defines the database driver and the URL to the database. If the database used is the one provided on our CD and it runs on the same machine as the SIGN prototype server application, these fields does not need to be changed. If another database is used or it runs on a different machine the fields must be changed accordingly. The user name and password for database access is also defined in this file.

The field `<login-domain>` defines the domain name that is given to new users. If the domain is *jabber.ntnu.no,* new users are given Jabber id's *username@-jabber.ntnu.no*.

Building and running the SIGN server

To build the application execute the `ant.bat` file, in the `sign` directory, by typing `ant sar`. Two new catalogues will be created. These are `apps` and `build`.

In order to start the server a database must be running. The database provided on the CD can be used. This is presented in the next section. To start the SIGN server simply execute the file `sign.bat`.

SQL for the generation of database tables used by the SIGN prototype server is provided in the file `sql.db`.

7.5.2 Database

The database provided on the CD is placed in the directory `db` and it is the HSQLDB database. This is a simple small database written in Java and available at [12]. Any database could be used, but for the sake of completion we provide HSQLDB.

The database is started by executing the file `runServer.bat`. The tables used by the SIGN prototype can be created by running the database manager. Execute the file `runManager.bat` and paste the script in `db.sql` for the table generation.

7.5.3 Mobile phone client

The source code of the J2ME or mobile phone client is contained in the catalogue `\clients\mobile\src`.

The compiled J2ME application, or MIDlet, is contained in the directory `\clients\mobile\bin`. MIDlets must be packaged in JAR(Java Archive) files for distribution, similar to the packaging of applets. Several extra fields must be included in the manifest file of the JAR file. This includes a new descriptor file known as a JAD(Java Application Descriptor) file.

The MIDlet JAR file for the mobile client application is named `UniChat.-jar`.

Installing the mobile client

When the MIDlet is compiled and packaged it is ready to be installed on a device. There are several ways MIDlets can be installed on different devices. Using a desktop PC the MIDlet can be transferred to the device by a data cable or an infrared (IR) connection. The MIDlet can also be installed remotely if deployed on a web site. When the MIDlet is installed it can be started from a menu on the device.

7.5.4 PDA client

The source code of the PersonalJava application can be found in the directory `\clients\pda\source`.

A compiled version of the application can be found in `\clients\pda\bin`, where there is one single class file – `Jabber.class` – and a jar file – `Jabber.jar` – with the remaining classes.

Installing the PDA client

Before the application can be run on a PocketPC device, the PersonalJava runtime environment must be installed. This can be downloaded from [27].

`Jabber.class` must be put in the `\windows\start menu` folder and `Jabber.jar` in `\Program Files\Jabber`. In addition, the images contained in the `\bin` directory must be placed together with the jar file. To transfer the files, Microsoft ActiveSync must be used, a program which comes along with the PocketPC.

The classpath for the virtual machine must be set, which requires a remote registry editor[3]

```
Change the following registry entry:

HKEY_CLASSES_ROOT
  Java Class File
    Shell
      Open
        Command
          "\Program Files\Java\bin\
          pjava.exe" -file "%1"

      to

          "\Program Files\Java\bin\pjava.exe"
          -classpath "\Program Files\Jabber\Jabber.jar"
          -setcwd "\Program Files\Jabber" -file "%1"
```

This completes the setup and the client can be run from the start menu. The PDA client can also be run from a regular PC. This is done with the following command:

```
java -classpath ".;Jabber.jar" Jabber
```

7.5.5 Desktop client

Desktop client source code is contained in the folder `\clients\desktop\src`.
An executable jar file – `JabberClient.jar` – can be found in `\clients\desktop\bin`.
The application is started by executing the jar file with the command:
`java -jar JabberClient.jar`.

[3]One come with Microsoft eMbeddedVisual Tools, available free of charge from [19].

Chapter 8

Evaluation

8.1 Scenario

In this section we give a detailed description of the scenario introduced in chapter 1. This scenario is used to demonstrate the SIGN prototype functionality. First we present the different actions that occur. Then we present the results from the run of the scenario and discuss the behaviour of the SIGN prototype.

8.1.1 Scenario description

Figure 8.1 depicts the main movements and devices used by the test user, who we have named Jill. The scenario describes a day at work, where she starts out at home, then going to a lunch meeting before ending up at the office. In the following tables we describe the actions associated to each device and important features they aim to demonstrate.

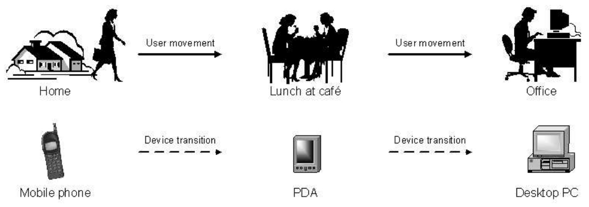


Figure 8.1: Evaluation scenario

Table 8.1 describes the beginning of the scenario where the user is at home in the morning connecting with a mobile phone.

Table 8.2 describes the actions that occur at the lunch meeting using the PDA. Jill connects to the IM service and set the presence to communicate that she is occupied in a meeting. After the meeting she communicates with a colleague at the office.

Back at the office Jill connects with the desktop PC and the subsequent actions are described in table 8.3.

8.1.2 Scenario run and results

In this section we present and discuss the results from the scenario. We conducted the test by carrying out the actions of the different users. Prerequisites of the test, as it is described, were that SIGN clients were installed on the different devices and that user accounts and contact lists are created. The different devices used during the test are listed below. The screen shots provided in this section are taken from a repetition of the scenario using device emulators.

- Mobile phone - Motorola Accompli 008
- PDA - Siemens SX45i

Table 8.1: Actions on mobile phone

#	Action	Description	Tested feature
M1	Log on		Log on, presence
M2	Receive contact list		Device dependent contact list
M3	Sending message to secretary, Albert, about the lunch meeting	Albert is displayed as available and using the desktop PC	Sending messages
M4	Receiving message from Albert		Receiving messages
M5	Set presence to “Do Not Disturb”		Presence
M6	Incoming message on server from user Simon	Message is blocked by filtering rules since not in contact list	Filtering

Table 8.2: Actions on PDA

#	Action	Description	Tested feature
P1	Log on		Log on, multiple resources, presence
P2	Receive contact list		Device dependent contact list
P3	Setting presence and context	”dnd” and ”I am in a meeting”	Presence, context
P4	Sending message to colleague Herman	Herman is displayed as available	Messaging
P5	Receiving message from Herman		Multiple resources
P6	Set presence to opaque		Transparent/Opaque communication
P7	Log off		Log off, presence

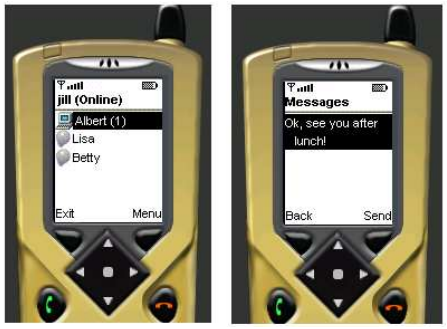
- Desktop PC

Table 8.3: Actions on Desktop PC

#	Action	Description	Tested feature
D1	Log on		Log on
D2	Receive contact list		Device dependent contact list
D3	Receiving offline message	Message sent earlier by user Simon, but blocked by filtering rule	Offline messages, filtering
D5	Adding Simon to mobile phone contact list		Contact list administration
D6	Setting preferences	Forwarding to SMS	Preferences
D7	Log off		Multiple resources

Mobile phone client

The screen shot in figure 8.2 shows the mobile phone client after completion of the 4 first actions(M1-M4) in table 8.1. Figure 8.3 shows the user Albert's view of our test user Jill, on his desktop client. The screen shot illustrates one of the extensions made to the awareness information, e.g. awareness of device. The icon illustrates that Jill is connected with a mobile phone.

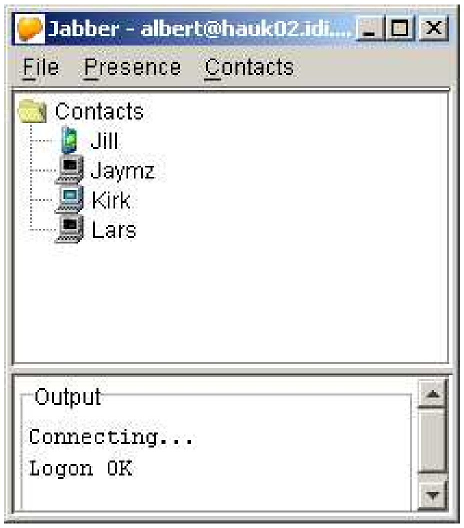


Ok, see you after lunch!

Figure 8.2: Mobile phone screen shot

After action M5 the awareness information available about Jill has changed. The presence is now set to “Do not disturb”

Due to the limited device capabilities client complexity should be reduced. This is accomplished by limiting wireless clients to only provide basic IM functionality. Messaging is illustrated by the mobile phone screen shot. The indication

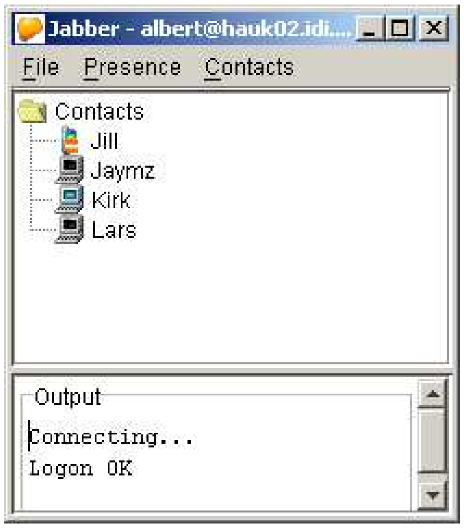


```
Connecting...
Logon OK
```

Figure 8.3: Desktop screen shot(1)

for the message received from Albert (action M4) is shown on the device by the number in the brackets, e.g. the number of unread messages.

Action M6 occurs at the server. A user, Simon, not in Jill's contact list, sends her a message. The handling of this message illustrates two important features introduced in the SIGN prototype. This is device dependent contact lists and the concept of pushing the control towards the recipient. The latter is controlled by the recipient by setting preferences.



```
Connecting...
Logon OK
```

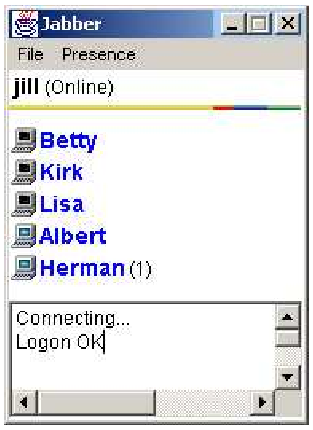
Figure 8.4: Desktop screen shot(2)

Due to the limited wireless networks the amount of data exchanged should be reduced. In the SIGN device dependent contact lists let the user control who can make contact when mobile. It reduces the amount of data exchanged as contact lists on desktop clients often contain many contacts. When on the move and using wireless devices the user should be able to limit activity and disruptiveness to only the most important contacts.

Jill has chosen, by setting preferences, to block messages from users not in her contact list when connected with a mobile phone. The message from Simon is therefore forwarded to Jill's desktop client. Since this is not online, the message is stored as an offline message. Figure 8.4 show other users' view of Jill, in this case Albert, after all actions in table 8.1have occurred.

PDA client

Screen shot 8.5 shows the PDA client after action P5 in table 8.2. Jill has connected with her PDA, exchanged messages with her colleague Herman and finally set her awareness information. Screen shot 8.6 shows Herman's view of Jill.



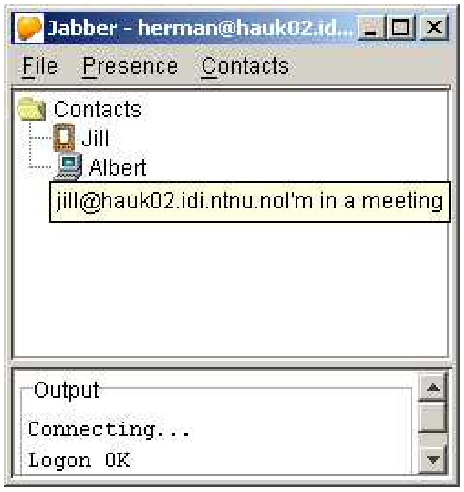
```
Connecting...
Logon OK
```

Figure 8.5: PDA screen shot

Important functionality illustrated is the possibility to have multiple active clients or resources. In many existing IM services this is not possible and login causes another active client to be logged off.

Jill is now displayed as online with her PDA, shown by the awareness of device. When Jill logs off with the PDA client (action P7), she will be displayed as online with her mobile phone which currently is her other active client. The screen shot also demonstrates the device dependent contact list since this is different than the one on the mobile phone. On the PDA Jill has added two other colleagues.

During her session with the PDA Jill also sets her context information, another extension of the awareness information. However, in the SIGN prototype the context must be written in by the user. On wireless devices with cumbersome input methods the input needed from users should be reduced as far as possible. In most situations the user is not interested in spending time writing information about the context.



jill@hauk02.idi.ntnu.noI'm in a meeting

```
Connecting...
Logon OK
```

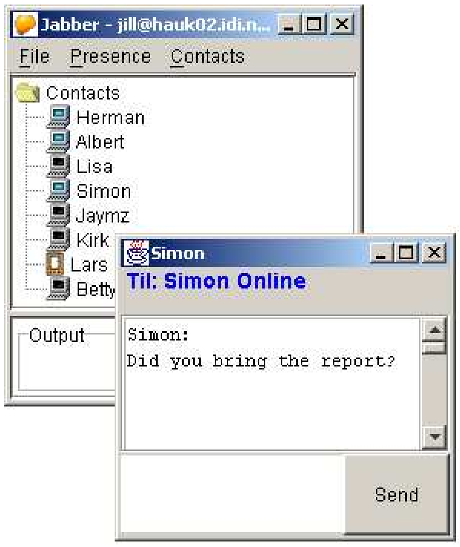
Figure 8.6: Desktop screen shot(3)

Context is provided in the SIGN prototype because it is regarded as important information about users online with wireless devices. However, context should be provided automatically with the aid of other systems. One obvious type of context information is the geographical position of a user. This can be provided by GPS. GPRS networks can also provide some degree of positioning. With interfaces to these systems the SIGN prototype could provide automatic context information since the system provides the ability to carry and display this information. Context can also be exceeded beyond the sheer geographical position of a user. Interfaces to scheduling or calendar systems could provide information about a user's current schedule and derive if the user is currently in a meeting or have other appointments. Further on sensors in buildings could derive if a user is in a meeting if it can sense more than one person in a room etc.

Desktop client

Back at the office, as Jill connects with the desktop client, the message sent earlier by Simon (action M6) is now received as an offline message. The desktop client is displayed in figure 8.7. Notice also that Jill's contact list on the desktop PC is much longer than the ones on the wireless devices.

The desktop is the interface to controlling the contact lists, setting the preferences and filter rules. Figure 8.8 shows the interface to administrate the contact lists on all the client devices and figure 8.9 show the interface to set preferences and filtering rules.



```
Simon:
Did you bring the report?
```

Figure 8.7: Desktop screen shot(4)

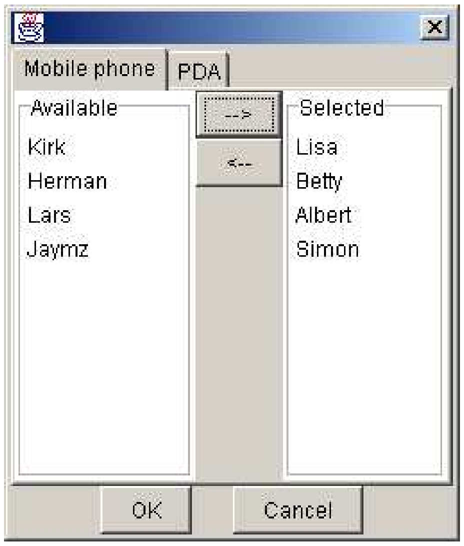
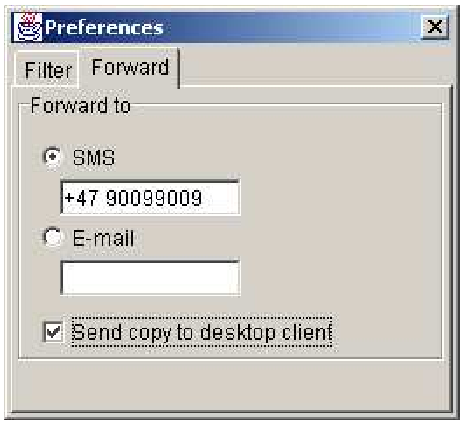


Figure 8.8: Desktop screen shot(5)

When Jill is done for the day she sets the preference to receive all incoming messages on SMS. She logs off her active clients, the mobile phone and the desktop.



Forward to

(●) SMS

+47 90099009

( ) E-mail

[✓] Send copy to desktop client

Figure 8.9: Desktop screen shot(6)

8.2 Wireless client development

During the development of the SIGN prototype we have implemented clients for desktop PCs, PDAs and mobile phones.

Our experiences from working with J2ME and PersonalJava for the wireless clients are good. With prior knowledge to Java little effort is needed to get acquainted with these Java version. Nevertheless, software development for wireless devices proved more time consuming than expected. IDEs and device emulators are available and easy to use but when testing on the actual devices, errors often occurred, not experienced on the emulators. No possibility for debugging on the actual device demanded that much time and effort had to be spent to correct errors. These errors often were due to the mismatch between the specification of the development language and the device provider's actual implementation of the Java virtual machine. This was specially the case with J2ME. In addition installation and testing on wireless devices are cumbersome.

Implementing practical and usable GUIs on the wireless devices proved difficult and time consuming.

8.3 Changes to Jabber

In chapter 6 we described how we intended to change Jaba to support the requirements to SIGN. Our focus for the changes was on isolating the changes and avoid having to implement functionality already provided by Jabber. Further on, we wanted to minimize the necessary changes to the existing source code in order to make the process of adapting to new Jabber versions easier.

Architecture

We made a decision not to make any fundamental changes to the existing Jabber system architecture, such as adding a service layer or separate server components.

As for adding the service layer, isolation of changes would have been achieved. However, a lot of functionality provided by Jabber would have to be duplicated in the layer, such as connection handling, XML parsing and user data storage. In addition we were unsure if it would actually be *possible* to implement all the requirements in the service layer without having to change Jabber at all.

The option to use separate server components was eliminated when the decision to use Jaba rather than *jabberd* was made. But in any case we had the same doubts, as with the service layer, whether or not we would be able to implement the SIGN requirements by using this approach. In some cases the need to ”undo” actions taken by the Jabber server would arise. In other cases we would not be able to use the existing functionality at all, which would have required duplication of existing functionality.

As a result, we did not change the architecture.

Design

Jaba is built up of several components which provide different functionality and we decided to implement the changes by modifying the existing components and adding a new.

The `Jabber` component is responsible for converting XML messages into object representation for further use by the other components. As changes to the message protocol was necessary, this component had to be changed. Two new namespaces are used for `<iq/>` messages containing device dependent contact lists and user preferences. This required the addition of two new classes and in turn changes in the existing class where namespaces are handled. The extended awareness information is carried within the `<presence/>` messages. Consequently the corresponding class in `Jabber` was changed.

The `Process` component is used for processing messages after they have been built up by the `Jabber` component. Changes to this component consisted of adding three new classes – one for processing each of the three top-level Jabber messages and some changes in one of the existing classes. The new class for processing `<iq/>` messages was used in parallel with the existing, with the first being responsible for handling messages under the two new namespaces. The other two classes replaced functionality which in Jaba was done by the `Core` component.

The *UserHome* component is the interface to persistent data stored by Jaba. With the additional storage requirements of SIGN, this component was changed. Instead of isolating the changes in a new class, methods for storing and retrieving data was added to the existing Jaba class.

Table 8.4 summarizes the number of changes made to the Jaba components.

Table 8.4: Changes to Jaba components

Component	**Modified classes**	**New classes**
`Jabber`	2	2
`Process`	1	3
`Core`	1	0
`Users`	1	0

Chapter 9

Related Work

9.1 Similar efforts and related work

Much research and work has been done to make tools or systems supporting collaborative work, providing messaging and awareness information. This has resulted in a number of commercial and research systems. In the resent years, with the proliferation of powerful mobile devices and better networks, some of the efforts also included support for user and device mobility.

In this chapter we present work related to the SIGN prototype. Among many systems and tools we have selected some efforts that address the same problems as we have, although there might be differences in the complexity, size and functionality of the different systems. We will compare the efforts to our work on the areas listed in table 9.1.

Table 9.1: Areas to compare

Area	**Description**
IM support	Support for basic IM features
Mobility	Support for different service access methods and devices
Awareness	Types of awareness information available
Personalization	Possibility to adapt the service to user needs
Technologies	Technologies used
Message protocol	Open, standard, proprietary or closed
System architecture	Architecture and middleware

Each of the efforts that will be discussed is listed in table 9.1 together with a short summary.

Table 9.2: Comparable work

Effort	**Domain**	**Project type**
MOTION	Platform for providing services to build collaborative applications	Research
CAMP	Modular mobile Internet portal with context awareness features	Research
PRAVATA	System providing awareness support for mobile users	Research
ConNexus	System that integrates awareness, IM and other communication channels	Research

9.2 MOTION

The MOTION project [15], MObile Teamwork Infrastructure for Organizations Networking started in February 2000 and has a duration of 30 months, making it at the time of writing near completion.

The focus is on supporting collaboration and distributed working methods. MOTION is a tool providing workers with the ability to locate artefacts, documents and experts through distributed searches, subscriptions to information and notifications, messaging and mobile information sharing and access.

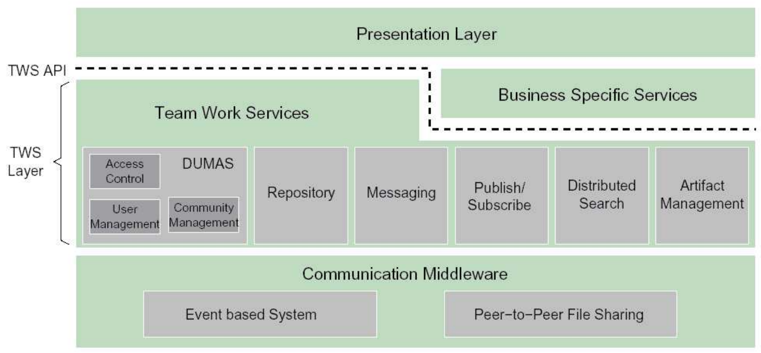


Figure 9.1: Overview of the MOTION Architecture

MOTION is a platform for providing services to build collaborative applications. It provides a basic set of services and an API such that these basic services can be customized or extended to meet specific organisation needs. The basic components are user management and access control, messaging, publish and subscribe component, repository, artefact manager and distributed search component. An overview of the MOTION architecture and its basic services is depicted in figure 9.1[1].

Here we will mainly discuss the messaging component of MOTION. This is one of the components depicted in figure 9.1 and a detailed view of this component is shown in figure 9.2[2].

The messaging service is used both for delivering messages based on subscription and for communication between users. Messages based on subscription can be received, for example, when new documents on topics the user subscribes to are entered in the system. Messages can thereby be of the following types, System-to-User and System-to-Community (mainly based on subscription), User-to-User and User-to-Community (similar to mailing lists). In this way MOTION supports basic IM features very similar to the SIGN prototype.

In MOTION the Presentation Layer provides the user interface to the services

[1]The figure is taken from [15]
[2]The figure is taken from [15]

and are built using the TWS API. In the SIGN prototype the presentation layer corresponds to the different clients currently available on mobile phones, PDAs and desktop PCs. MOTION provides access to the services through the same devices as the SIGN prototype. It is described that a typical configuration of MOTION has a number of user interfaces for different devices such as desktop computers, laptops, PDAs, Web browsers and WAP. In the implemented prototype a native Java user interface and an experimental lightweight Java PDA interface is available, although it is not specified if the TWS API can be used to build clients on all mobile devices.

The MOTION front-end component is the interface to the MOTION messaging component. It provides simple primitives for sending messages. These messages, as in the SIGN prototype, are transformed to XML events that are published through the publish/subscribe component.

There is no description of the current user interfaces and nothing is described about how contacts are handled and what awareness information is available about these contacts. However, by setting preferences, a user can decide how messages addressed to her should be delivered. In this way MOTION provides some of the same functionality as the SIGN prototype regarding personalization allowing a user to set preferences.

As shown in figure 9.2, MOTION categorises messages based on delivery mechanism and distinguishes between SMTP, SMS, WAP SI and desktop messages. The appropriate gateway receives the message, transforms it to the specific protocol and forwards it to the user. This is similar to the SIGN prototype, which also provides interfaces to SMS and SMTP. To further support mobility in MOTION, a WAP gateway is provided to support access by WAP enabled devices. The SIGN prototype does not support WAP, but we have instead chosen to provide clients developed in J2ME and PersonalJava to support PDAs and mobile phones. This makes it possible to maintain a certain consistency in the user interface across platforms. This also reduces the number of protocols that the systems needs to handle and can reduce time and complexity when implementing changes or new features.

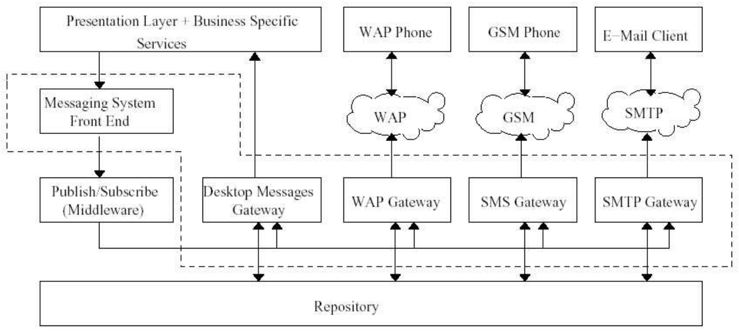


Figure 9.2: The MOTION Messaging Architecture

MOTION is a framework for building applications, whereas the SIGN prototype is focused towards instant messaging. Nevertheless, comparing the architecture of the two systems there are similarities. The functionality in MOTION is separated in to components more than is the case for Jaba implementation of Jabber. The functionality provided by Jabber can be compared to the Messaging component and Publish/Subscribe component in MOTION. Jabber also provides access control functionality, were in MOTION this is done by DUMAS. Between these components XML is used as the data format, which also is the case in Jabber.

The presentation layer in the SIGN prototype is the different clients. This layer is only coupled to the server through the XML message protocol. Thus any device able to interpret XML can function as a SIGN client. In MOTION the presentation layer is more tightly coupled to the other layers of the system through the use of specific MOTION libraries. Whereas SIGN is a client - server architecture, MOTION can also be a peer-to-peer architecture since some of the clients can host services.

9.3 CAMP

CAMP [18] is a concept for a modular mobile Internet portal enhanced with context awareness features. The goal is to make Internet services more independent of different devices, access network technologies and service providers. In addition, ensuring that services are delivered in ways that best benefit the user needs and satisfying users with small devices and limited input capabilities. CAMP comprises:

- A framework for hosting services
- A middleware for authenticating users, managing and accessing context information
- An adaptation process that allows various adaptation steps based on context information.

Since CAMP is a framework for hosting services it does not directly offer IM functionality. IM could be one of the services built upon the framework. As network services GSM SMS, WAP and WEB are mentioned, but client development is not thoroughly discussed.

CAMP is heavily focused on the context of the users. The inclusion of context awareness makes it possible to provide users with access to a variety of services that are automatically adapted to the sensed user's context. In CAMP the following is stated concerning context awareness: ”The environment is modelled as a multidimensional context space, by taking into account any factor that might influence not only presentation aspects, but also the business logic itself of any user - service relationship.” Since CAMP is an architecture and a framework for providing services, no specific model for this multidimensional context space is provided. For the SIGN prototype we have developed the dimensions of the environment for an instant messaging user. This model was presented in chapter 3.

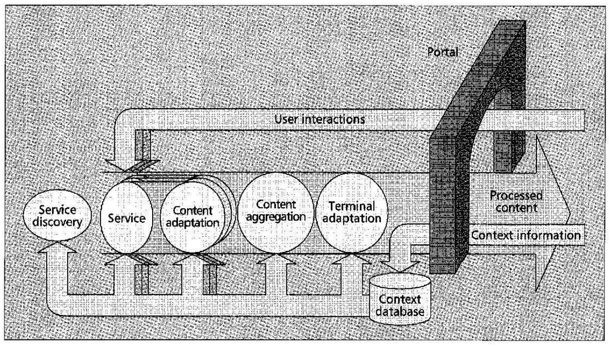


Figure 9.3: The CAMP adaptation mechanism

In CAMP actions can be triggered by comparing where the users are in this dimension against preferences set in a user profile. Adaptation of services to device and user context is discussed in CAMP but with the main focus on the server side of the system. This adaptation mechanism is presented in figure 9.3[3] and is comparable to the SIGN prototype. The user interacts with the system invoking a service. The result is adapted according to preferences, device capabilities and other context information. This adaptation mechanism is similar to the handling of data in the SIGN prototype were the XML data is adapted and sent to the users based on device and user preferences.

Internally, the system handles information in a terminal and network agnostic way where XML is used. Furthermore XSLT is used for transformation of XML documents in combination with adding or removing information based on device capabilities. XML is the message format used in the SIGN prototype.

CAMP is built upon an underlying object oriented component based Mobile Multimedia Architecture (M3A). This architecture was built for targeting mobile computing, enabling context awareness information and supporting device adaptation. M3A is based on the experiences from the MASE architecture (Mobile Information Support Environment) from the ATCS/IST On-The-Move project [16]. The component-based architecture is similar to the Jabber reference architecture describing a framework for routing XML messages. However CAMP is more component based and comprises more functionality than our SIGN prototype. As CAMP is a comprehensive framework for offering services over the Internet to different types of devices using context awareness to provide terminal adaptation, the SIGN prototype is one service that can fit well in the general framework of CAMP.

[3]The fi gure is taken from [18]

9.4 The PRAVATA prototype

The PRAVATA prototype [11] is an example implementation of a concept for ubiquitous awareness and context specific support. The prototype is developed in the context of computer-supported cooperative work, where individuals who are at different places need to cooperate. The motivation is that it is often difficult to spontaneously reach the persons that are needed at a current time. In contrast to the SIGN prototype PRAVATA does not support messaging, only awareness.

The PRAVATA prototype uses an awareness information environment called NESSIE to support awareness. In NESSIE sensors are associated with actors or other objects and capture events related to them. Events are then generated and sent to the NESSIE server that stores these events. Indicators are then used to present these events to other users. A range of indicators or interfaces are available in NESSIE. Examples are pop-up windows for computers, ticker tapes, 3D graphical presentation etc. However, all these are for stationary devices and PRAVATA extends this presentation to mobile devices. The PRAVATA Client realises the interface for mobile devices by the use of WAP and WML and enables remote access from any device equipped with a WAP browser.

However, the user has to make a query to get new information, since information-push is not supported. Only supporting WAP limits the ability to make functional user interfaces especially on more powerful devices such as PDAs and not supporting push makes the service cumbersome to use. In the SIGN prototype the different client applications act as sensors and indicators. Mobility is supported by providing Java clients for mobile devices with push support.

The PRAVATA prototype addresses some of the same problems as the SIGN prototype concerning awareness information and mobility. It is distinguished between 3 types of awareness; presence awareness, availability awareness and task awareness. More detailed description of what each of these awareness types should include is not described, but in the prototype examples of what these can be are described though very limited.

In addition, the concept of awareness contexts are described as a tool to structure and provide the information that is most relevant for a certain situation. Awareness information is structured and pre-selected in different contexts specially constructed for a given situation. Especially for mobile users it is important to easily receive the awareness information and only information that is relevant to the current situation or context the user is in. Awareness information is very similar to that provided by the SIGN prototype.

In the SIGN prototype information concerning device is captured automatically without interaction from the user. Focus is on making the system easy to use on cumbersome mobile devices and information is automatically pushed to users when the state of a contact changes. There is no mention of the user being able to set preferences in the PRAVATA prototype specifying what information that should be received.

Figure 9.4 [4] show the PRAVATA software architecture where NESSIE is depicted.

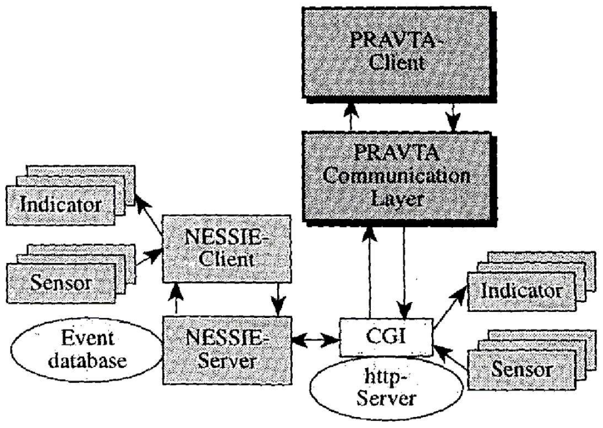


Figure 9.4: The PRAVATA architecture

The architecture of the PRAVATA prototype is similar to the SIGN prototype in many ways. PRAVATA uses NESSIE to support awareness and interfaces to stationary devices. The PRAVATA Communication Layer translates the data from NESSIE format to WML and vice versa for mobility support. It is not mentioned what protocol or data format that is used in NESSIE.

In the same way as NESSIE, Jabber handles awareness information by receiving awareness events (e.g. XML messages) and notifying other users. In the SIGN prototype the sensors are the different clients capturing information set by the user and dependent on device. In addition the clients are also the indicators presenting information to the users.

9.5 ConNexus and Awarenex

ConNexus and Awarenex [30] is an effort to integrate awareness information, instant messaging and other communication channels facilitating people to contact each other. ConNexus offers this functionality in an interface for desktop PCs. Awarenex extends this functionality to mobile devices.

ConNexus/Awarenex provides basic IM functionality similar to the SIGN prototype. In addition to providing awareness information through a contact list and supporting IM chat, the interface provides a contact toolbar enabling relevant communication resources. In this way users can switch to communication channels such as phone, e-mail etc. by clinking on a button. Figure 9.5 depicts a screen shot of the contact list, awareness information and the communication toolbar. In

[4]The fi gure is taken from [11]

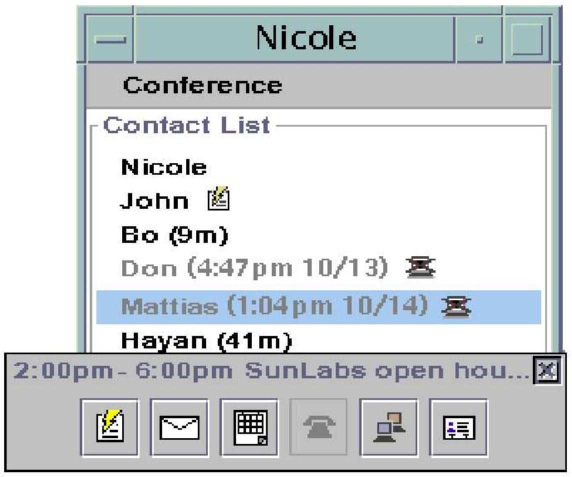


Figure 9.5: ConNexus screenshot

addition to the desktop interface interfaces for RIM [5] and Palm clients are provided to support mobility. These clients are written in C++. In this way ConNexus/Awarenex support mobility the same way as the SIGN prototype - by providing client applications.

ConNexus/Awarenex handles awareness conceptually different than in the SIGN prototype in that a user is provided with much information and has to deduce the best way to contact another user. The system provides the user with information about the local for where other users have been most recently and any communication they might be engaged in. If a user is active on the desktop computer, it is displayed that the user is in the office. By selecting one user, a list of all the locals available is displayed, including if the user is currently active in the local or how long it has been idle. For each user there will be several such locals where office, home and mobile are examples. This is similar to the SIGN prototype handling of multiple active clients or resources.

Based on local information, the user must choose how to make contact. In ConNexus/Awarenex there are little information provided about the availability of the user. This is in contrast to the SIGN prototype that provides awareness information about the availability or presence of a user. In addition the control of how a user can be reached is pushed towards the recipient. Based on user preferences and awareness the system routes communication to the desired communication channel of the recipient and not the sender.

The ConNexus/Awarenex components are similar to the SIGN prototype. Figure 9.6[6] shows the main components. The desktop and mobile device clients con-

[5]Research in Motion(RIM) Blackberry devices
[6]The fi gure is taken from [30]

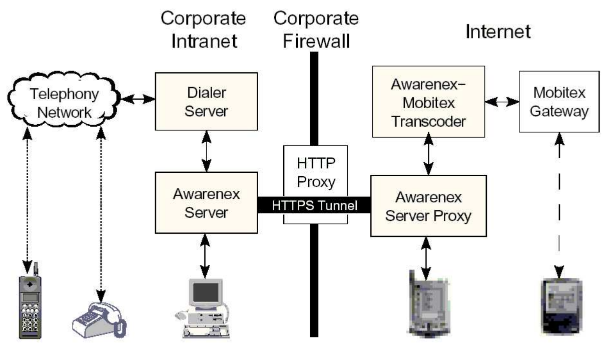


Figure 9.6: The Awarenex components

nect to the Awarenex server. This server communicates with the clients and redirect messages and events to appropriate clients. In this way Awarenex is similar to the Jabber server. There are two additional server-side components to enable the service for wireless devices, the Server Proxy and Mobitex Transcoder. It is not specified in detail what these components do. Awarenex also provides an interface to the telephone service through the Dialer Server. In this way it is possible to place calls between Awarenex users. The server components are written in Java.

The communication is done by the use of an ASCII message protocol. In the SIGN prototype XML is used to facilitate easy manipulation of data and conversion to other message protocols.

Chapter 10

Conclusion

10.1 Conclusion

We claim that in order to make IM systems mobile, merely providing clients for wireless devices is not sufficient. The SIGN prototype is a realization of an IM system supporting mobility, where Jabber is used to provide the basic IM functionality. Some functionality is extended and new functionality found important to support mobility is introduced into the system. We have developed clients for mobile phones, PDAs and desktop PCs.

Extending existing IM functionality

An important extension of basic IM functionality is the implementation of extended awareness information. The available awareness information in existing IM systems does not provide users with sufficient information about the contacts to infer their *capability* or *willingness* to communicate.

The SIGN prototype structures the awareness information along three axes. These are device, presence and location/context. Values on the three different axes can be set independently of each other. Device is an important extension because it represents information about a user's ability to communicate with others.

Context/location is added to capture that users can be in different environments and settings, which affect the user's ability and willingness to communicate. Context or location information will however not be truly valuable before the information can be deduced automatically from systems such as GPS, calendars or different types of sensors.

New functionality introduced

An important concept introduced in the SIGN prototype is that it is highly configurable by the user. In contrast to existing IM systems SIGN pushes control towards the recipient rather than the sender. SIGN provides the user with means to control how he can be reached. The user can set filtering rules for controlling how incoming messages shall be treated based on different criteria such as which type of device he is connected with.

For IM systems supporting mobility, we believe it is important to provide differentiation of functionality dependent of device. This is due to the highly different device capabilities and communication networks used. The client applications for thin wireless devices should be kept simple and the amount of information exchanged over limited wireless networks should be kept small. In the SIGN prototype the desktop client functions as a fully featured IM client and as the interface to setting preferences. The mobile phone client, on the other hand, only provides the basic IM functionality such as messaging and awareness. To reduce information exchanged the contact list can be set independently for each device.

Wireless client development

The SIGN prototype provides three different client applications. Two for wireless devices, namely mobile phones and PDAs. One client is available for desktop PCs. SIGN demonstrates that IM can be extended to mobile device used in every day life. The use of Java, J2ME and PersonalJava, shows that clients for wireless devices can be provided using the same technology. This limits the number of technologies that a user needs to know of and use for wireless access to IM. In addition it provides consistency in the user interface across different devices and eases the development effort.

Changing Jaba architecture and design

One of our research questions related to what changes were necessary to extend the Jabber architecture and design to support mobility. Our options on architectural changes was affected by the decision to use the Jaba implementation of Jabber, a decision which was influenced by factors such as Java being the preferred programming language and use of the architecture as a case study in the software architecture course SIF8056.

Having decided on Jaba, we made the decision not to make any fundamental changes to the *architecture*, instead incorporating changes to support mobility by altering the *design*. This involved extending some of the existing Jaba components, moving some functionality between components and the addition of a new component.

This violated our goal of isolating changes and avoiding duplication of functionality, however we believe the work effort was reduced compared to a more fundamental architectural change. A downside of our chosen approach is that adapting the system to new Jaba versions will require more work.

In our opinion Jaba is currently quite immature. It is not fully in compliance with the Jabber standard and lacks the ability to communicate with other IM system as the Jabber *transports* are not implemented. Until Jaba is further developed, we conclude that *jabberd* is the preferred choice.

XML and wireless clients

With regards to whether or not XML is a suitable message format for thin, wireless devices, our experience with XML in SIGN suggests that XML works very well, at least for use in an IM system. Due to the limited message size, the low frequency of messages and the relatively simple structure of the Jabber XML messages, the process of parsing the messages did not interfere with regular use of the clients.

We used the `kNanoXML` parser for the wireless clients. This is a DOM parser which builds an XML tree in memory for each message. Due to the limited memory budget of the wireless devices, using a SAX parser, which is less memory intensive, may yield better performance.

The XML element– and attribute names used in Jabber favours human readability rather than bandwidth efficiency. Together with the inherent overhead in XML messages, this is a drawback in low-bandwidth networks and packet-switched networks where users pay for the amount of data transferred.

Extending the Jabber protocol

The three building blocks of the Jabber message protocol are `<message/>`, `<presence/>` and `<iq/>`. In this thesis we wanted to find out if the message protocol could be extended to support mobility.

In SIGN the necessary extensions included information about device and context/location for the wireless clients and the ability to set the various preferences which were introduced to support mobility.

This required no changes to the `<message/>` element. The `<presence/>` element was extended to include a sub element for carrying device information, whereas context/location information was conveyed within the existing sub elements. As the Jabber message protocol makes use of XML namespaces to allow flexible extensions of the `<iq/>` message, the remaining information was contained in `<iq/>` messages under two new namespaces.

Changing the message protocol in itself was easy, but it required changes on the server.

10.2 Further work

This section presents some possible areas for further work on the SIGN prototype.

Awareness

Little was done in SIGN with regards to awareness of context and location other than enabling the users to set and view this information. Further work on automating the extraction of this information is required in order to make the system useful. If the user is responsible for providing both presence *and* context/location information, it will become a burden on the users to constantly update the information. Neglecting of this task will lead to discrepancy between the actual state of a user and the state represented in the system.

By connecting SIGN to various systems, such as GPS, calendar and sensors, the system can be developed to extract both presence and context/location information.

Usability study/experiment

In this thesis we have made the claim that in order to make IM useful in a mobile environment, it is not sufficient simply to provide IM clients for wireless devices. We thus have extended some existing IM concepts and introduced new functionality.

In order to establish if our claim is legitimate or if our proposed solutions provide users with the right tools for efficient communication, further studies are required. This could consist of formal experiments, case studies or user surveys, where a population of users make use of the system for some period of time. This could provide feedback on the existing functionality and suggestions for further improvement.

Jaba and jabberd

As we stated in the previous section Jaba is, at least for the moment, lacking in terms of functionality and maturity compared to *jabberd*. We therefore regard transition to *jabberd* as a necessity in the process of converting SIGN from its current status as a prototype to a system in production.

SIGN clients

The client applications in SIGN are implemented using different Java technologies, with J2ME, PersonalJava and J2SE used for mobile phone, PDA and desktop PC clients, respectively. Although the clients share the same design for implementing the Jabber message protocol, we did not implement a generic component for this purpose.

By implementing this component, compatible with all the different Java version, maintaining and extending the clients becomes an easier task. Further on, developing SIGN clients for emerging, Java-enabled devices will be reduced to applying this component and implementing a user-interface which can be customized to the new device.

Internal, corporate messaging

Over the last years, the business community has adopted instant messaging to facilitate collaboration between employees, and the trend seems to be increasing [5]. Using the publicly available IM systems introduces privacy and security risks as messages are sent over a public network without encryption. Further on, they do not provide more than basic messaging and presence functionality, which may not be sufficient to promote collaboration in itself.

By setting up SIGN for use on internal networks, companies can enforce security policies. Security is also one of the focus areas for the *jabberd* project. Chapter 9 showed that a lot of research has been done on systems for collaborative work, and further work on extending SIGN with functionality for this purpose can make SIGN a good alternative for internal messaging systems as the basic framework for extended awareness in place.

Appendix A

Requirements Specification

A.1 Functional requirements

The functional requirements are numbered and labelled according to table A.1. Requirements labelled FC are requirements which shall be implemented for all versions of the client applications. Requirements with the FCD prefix concern the desktop application only, whereas requirements labelled FCW are for the wireless device applications.

Requirements to the server are labelled FS. As the Jaba server already contains most of the basic instant messaging functionality, the requirements listed here are only those which are specific to the SIGN prototype.

Table A.1: Requirements labelling

Application	**Label**
Client (all versions)	FC*n*
Desktop client	FCD*n*
Wireless client	FCW*n*
Server	FS*n*

A.1.1 Client

The requirements presented in this section applies to all versions of the client applications.

General IM functionality

The following requirements concerns the basic IM functionality which shall be available in the client applications.

FC 1 - Log on
The user must enter the address of the server to log on to and must have created an account on this server beforehand (see FCD1). The user must enter the user identification (*jid*) and password. A priority may optionally be provided.

FC 2 - Log off
The user shall be able to log off the system.

FC 3 - Send message
The user shall be able to send a message to users in the contact list. Only chat-style messages shall be supported, i.e. messages that have a body but no subject field. The maximum length of a message shall be 500 characters,

however limitations may exist on some devices, which will enforce a lower maximum message length.

FC 4 - Read message
The user shall be able to read incoming messages and shall receive a notification when a new message has arrived.

FC 5 - Show *available* users
The user shall be able to choose whether he wants all contacts displayed in the contact list or only the *available* contacts.

FC 6 - Block messages
The user shall be able to set the option to block messages from users not in the contact list. The filtering shall be done on the server.

Awareness requirements

The following requirements specify how the client applications shall handle awareness.

FC 7 - Set presence
The user shall be able to set the presence state by selecting one of the values available for each version of the client application. Table A.2 shows which states are available for the three different client applications. The selected presence state shall be sent to the server.

Table A.2: Presence states

	Desktop PC	**PDA**	**Mobile phone**
Online	X	X	X
Free for chat	X	X	X
Do not disturb	X	X	X
Away	X	X	
Extended away	X		

FC 8 - Show/hide presence
The user shall be able to show or hide his presence. Show shall be the default value. In the first case, changes in the presence state shall continuously be sent to the server. In the latter case, an *unavailable* presence message shall

be sent to the server. The user will then appear to be offline to other users, but shall continue to receive both messages and awareness information from the contacts.

FC 9 - Display presence
The presence of the contacts shall be displayed in the contact list. When presence changes are received by the application, the changes shall be reflected on the screen.

FC 10 - Display device
The user shall be able to view the device information set by the contacts. If no device information is available for the contact it shall be interpreted as if he is using a desktop PC. Separate representations shall be available for users connected with a mobile phone and users connected with a PDA.

FC 11 - Set context/location
The user shall be able to set the context/location in which he is using the system. This shall be done by allowing the user to write textual representation of the context/location. The maximum length shall be 15 characters. When entered, the information shall be sent to the server.

FC 12 - Display context/location
The user shall be able to view context/location information set by the contacts.

A.1.2 Desktop client

Requirements FC1 – FC12 apply to the three different client applications. The requirements presented in this section apply to the desktop application only.

FCD 1 - Create account
User must create an account before they can start using the system. The user must enter the address of the server on which the account shall be created and enter a username and password. If the username is not unique for that server, the user shall be prompted to enter a different username.

FCD 2 - Set personal details
The user shall be able to provide personal details which can be viewed by other users. The following details can be set: name, nickname, e-mail address and phone number.

FCD 3 - View contact details
The user shall be able to view the personal details of users in the contact list.

FCD 4 - Authorize subscription
The user may require to authorize other users before they can add the user to their contact lists and subscribe to awareness information.

FCD 5 - Add contact

The user shall be able to add a contact to the contact list by entering the contact's *jid* and optionally a nickname to be shown in the contact list. The application must then send a subscription request to the contact.

Contact lists may be specific for each device, and if the contact allows the subscription, the user shall be able to choose which contact lists the contact shall be added to.

FCD 6 - Remove contact

A user may remove a contact. The contact shall be removed from all contact lists, and an un-subscription message shall be sent to the server.

FCD 7 - Edit contact lists

The user shall be able to edit the contact lists for all the devices using the desktop application. For each contact the user has added, he shall be able to choose if that contact shall be included in the contact list sent to each device category. Changes in the contact lists are effectuated the next time the user logs on from each version of the client application.

FCD 8 - Block messages

The user shall be given the option to block messages from users not in the contact list. This shall be done on a per device basis, for instance all messages are accepted by the desktop application, but only messages from contacts on the mobile devices. The blocking shall be done on the server.

FCD 9 - Set maximum message length

The user shall be able to specify a maximum length for messages sent to the mobile client. Messages exceeding the specified shall be truncated on the server.

FCD 10 - Set forwarding

The user shall be able to set the option to forward messages to SMS or e-mail. When set, the user must enter the phone number and/or the e-mail address. The forwarding shall be done based on the following criteria set by the user:

Forward if not connected with any client

Forward all messages

The user shall be able to put restrictions on the messages that are sent to the mobile devices, however this can only be done from the desktop client. If the user is connected only with a mobile device client, filtered messages are treated as offline messages and sent when the user connects with a desktop client.

A.1.3 Wireless clients

The following requirements are specific for the J2ME and PersonalJava client applications.

FCW 1 - Device type
The wireless device client application shall be responsible for notifying the server of what type of device it is running on. This shall be done immediately after a successful logon, unless the user has chosen not to show the device type (see FC2). The PDA application shall use the value *'pda'* and the mobile phone application shall use the value *'phone'*.

FCW 2 - Show/hide device
The user shall be able to show or hide the device with which device he is connected. The first case is the default value. In the latter case a presence message with no device information shall be sent to the server.

FCW 3 - Set auto reply message
The user shall be able to specify a message that will be stored on the server and sent as an automatic reply when new messages are received. For example: “I’ve read your message. I’ll get back to you in a few minutes”.

FCW 4 - Set forwarding
The user can set the option to forward incoming messages to SMS to a predefined mobile phone number.

FCW 5 - Automatic presence change
If the user accepts a phone call, the presence shall automatically be set to *busy*.

A.1.4 Server

FS 1 - Create account
A username and password must be provided by the user in order to create a new account. If the username already exists, the account shall not be created and the user shall be notified of this. The new user information shall be stored in the database.

FS 2 - Log on
During a logon attempt the following information must be received from the user:

- User name or Jabber ID
- Password
- Device category

If the user name and password are valid, the device dependent contact list shall be sent to the user along with awareness information about the users in the list.

FS 3 - Authentication
Authentication shall be performed based on the user name and password provided during log on.

FS 4 - Contacts list storage
Contact lists shall be stored on the server and sent to the client subsequent to log on.

FS 5 - Device dependent contact lists
The user may have separate contact lists for the different devices. On the server one global contact list shall be stored in addition to one contact list per device. The global list will contain all the user's contacts and the contacts in the different device dependent lists will come from this list.

FS 6 - Process message
The server shall receive messages from the users and route them to the recipient(s). A message shall be processed according to preferences set by the users.

FS 7 - Process preferences
The client can set the preferences specified in table A.3.

Table A.3: Preferences

Preference	Description
Maximum message size	This preference can be set for wireless devices and set a limit for maximum size of a messages sent to these clients.
Block messages	If this preferences is set messages from other users than listed in the contact list are blocked.
Forward to e-mail	Preference set to enable forwarding of messages to an e-mail address.
Forward to SMS	Preference set to enable forwarding of messages a mobile phone as SMS.

The preferences shall be stored in the database.

FS 8 - Message filtering
Messages can be filtered on the server according to user preferences. Filter-

ing is done based on the rules listed in table A.4. Following the table is a further description of the rules.

Table A.4: Filter rules

Filter rules	Description
Message size	The size of a message exceeds a certain limit
Block messages	Only messages from the users in the current contact list is sent to the client

Message size
If the message size preference is set and a message exceeds the limit the following actions shall be performed.

- Reduce message to the specified size
- Send reduced message
- Send complete message to desktop

Block messages
If the sender is not in the contact list and this preference is set the following actions shall be performed.

- Check other desktop contact lists
- If found, send to desktop
- If not found, discard message

FS 9 - Auto reply
If the user has set the preference for auto-reply message the server shall automatically send the reply message to the sender of the message.

FS 10 - E-mail message forwarding
The server shall be capable of forwarding messages to an e-mail account. Based on preferences set by the user messages can be sent to a predefined e-mail address stored on the server. The server shall provide an interface to an e-mail gateway.

FS 11 - SMS message forwarding
The server shall be capable of forwarding messages to SMS. Based on preferences set by the user messages can be sent to a predefined mobile phone number stored on the server. The server shall provide an interface to an SMS gateway.

FS 12 - Process awareness events
Awareness events received from the users shall be routed to other users. Only

users who subscribe to awareness information from the sender shall receive the information. Awareness messages shall be routed according to preferences set by the recipient.

FS 13 - Preferences stored on the server
Preferences for each user shall be stored on the server.

FS 14 - Device dependent preferences
Preferences can be set up independently for each device category. These shall be stored on the server and during message processing and filtering the preferences used shall be according to the currently active device.

FS 15 - Offline messages
A message destined for a user which is not currently online or has not set any forwarding rules, shall be stored in the database. The offline messages are sent to the user the next time he logs on to the system, and are then subject to the same filtering rules as online messages.

FS 16 - Multiple active clients
A user shall be allowed to keep several clients active simultaneously, allowing connections from different devices at the same time. The concept of resources in Jabber shall be used to make distinctions between simultaneous connections and decide to which client to route messages. Each client is attached to one resource. In addition each resource has a priority that is used to determine to which connection or client messages intended to that recipient should be sent. The connection with the highest priority value is the connection to which the messages are sent. Priority values must be a positive integer and cannot be 0. The connection with the highest priority value has the highest priority, thus 1 has lower priority than 5. In the event of a tie between priorities, messages should be routed to the most recent connection. In addition preferences set by the user can effect where messages should be routed, thus overriding the use of priorities.

A.2 Non-functional requirements

A.2.1 Client

NF 1 - User training
User should be able to use the system without any formal training or need for reading the user manual, provided that they have basic knowledge of Instant Messaging systems and concepts.

NF 2 - Consistent user interface
User interfaces on the different client applications shall be similar to a maximum possible extend. Menus shall have the same names and consist of the same choices when similar functionality is available on the different clients. This shall assure that users familiar with one type of client easily can use other client applications.

NF 3 - Core client parts
Three different client applications shall be made. These clients will run on different devices and operating systems. Clients shall be written in Java and core parts of the client applications shall be identified and tried separated into components. In this way core components can be reused in all three client applications.

A.2.2 Server

Reliability

The Jabber technology used in this system is open source. Since no measure concerning the failure rate of Jabber technology is available and extensive testing of Jabber technology stability is out of the scope for this project, specifying maximum downtime for the system is difficult.

Security

NF 4 - Authentication
All users of the system must be authenticated by a username and password.

NF 5 - Protection of data
Standard users shall only have the possibility to read, delete, modify or add data concerning their own user preferences or standard data concerning these users. Only users granted Administrator access rights shall gain access to Administrator functionality. Access to the server machine is assumed handled by access restrictions in the operating system.

Maintainability

The system will be made in an educational setting. Time may prevent implementation of all possible features of a fully functional IM system. However emphasis shall be placed on making an architecture and code such that changes and additions easily can be made to the system. This makes it possible for others to use this system as a basis for further studies on IM systems. The system shall also be used for an architecture assessment study for 4th grade students at NTNU.

NF 6 - Components
The functionality of the system shall be divided in separate components handling logically different functionality. It should be possible to make changes in one component without having to make changes in other components. This shall make it possible to work with and change one part of the system with limited knowledge of other parts.

NF 7 - Loose coupling
There shall be very loose coupling between the different components in the system, handling logically separated functionality. Substituting one component with another component handling the same functionality shall take less than one working day provided that the new component are ready for substitution.

NF 8 - Adding of extensions
Adding of new functionality to the system shall be simple. Functionality natural in a complete IM service may be left out due to time constraints. However it should be possible to continue work on this system in other projects making it into a more complete system.

A.2.3 Implementation

NF 9 - Programming language
The primary programming language of the project is Java. The following versions shall be used for the different parts of the system:

- Java Development Kit 1.3.1 (JDK 1.3.1) for programming server side and desktop client.
- J2ME Wireless Toolkit 1.0.3 for programming mobile phone client.
- PersonalJava 1.2a for programming PDA client.

Other programming languages can be used if necessary to change existing software components or produce necessary glue code. All versions of the programming environments above are the latest release at the time of writing.

NF 10 - Modelling language
UML shall be used as the standard modelling language. This is a requirement

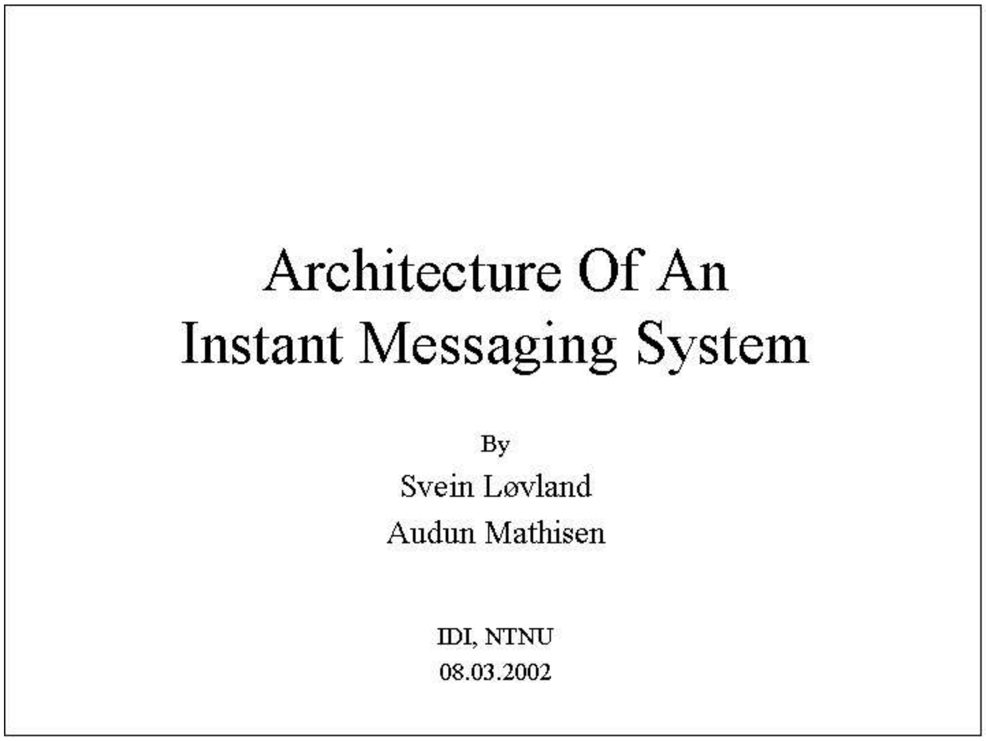
because the students who will evaluate the system in the software architecture course use UML.

NF 11 - XML

The system shall be based on exchange of XML messages. XML is the standard used in Jabber and will ensure interoperability with existing Jabber clients. In addition is XML a flexible and standard way of exchanging information across different platforms. The Jabber XML message protocol shall be used. In addition other message protocols or changes to Jabber XML protocol can be made to support thin clients such as mobile phones or PDAs. Clients using the standard Jabber message protocol shall be able to use the system.

Appendix B

SIF8056 Presentation



Architecture Of An Instant Messaging System

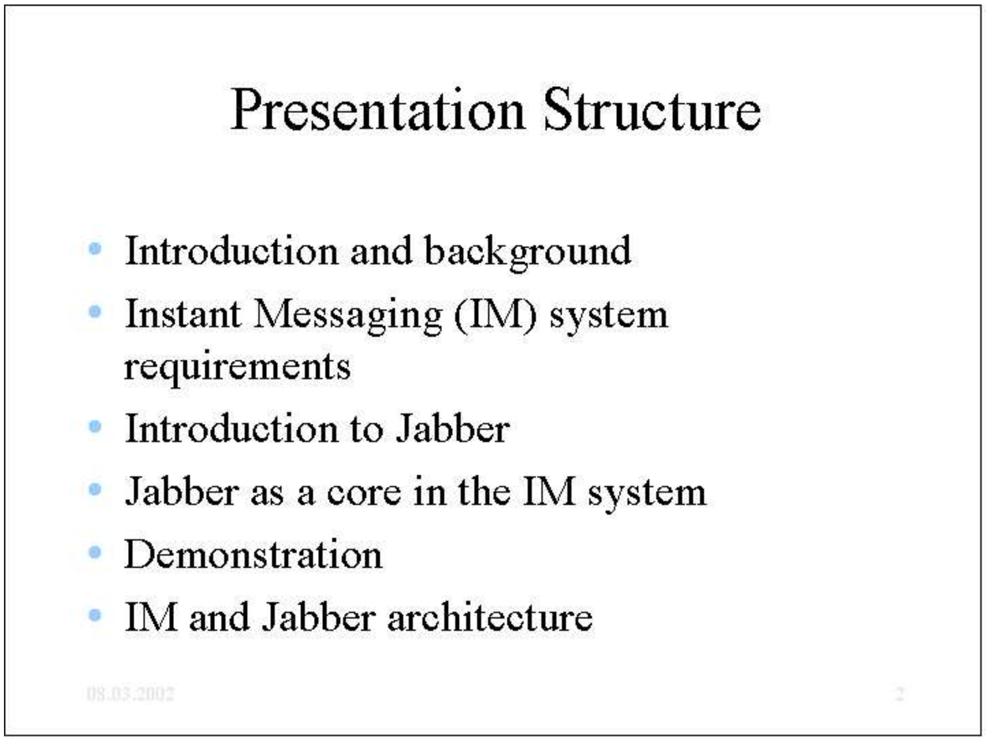
By

Svein Løvland

Audun Mathisen

IDI, NTNU

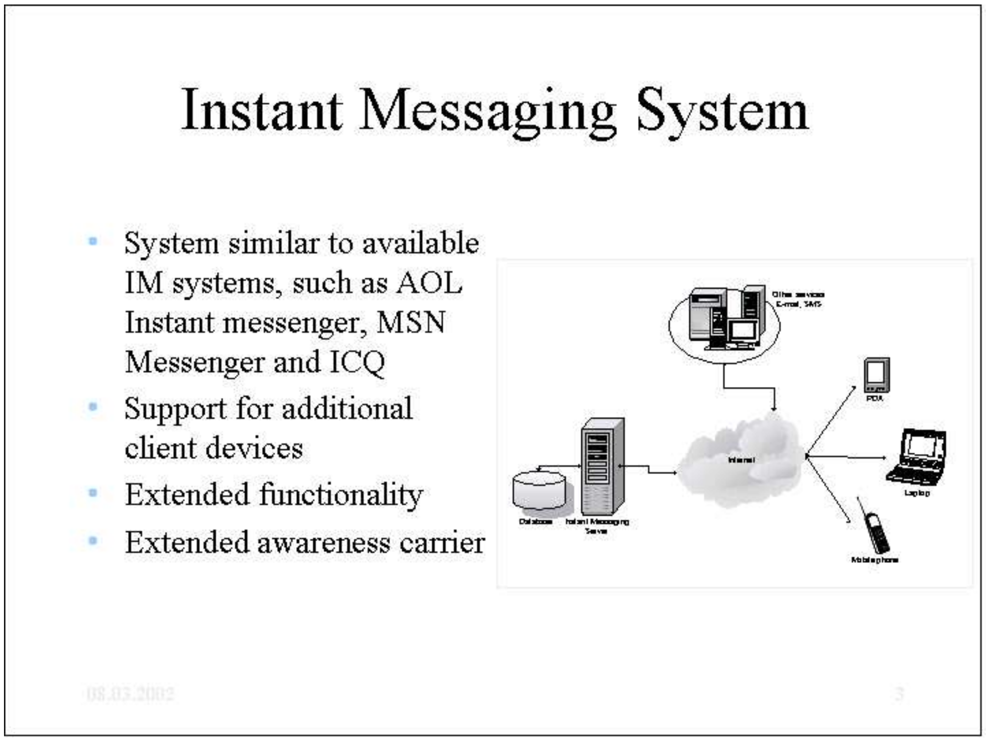
08.03.2002



Presentation Structure

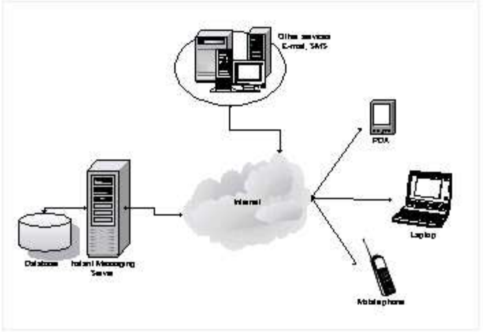
- Introduction and background
- Instant Messaging (IM) system requirements
- Introduction to Jabber
- Jabber as a core in the IM system
- Demonstration
- IM and Jabber architecture

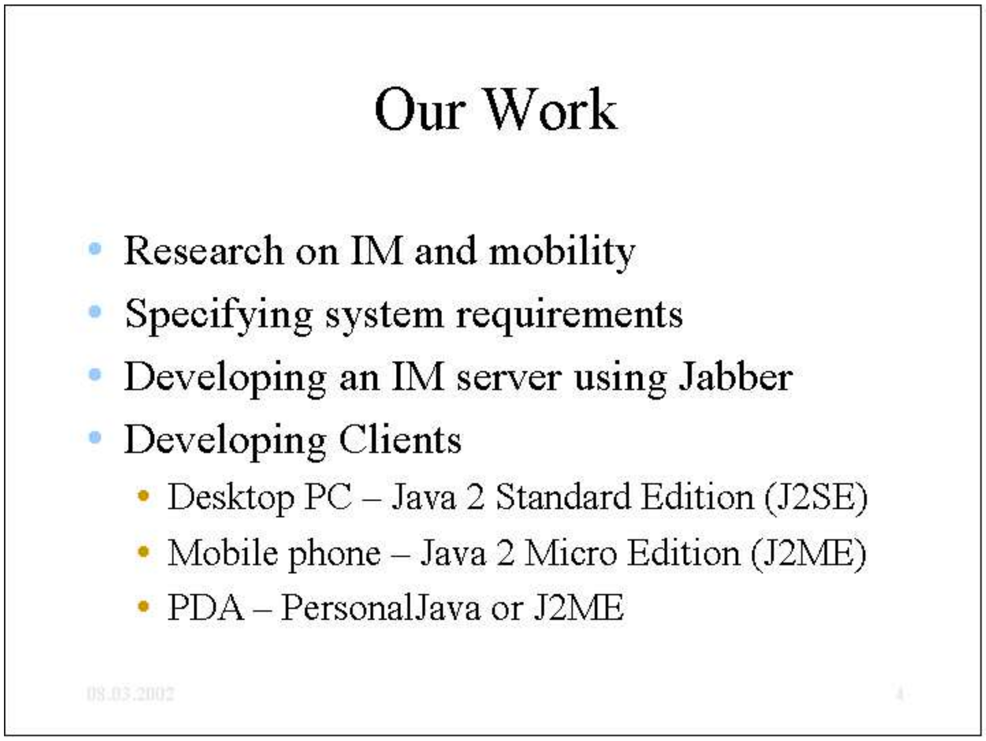
08.03.2002 2



Instant Messaging System

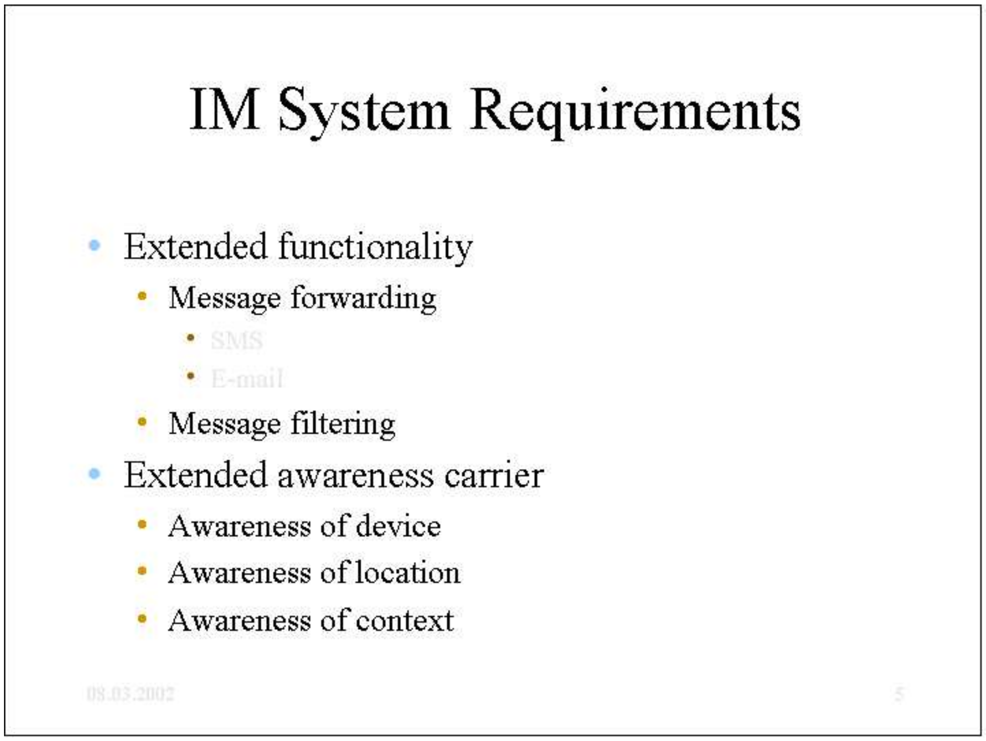
- System similar to available IM systems, such as AOL Instant messenger, MSN Messenger and ICQ
- Support for additional client devices
- Extended functionality
- Extended awareness carrier





Our Work

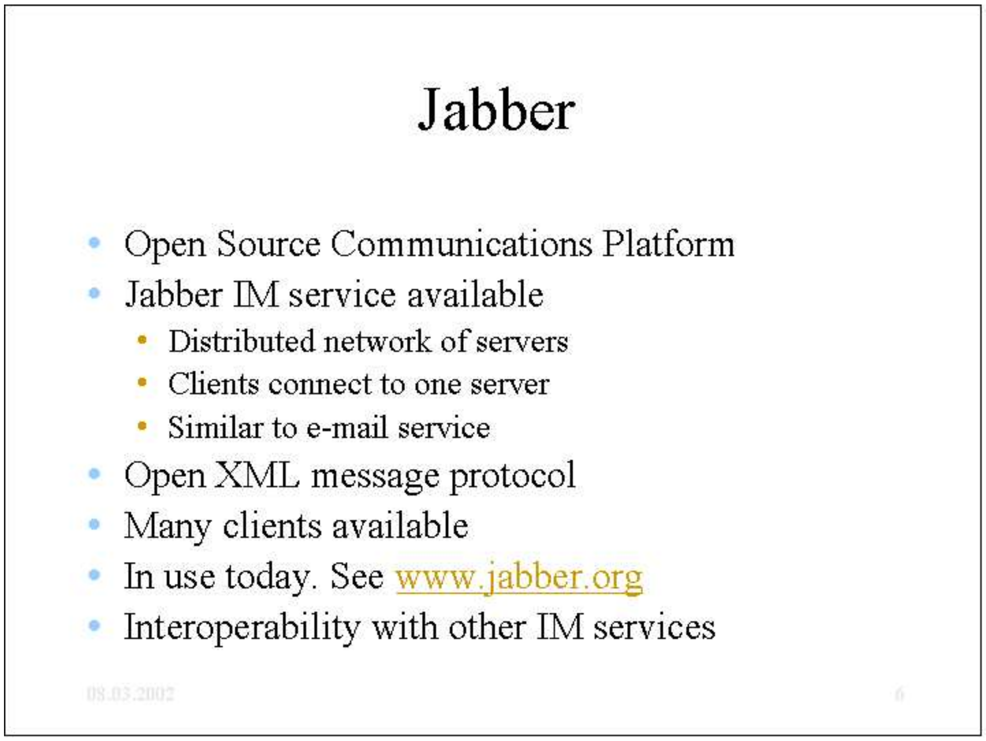
- Research on IM and mobility
- Specifying system requirements
- Developing an IM server using Jabber
- Developing Clients
  - Desktop PC – Java 2 Standard Edition (J2SE)
  - Mobile phone – Java 2 Micro Edition (J2ME)
  - PDA – PersonalJava or J2ME



IM System Requirements

- Extended functionality
  - Message forwarding
    - SMS
    - E-mail
  - Message filtering
- Extended awareness carrier
  - Awareness of device
  - Awareness of location
  - Awareness of context

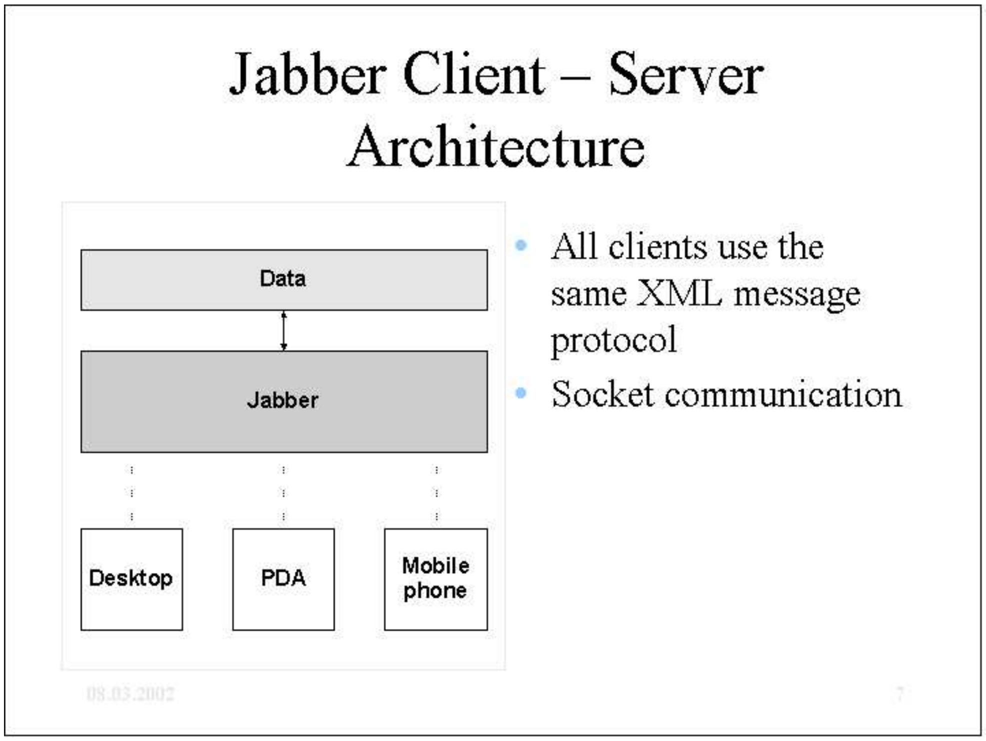
08.03.2002 5



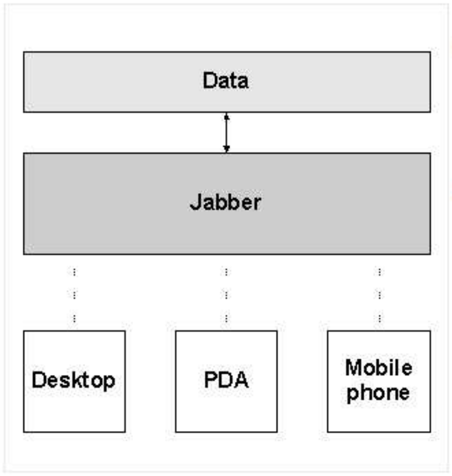
Jabber

- Open Source Communications Platform
- Jabber IM service available
  - Distributed network of servers
  - Clients connect to one server
  - Similar to e-mail service
- Open XML message protocol
- Many clients available
- In use today. See www.jabber.org
- Interoperability with other IM services

08.03.2002 6

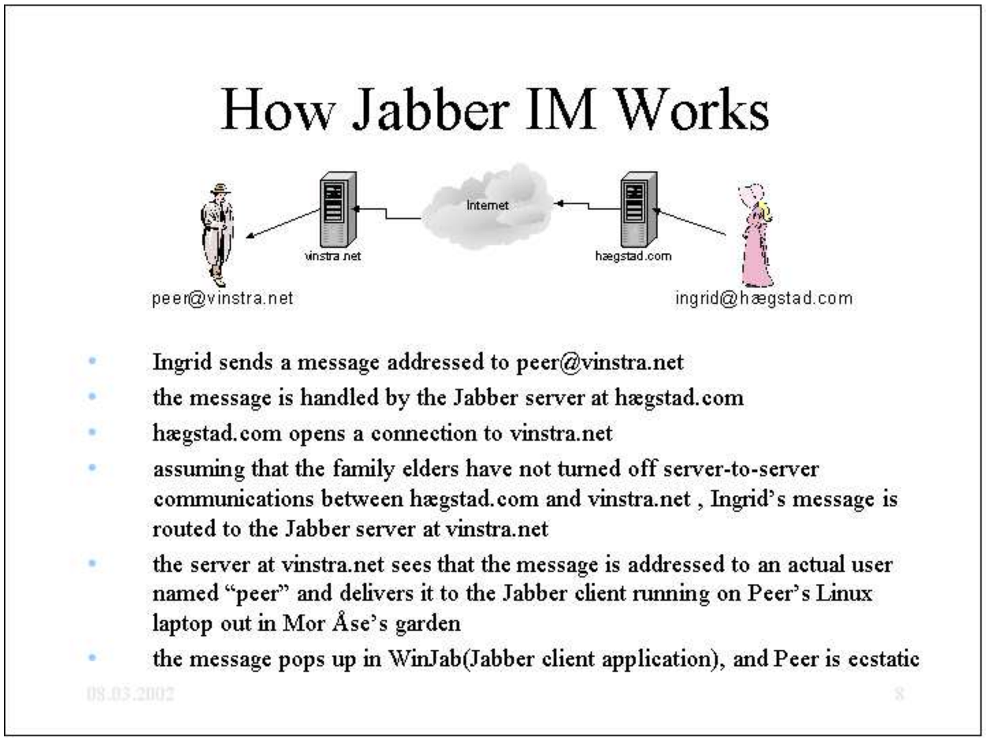


Jabber Client – Server Architecture

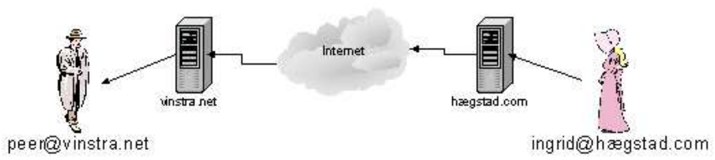


- All clients use the same XML message protocol
- Socket communication

08.03.2002 7

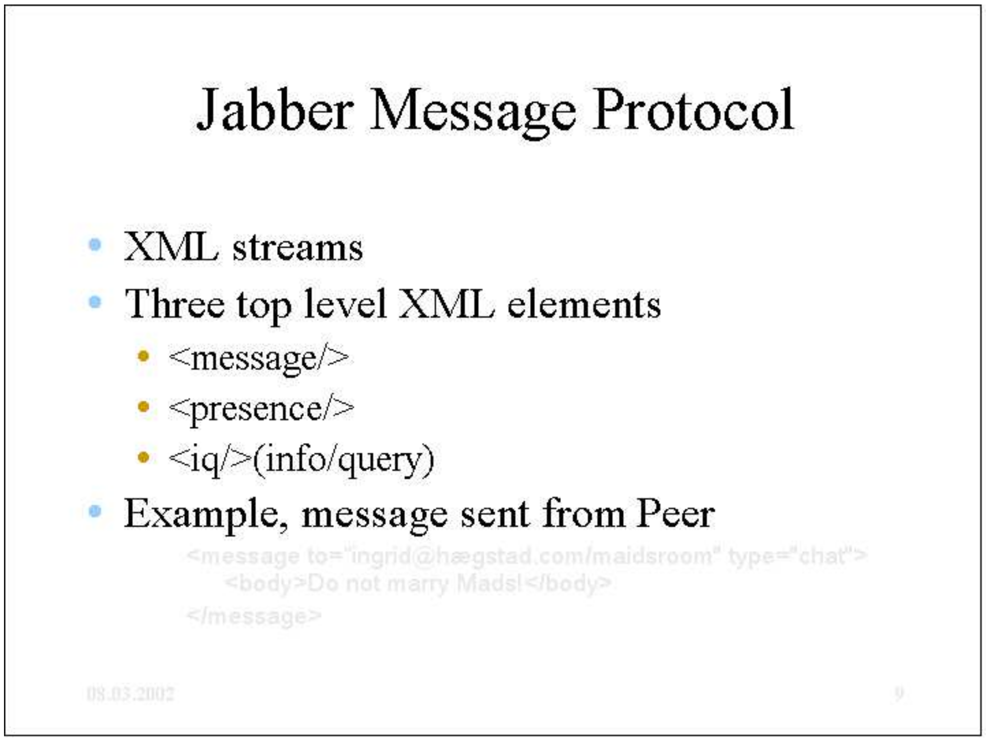


How Jabber IM Works



- Ingrid sends a message addressed to peer@vinstra.net
- the message is handled by the Jabber server at hægstad.com
- hægstad.com opens a connection to vinstra.net
- assuming that the family elders have not turned off server-to-server communications between hægstad.com and vinstra.net , Ingrid's message is routed to the Jabber server at vinstra.net
- the server at vinstra.net sees that the message is addressed to an actual user named "peer" and delivers it to the Jabber client running on Peer's Linux laptop out in Mor Åse's garden
- the message pops up in WinJab(Jabber client application), and Peer is ecstatic

08.03.2002 8



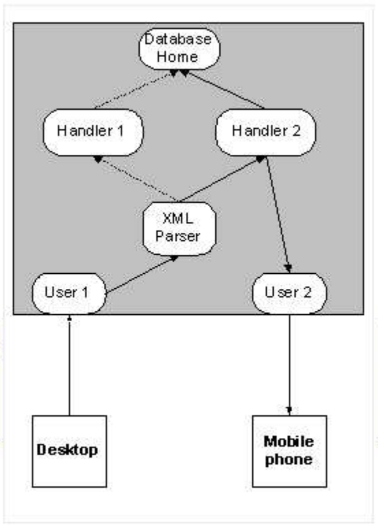
Jabber Message Protocol

- XML streams
- Three top level XML elements
  - <message/>
  - <presence/>
  - <iq/>(info/query)
- Example, message sent from Peer

```
<message to="ingrid@hægstad.com/maidsroom" type="chat">
    <body>Do not marry Mads!</body>
</message>
```

08.03.2002 9

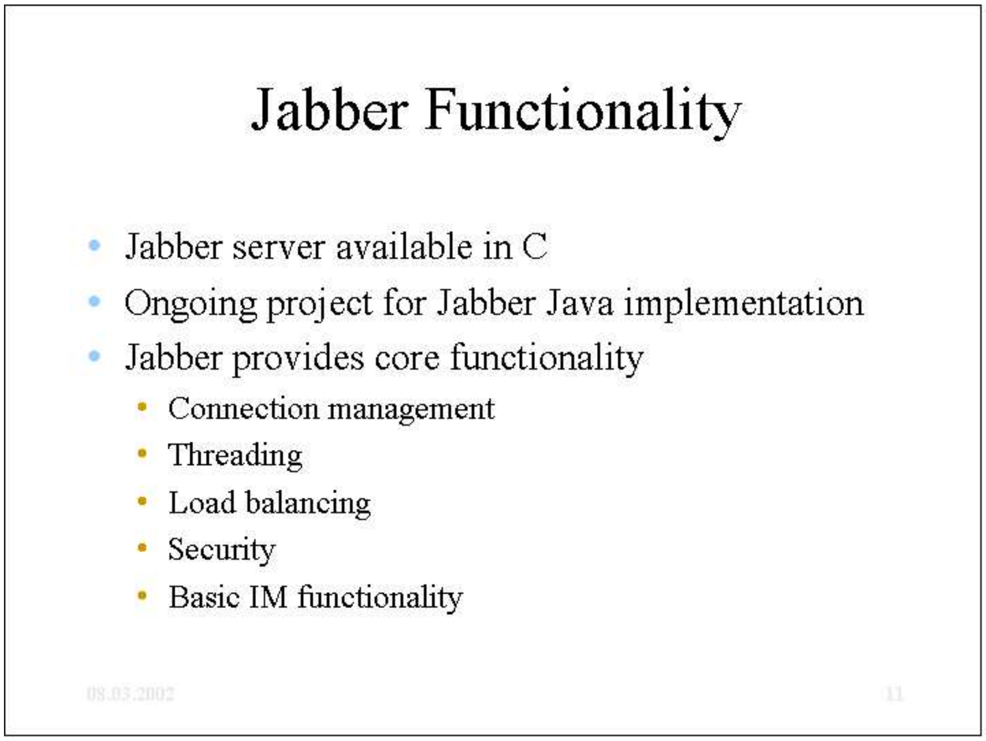
Jabber Components



```
<message to="ingrid@haegstad.com/maidsroom">
<body>Do not marry Mads!</body>
</message>
```

```
<message from="peer@vinstra.net">
 <body>Do not marry Mads!</body>
</message>
```

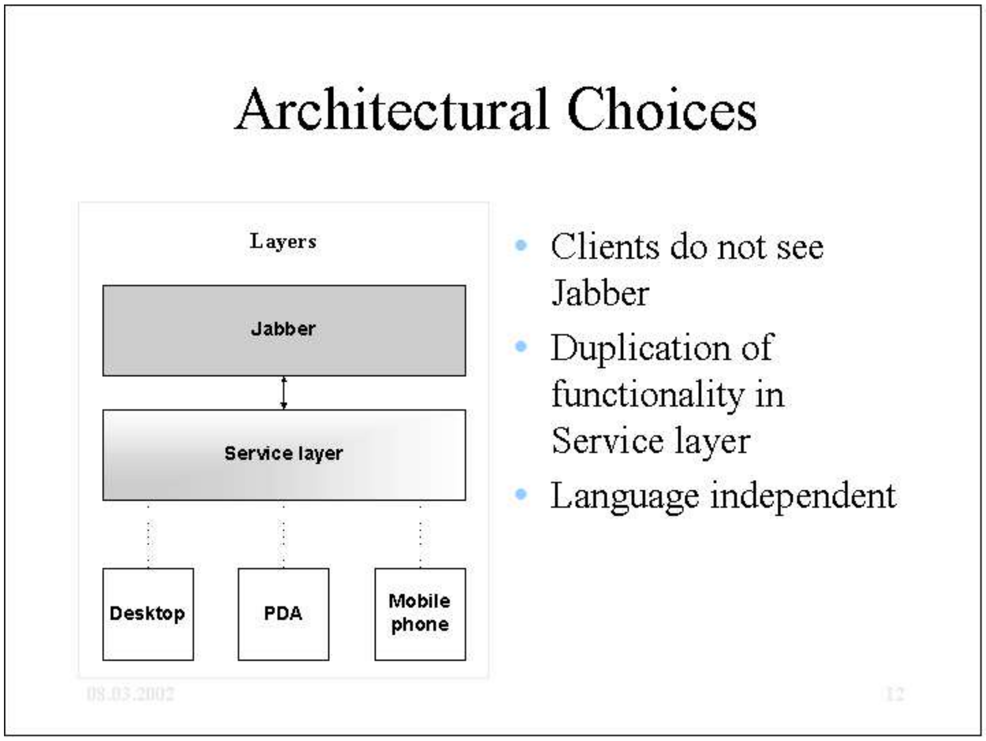
08.03.2002 10



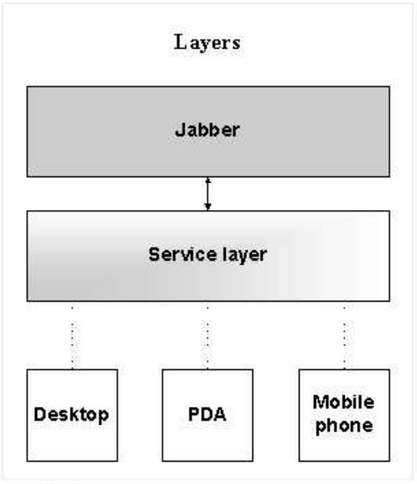
Jabber Functionality

- Jabber server available in C
- Ongoing project for Jabber Java implementation
- Jabber provides core functionality
  - Connection management
  - Threading
  - Load balancing
  - Security
  - Basic IM functionality

08.03.2002 11



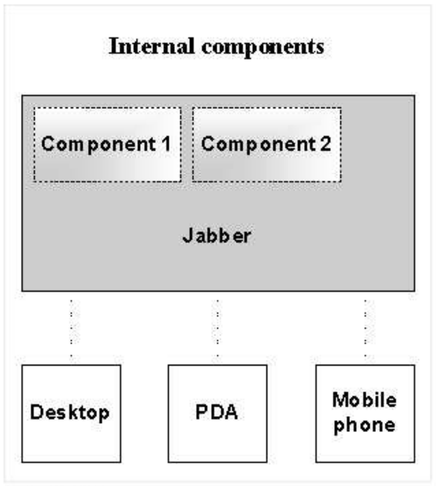
Architectural Choices



- Clients do not see Jabber
- Duplication of functionality in Service layer
- Language independent

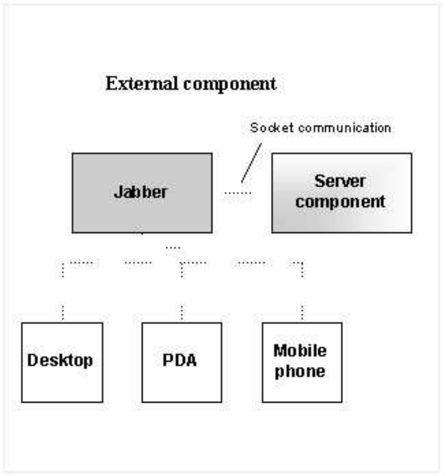
08.03.2002 12

Architectural Choices



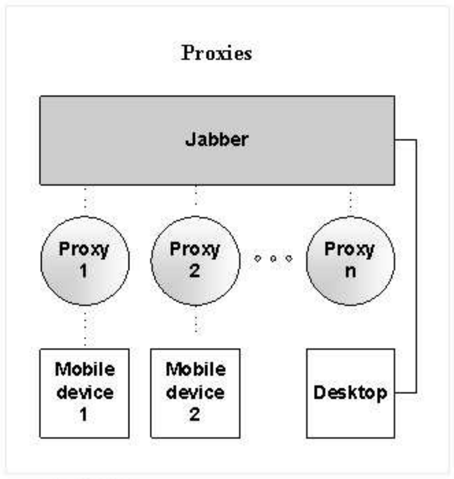
- Clients only see Jabber
- New functionality is added by components
- Internal communication by method calls
- Language dependent
- Components loaded at runtime

Architectural Choices



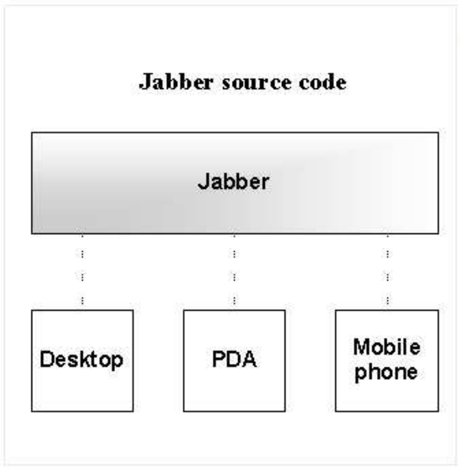
- Clients only see Jabber
- New functionality is added by components
- Server components communicate through sockets
- Language independent
- Components connect at start-up

Architectural Choices



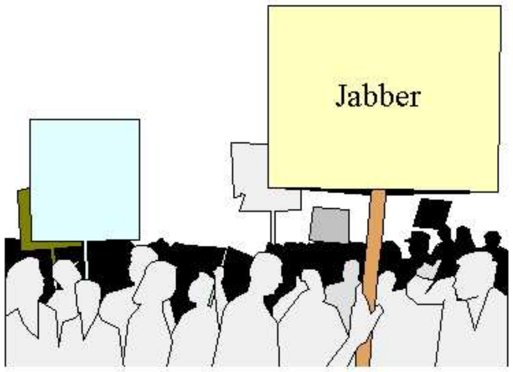
- Clients may or may not see Jabber
- Separate proxy for each type of device
- Some duplication of functionality
- Language independent

Architectural Choices

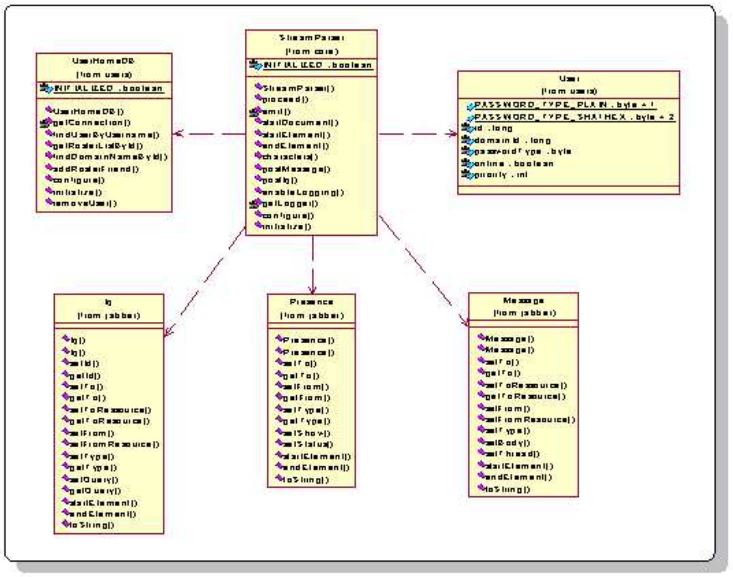


- Changes done in source code

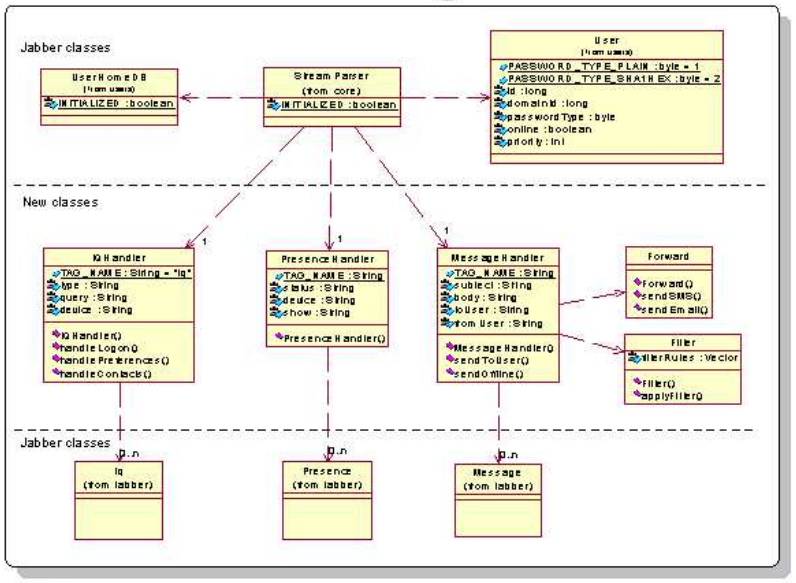
Demonstration



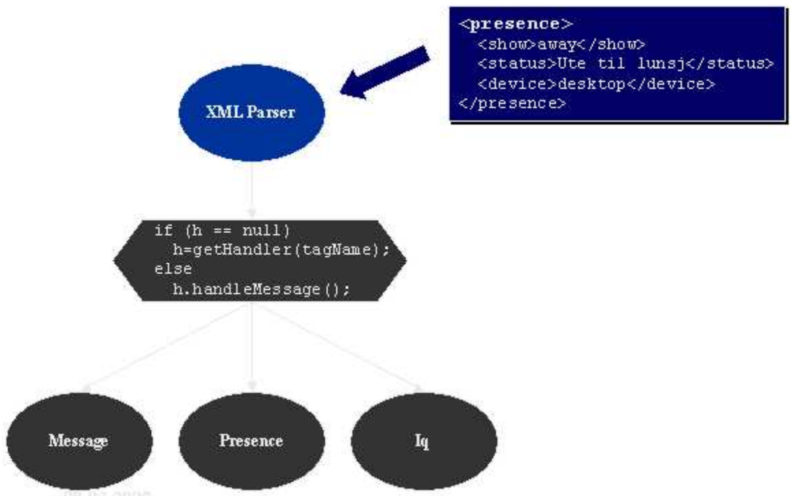
Jabber Classes



Jabber Integration

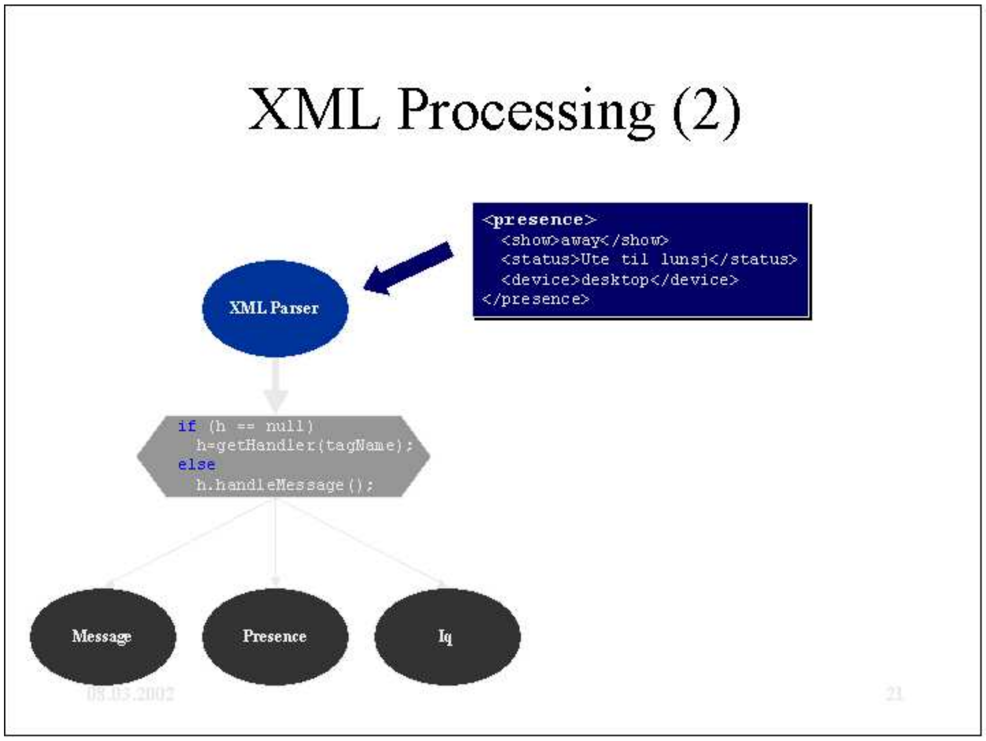


XML Processing (1)



```
<presence>
  <show>away</show>
  <status>Ute til lunsj</status>
  <device>desktop</device>
</presence>
```

```
if (h == null)
  h=getHandler(tagName);
else
  h.handleMessage();
```

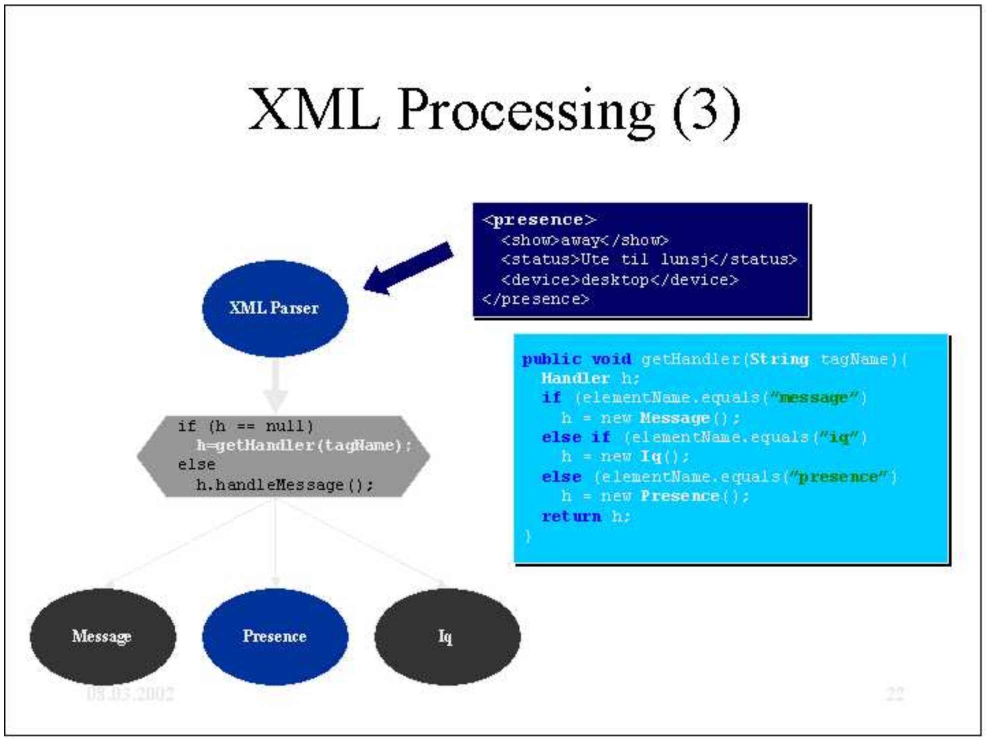


XML Processing (2)

```
<presence>
  <show>away</show>
  <status>Ute til lunsj</status>
  <device>desktop</device>
</presence>
```

```
if (h == null)
  h=getHandler(tagName);
else
  h.handleMessage();
```

08.03.2002 21



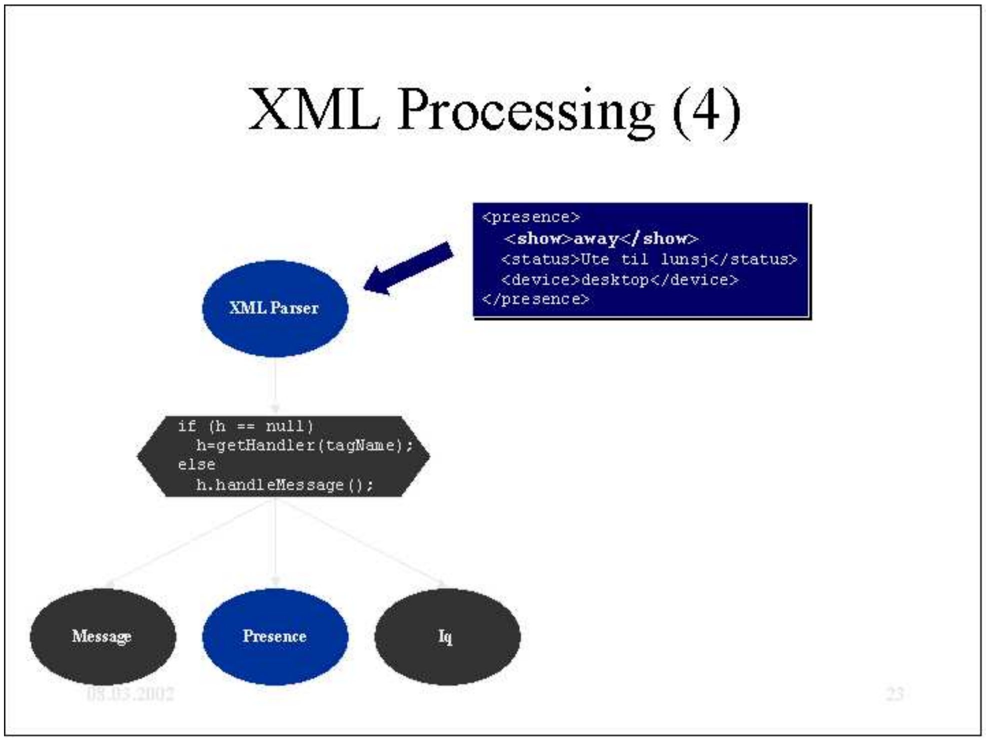
XML Processing (3)

```
<presence>
  <show>away</show>
  <status>Ute til lunsj</status>
  <device>desktop</device>
</presence>
```

```
public void getHandler(String tagName){
  Handler h;
  if (elementName.equals("message")
    h = new Message();
  else if (elementName.equals("iq")
    h = new Iq();
  else (elementName.equals("presence")
    h = new Presence();
  return h;
}
```

```
if (h == null)
  h=getHandler(tagName);
else
  h.handleMessage();
```

08.03.2002 22

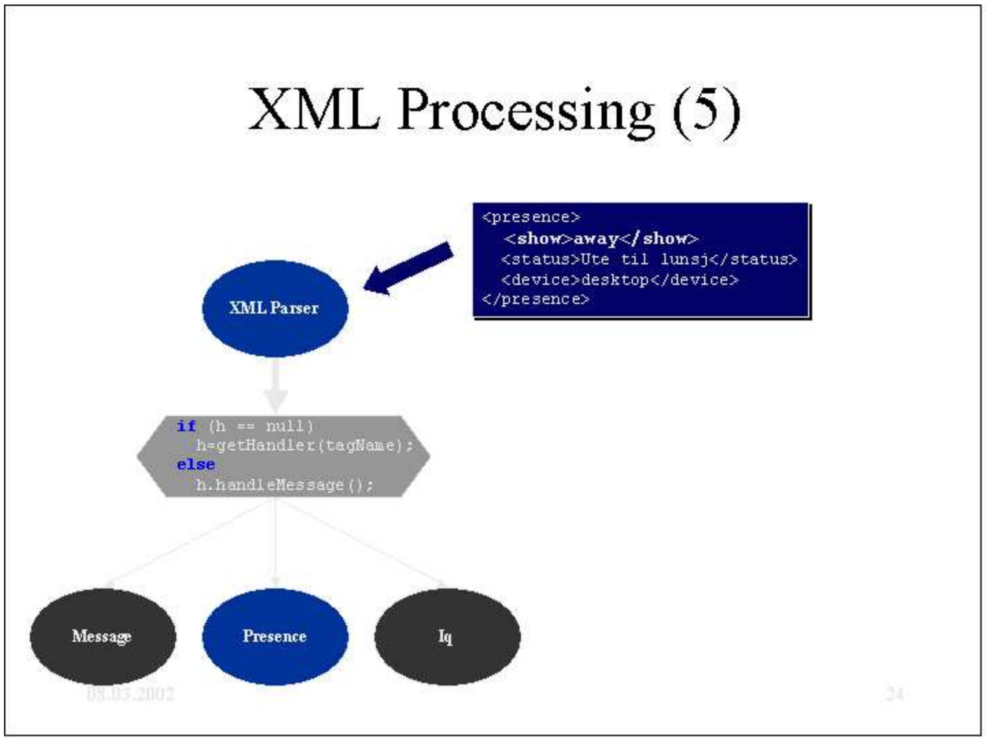


XML Processing (4)

```
<presence>
  <show>away</show>
  <status>Ute til lunsj</status>
  <device>desktop</device>
</presence>
```

```
if (h == null)
  h=getHandler(tagName);
else
  h.handleMessage();
```

08.03.2002 23

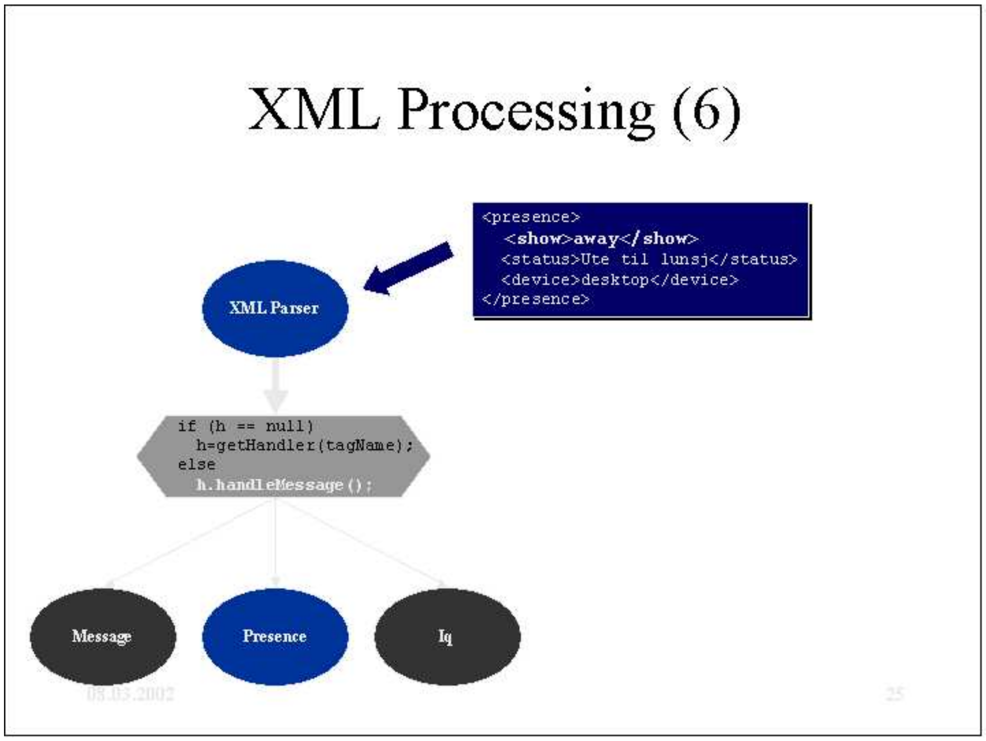


XML Processing (5)

```
<presence>
  <show>away</show>
  <status>Ute til lunsj</status>
  <device>desktop</device>
</presence>
```

```
if (h == null)
  h=getHandler(tagName);
else
  h.handleMessage();
```

08.03.2002 24

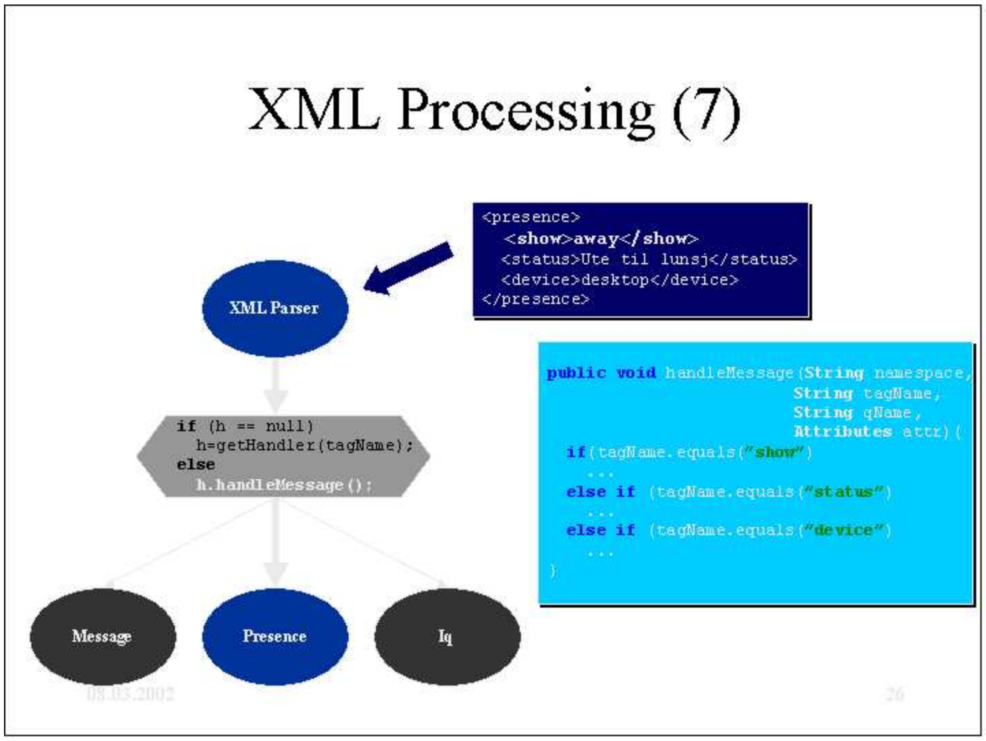


XML Processing (6)

```
<presence>
  <show>away</show>
  <status>Ute til lunsj</status>
  <device>desktop</device>
</presence>
```

```
if (h == null)
  h=getHandler(tagName);
else
  h.handleMessage();
```

08.03.2002 25



XML Processing (7)

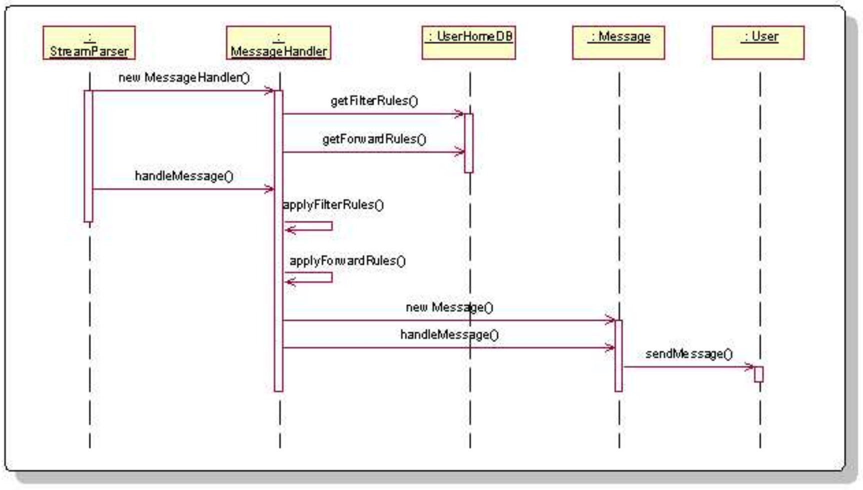
```
<presence>
  <show>away</show>
  <status>Ute til lunsj</status>
  <device>desktop</device>
</presence>
```

```
public void handleMessage(String namespace,
                          String tagName,
                          String qName,
                          Attributes attr){
  if(tagName.equals("show")
    ...
  else if (tagName.equals("status")
    ...
  else if (tagName.equals("device")
    ...
}
```

```
if (h == null)
  h=getHandler(tagName);
else
  h.handleMessage();
```

08.03.2002 26

Send Message



08.03.2002

27

Appendix C

SIGN Source Code

C.1 Mobile phone client

Commands.java

```
/*
 * Commands.java
 *
 */
package UniChat;

import javax. microedition . midlet .*;
import javax. microedition . lcdui .*;
import javax. microedition . io .*;
import java . io .*;

/**
 * @version
 */

class Commands {
    public Command exitCommand;
    public Command goCommand;
    public Command backCommand;
    public Command readCommand;
    public Command writeCommand;
    public Command sendCommand;
    public Command viewCommand;
    public Command addcontactCommand;
    public Command addCommand;
    public Command removecontactCommand;
    public Command logoffCommand;
    public Command logonCommand;
    public Command setCommand;
    public Command settingsCommand;
    public Command userCommand;
    public Command userDetailCommand;
    public Command presenceCommand;
    public Command setPresenceCommand;

    public Commands() {
        // General Commands
        goCommand = new Command("Go", Command.OK, 2);
        backCommand = new Command("Back", Command.BACK, 2);
        exitCommand = new Command("Exit", Command.EXIT, 9);

        // Contatcsscreen
        readCommand = new Command("Read", Command.SCREEN, 1);
        writeCommand = new Command("Send", Command.SCREEN, 1);
        logoffCommand = new Command("Logoff", Command.SCREEN, 6);
        logonCommand = new Command("Logon", Command.SCREEN, 5);
        settingsCommand = new Command("Settings", Command.SCREEN, 3);
        userCommand = new Command("User", Command.SCREEN,6);
        presenceCommand = new Command("Presence", Command.SCREEN, 2);

        // Write Message screen
        sendCommand = new Command("Send", Command.SCREEN, 2);

        // Message screen
        viewCommand = new Command("Read", Command.SCREEN, 2);

        // Settingsscreen
        setCommand = new Command("Set", Command.SCREEN, 2);

        // Presencescreen
        setPresenceCommand = new Command("Set", Command.SCREEN, 2);

        // Userdetailsscreen
        userDetailCommand = new Command("Details", Command.SCREEN, 2);
    }
}
```

Communication.java

```java
/*
 * Connection.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;

/**
 * @version
 */
public class Communication implements Runnable{

    private JabberComm adapter;
    private StreamConnection conn;
    private InputStream in;
    private OutputStream out;
    private StringBuffer data;
    private ByteArrayInputStream datain;
    private String outmsg;
    private int count;
    public boolean read;
    private byte sendbuffer[];
    private byte databuf[];
    // private byte receivebuffer[];
    public static final String ENVELOPE = "#\n";


    public Communication(JabberComm a) {
        adapter = a;
        read = true;
        count = 0;
        data = new StringBuffer();
        // receivebuffer = new byte[500];
        databuf = new byte[1000];
        // datain = new ByteArrayInputStream(databuf);

    }

    public void connect(){
        try{
            conn = (StreamConnection)Connector.open("socket://129.241.102.214:5222");
            // conn = (StreamConnection)Connector.open("socket://129.241.103.179:5222");
            // conn = (StreamConnection)Connector.open("socket://ravn04.idi.ntnu.no:5222");
            // conn = (StreamConnection)Connector.open("socket://hauk02.idi.ntnu.no:5222");
            in = conn.openInputStream();
            out = conn.openOutputStream();
            // datain = (ByteArrayInputStream)conn.openInputStream();

        }catch(IOException e){
            System.err.println("Connection not established.");
        }
    }

    public void disconnect(){
        try{
                conn.close();
                in.close();
                out.close();
            }catch(IOException e){
                System.err.println("Error closing connection.");
            }
        }

        public void run() {
            byte receivebuffer[] = new byte[1000];
            try{
                while(read){
                    int ch;

                    while((ch = in.read()) != -1) {
                        if (ch != '>'){
                            data.append((char)ch);
                        }
                        else{
                            data.append((char)ch);
                            adapter.receiveMsg(data.toString());
                            data = new StringBuffer();
                        }
                    }


                    //System.out.println("øFr read");
                    ch = in.read();
                    ch = in.read( receivebuffer , 0, receivebuffer .length);
                    //ch = datain.read( receivebuffer , 0, receivebuffer .length);
                    //System.out.println(" etter read");
                    //System.out.println((char)ch);
                    System.out.println(new String( receivebuffer , 0, ch));
                    //adapter.receiveMsg(new String( receivebuffer ));
                }
            }catch(Exception e) {
                System.err.println("Error reading from input.");
                System.out.println("Error reading from input.");
                disconnect();
                read = false;
            }
        }

        public void send(String str){

            try{
                sendbuffer = str.getBytes();
                out.write(sendbuffer);
                out.flush();
                System.out.println("XML --->: " + str);
            }catch(IOException e){
                System.err.println("Error sending to output.");
                disconnect();
                read = false;
            }
        }
    }
```

Contact.java

```
/*
 * Contact.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import java.util.*;

/**
 *
 * @version
 */
public class Contact {
    public Vector messages;
    public boolean status;
    public String context;
    public String show;
    public int device;
    public String name;
    public String id;
    public String subs;
    public int unread;
    private Message message;
    public boolean statuschanged;



    public Contact(String id, String username, String subscr){
        messages = new Vector();
        status = false;
        this.name = username;
        this.id = id;
        this.subs = subscr;
        this.statuschanged = false;
        this.device = 0;
        unread = 0;
    }

    public Contact(String username, boolean status){
        messages = new Vector();
        this.status = status;
        this.name = username;
        this.unread = 0;
    }

    public void addMessage(String msg){
        message = new Message(msg);
        messages.insertElementAt(message, 0);
    }
}
```

Contactsscreen.java

```
/*
 * ContactsScreen.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;
import java.lang.*;

/**
 * @version
 */
public class Contactsscreen extends javax.microedition.lcdui.List{

    private Controller controller;
    private Model model;
    private Vector contactlist;
    private Contact contact;
    private int listindex;
    private Image offlineimage;
    private Image onlinemobileimg;
    private Image onlinedesktopimg;
    private Image onlinepdaimg;
    private String s;
    private Vector names;
    private Integer integer;
    private int index;
    private int notconnected;
    private Image[] images;


    public Contactsscreen(Controller c, Model m){
        super("Contacts", javax.microedition.lcdui.List.IMPLICIT);
        this.controller = c;
        this.model = m;
        names = new Vector();
        images = new Image[4];

        try{
            images[0] = Image.createImage("/UniChat/images/offline.png");
            images[1] = Image.createImage("/UniChat/images/mobile_online.png");
            images[2] = Image.createImage("/UniChat/images/pda_online.png");
            images[3] = Image.createImage("/UniChat/images/desktop_online.png");

        }catch(IOException e){
            System.out.println("Error loading images");
        }

        addCommand(controller.commands.readCommand);
        addCommand(controller.commands.writeCommand);
        addCommand(controller.commands.presenceCommand);
        addCommand(controller.commands.logonCommand);
        addCommand(controller.commands.logoffCommand);
        addCommand(controller.commands.settingsCommand);
        addCommand(controller.commands.userCommand);
        addCommand(controller.commands.exitCommand);
        addCommand(controller.commands.userDetailCommand);

        setCommandListener( controller );
    }

    public void updateData(){

        setTitle ("␣" + model.getUserName() + "␣(" + model.getUserPresence () + ")");

        int listsize = this.size();
        for(int num = 0; num<listsize; num++){
            this.delete(0);
        }

        listsize = names.size();
        for(int p = 0; p< listsize ; p++){
            names.removeElementAt(0);
        }

        contactlist = model.getContacts();
        int size = contactlist.size();
        for(int i=0; i<size; i++){
            String displaystring = new String();
            contact = (Contact) contactlist.elementAt(i);
            displaystring = contact.name;
            // Show status
            String show = contact.show;
            if(show != null){
                displaystring = displaystring.concat("␣(" + show + ")");
            }
            // Show unread messages
            int unread = contact.unread;
            if(unread > 0){
                displaystring = displaystring.concat("␣(" + new Integer(unread).toString() + ")");
                // if (model.sound == true){
                AlertType.WARNING.playSound(controller.display);
                //}
            }
            if(contact.status == true){
                append( displaystring , images[contact.device]);
            }
            else{
                names.addElement( displaystring );
            }
        }

        if(model.showall == true){
            listsize = names.size();
            for(int i=0; i< listsize ; i++){
                append((String)names.elementAt(i), images[0]);
            }
        }
    }


    public void deleteData(){
        int listsize = this.size();
        for(int num = 0; num<listsize; num++){
            this.delete(0);
        }
    }

    public void updateContact(){
        int index = getSelectedIndex();
        s = getString(index);
        int i = s.indexOf('␣');
        if(i>0){
            s = s.substring(0, i);
        }
        model.setCurrentContact(s);
    }

    public String getContact(String name){
        int i = name.indexOf('␣');
        if(i>0){
            name = name.substring(0, i);
        }
        return name;
    }


    public void test(Contact c){
        int size = this.size();
        for(int i=0; i<size; i++){
            if(getContact(this.getString(i)).startsWith(c.name)){
                if(c.statuschanged){
                    if(c.status == true){
                        insert(0, this.getString(i), onlinemobileimg);
                        delete(i);
                    }else if(c.status == false){
                        delete(i);
                        if(model.showall == true){
                            append(this.getString(i), offlineimage);
                            delete(i);
                        }
                    }
                }else{
                    int unread = c.unread;
                    String name = c.name;
                    if(unread > 0){
                        name = name.concat("␣(" + new Integer(unread).toString() + ")");
                        AlertType.WARNING.playSound(controller.display);
                        set(i, name, onlinemobileimg);
                    }
                }
                return;
            }else{
                if(c.status == true){
                    if(c.unread > 0){
                        String name = c.name;
                        name = name.concat("␣(" + new Integer(c.unread).toString() + ")");
                        insert(0, name, onlinemobileimg);
                        AlertType.WARNING.playSound(controller.display);
                    }else{
                        insert(0, c.name, onlinemobileimg);
                    }
                }
            }
        }
    }
}
```

Controller.java

```
/*
 * Controller . java
 *
 */
package UniChat;

import javax. microedition . midlet .*;
import javax. microedition . lcdui .*;
import javax. microedition . io .*;
import java . io .*;

/**
 * @version
 */
class Controller implements CommandListener {
    private Model model;
    public UniChat midlet;
    public Commands commands;
    public Display display ;

    // Screens
    private Logonscreen logonscreen;
    private Contactsscreen contactsscreen ;
    private Sendscreen sendscreen;
    private Messagescreen messagescreen;
    private Settingsscreen settingsscreen ;
    private UserDetailsscreen userDetailsscreen ;
    private Presencescreen presencescreen ;

    public Controller (UniChat mid) {
        midlet = mid;
        model = new Model(this);
        commands = new Commands();
        display = Display . getDisplay ( midlet ) ;
        contactsscreen = new Contactsscreen ( this , model);
        sendscreen = new Sendscreen(this , model);
        messagescreen = new Messagescreen(this , model);
        settingsscreen = new Settingsscreen ( this , model);
        userDetailsscreen = new UserDetailsscreen ( this , model);
        presencescreen = new Presencescreen( this , model);
        setMainScreen() ;
    }

    public void commandAction(Command c, Displayable d) {
        if ( c == commands.exitCommand) {
            midlet . destroyApp( false ) ;
            midlet . notifyDestroyed () ;
        }
        if ( c == commands.logonCommand) {

            model.logon() ;
            /*
            if (model.logon() ){
                logonscreen = new Logonscreen( this , model);
                display . setCurrent ( logonscreen ) ;
            }else{
                display . setCurrent ( contactsscreen ) ;
            }
             */
        }
        if (c == commands.userCommand){
            logonscreen = new Logonscreen(this, model);
            display . setCurrent (logonscreen);
        }
        if (c == commands.goCommand){
            logonscreen . updateData () ;
            model.logon () ;
        }
        if (c == commands.writeCommand){
            contactsscreen . updateContact () ;
            display . setCurrent (sendscreen);
        }
        if (c == commands.sendCommand){
            sendscreen . updateData () ;
            contactsscreen . updateData () ;
            display . setCurrent ( contactsscreen );
        }
        if (c == commands.readCommand){
            contactsscreen . updateContact () ;
            messagescreen.updateData () ;
            display . setCurrent (messagescreen);
        }
        if (c == commands.backCommand && (d == messagescreen || d == sendscreen || d ==
              settingsscreen || d == logonscreen || d == userDetailsscreen || d == presencescreen
              )){
            contactsscreen . updateData () ;
            display . setCurrent ( contactsscreen );
        }
        if (c == commands.logoffCommand){
            model. logoff () ;
            display . setCurrent ( contactsscreen );
            contactsscreen . updateData () ;
        }
        if (c == commands.setCommand){
            settingsscreen . updateData () ;
            display . setCurrent ( contactsscreen );
            contactsscreen . updateData () ;
        }
        if (c == List .SELECT_COMMAND){
            contactsscreen . updateContact () ;
            messagescreen.updateData () ;
            display . setCurrent (messagescreen);
        }
        if (c == commands.settingsCommand){
            display . setCurrent ( settingsscreen );
        }
        if (c == commands.userDetailCommand){
            contactsscreen . updateContact () ;
            userDetailsscreen . updateData () ;
            display . setCurrent ( userDetailsscreen );
        }
        if (c == commands.presenceCommand){
            display . setCurrent ( presencescreen );
        }
        if (c == commands.setPresenceCommand){
            presencescreen . updateData () ;
            contactsscreen . updateData () ;
            display . setCurrent ( contactsscreen );
        }
    }

    public void setMainScreen() {
            contactsscreen .updateData();
            display.setCurrent(contactsscreen);
        }

        public void newContactEvent(){
            contactsscreen .updateData();
            display.setCurrent(contactsscreen);

        }

}
```

ElementHandler.java

```
/*
 * ElementHandler.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.util.*;
import java.io.*;
import nanoxml.*;

/**
 * @version
 */
public class ElementHandler {

    protected JabberComm com;
    protected Model model;

    public ElementHandler(JabberComm c, Model m){

        this.model = m;
        this.com = c;

    }

    public void handleElement(kXMLElement node){
    }

}
```

IqHandler.java

```
/*
 * IqHandler.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;
import java.io.*;
import java.util.*;
import nanoxml.*;

public class IqHandler extends ElementHandler {

    private StringBuffer msg;

    public IqHandler(JabberComm c, Model m){
        super(c, m);
        msg = new StringBuffer();
    }

    public void handleElement(kXMLElement node){
        String type = node.getProperty("type");
        Enumeration enum = node.enumerateChildren();
        kXMLElement element = null;
        if(enum.hasMoreElements()){
            element = (kXMLElement)(enum.nextElement());
        }
        if(type.startsWith("result")){
            if(element == null){
                System.out.println("INFO: Logon successful");
                model.setUserStatus(true);
                com.sendRosterQ();
            }else if(element.getTagName().startsWith("query")){
                com.sendLogonMsg();
            }
        }else if(type.startsWith("set")){
            if(element != null){
                Enumeration children = element.enumerateChildren();
                while(children.hasMoreElements()){
                    kXMLElement child = (kXMLElement)(children.nextElement());
                    String jid = child.getProperty("jid");
                    String name = child.getProperty("name");
                    String subs = child.getProperty("subscription");
                    Enumeration e = child.enumerateChildren();
                    while (e.hasMoreElements()){
                        kXMLElement ch = (kXMLElement)(e.nextElement());
                        // String group = ch.getContents();
                    }
                    model.newContact(jid, name, subs);
                }
                model.newContact();
                com.sendStatus();
            }
        }
    }
}
```

JabberComm.java

```java
/*
 * JabberComm.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.util.*;
import java.io.*;
import nanoxml.*;

/**
 * @version
 */
public class JabberComm {

    protected StringBuffer msg;
    public Communication communication;
    protected Model model;
    protected String name;
    protected String message;
    protected String outmessage;
    private MessageHandler messageHandler;
    private PresenceHandler presenceHandler;
    private IqHandler iqHandler;

    // Receiving messages
    public static final String TAG_CONTACTS = "contacts";
    public static final String PROP_USER = "user";
    public static final String PROP_STAT = "status";
    public static final String PROP_MSG = "msg";
    public static final String PROP_FROM = "fromUser";
    // Sending messages
    public static final String TAG_LOGOFF = "logoff";
    public static final String TAG_MSG = "msg";
    public static final String TAG_MSGBODY = "body";
    public static final String TAG_ADDCONTACT = "addcontact";
    public static final String TAG_REMOVECONTACT = "removecontact";
    public static final String PROP_TOUSER = "toUser";

    private String agent;
    // private StringBuffer msg;

    public JabberComm(Model m){
        this.model = m;
        agent = "mobile";
        msg = new StringBuffer();
        messageHandler = new MessageHandler(this, model);
        presenceHandler = new PresenceHandler(this, model);
        iqHandler = new IqHandler(this, model);

    }

    public void sendMsg(String msg){
        communication.send(msg);
    }

    public String generateRosterQ(){
        kXMLElement node = new kXMLElement();
        node.setTagName("iq");
        node.addProperty("type", "get");
        node.addProperty("id", " fullroster 30 ");
        kXMLElement child = new kXMLElement();
        child.setTagName("query");
        child.addProperty("xmlns", "jabber:iq:roster");
        node.addChild(child);
        String s = "<iq type=\"get\"><query xmlns=\"jabber:iq:roster\"/></iq>";
        // return node.toString();
        return s;
    }

    public void sendLogoffMsg(){
        String logoff = "<presence type=\"unavailable\"/>";
        sendMsg(logoff);
        logoff = "</stream:stream>";
        sendMsg(logoff);
    }

    public void sendContactsMsg(String contact, String msg){
        message = generateContactMsg(contact, msg);
        sendMsg(message);
    }

    public void parseContactMsg(kXMLElement node) {

        String from = node.getProperty(PROP_FROM);
        String time = null;
        Enumeration enum = node.enumerateChildren();
        kXMLElement element = (kXMLElement)(enum.nextElement());
        String body = getContent(element.toString());
        System.out.println("From: " + from);
        // System.out.println("time: " + time);
        System.out.println("Message: " + body);
        // model.newContactMsg(from, body, time);
    }

    public void parseContactsInfo(kXMLElement node){
        // node.getTagName();
        Enumeration enum = node.enumerateChildren();
        while(enum.hasMoreElements()){
            kXMLElement element = (kXMLElement)(enum.nextElement());
            String user = element.getProperty(PROP_USER);
            String status = element.getProperty(PROP_STAT);
            System.out.println(user + " " + status + "\n");
            // model.newContact(user, status);
        }
    }

    public void parseErrorMsg(kXMLElement node) {
        String errormsg = node.getProperty(PROP_MSG);
        System.out.println("Error: " + errormsg + "\n");
    }


    public void parseInfoMsg(kXMLElement node){
        String info = node.getProperty(PROP_MSG);
        // model.newInfo(info);
    }


    public String generateContactMsg(String toUser, String msg){
        kXMLElement node = new kXMLElement();
        node.setTagName("message");
        node.addProperty("to", toUser);
        kXMLElement child = new kXMLElement();
        child.setTagName("body");
        child.setContent(msg);
        node.addChild(child);
        //System.out.println(node.toString());
        // String xml = node.toString();
        // return node.toString();
        String s= "<message to=\"" + toUser + "\"><body>"+ msg + "</body></message>";
        return s;
    }

    public String generateSubReq(){
        String s = "<iq type=\"set\"><query xmlns=\"jabber:iq:roster\"><item jid=\"
            audun@jabber.org\" name=\"audun\" subscription=\"none\" ask=\"subscribe\"><
            group>friends</group></item></query></iq>";
        return s;
    }


    public String getContent(String node){
        int length = node.length();
        int end = length - 8;
        int size = (length - 6) - 7;
        return node.substring(6, end).trim();

    }

    public void sendRosterQ(){
        String query = generateRosterQ();
        communication.send(query);
    }

    public void sendStatus(){
        String query = "<presence type=\"available\"><show>online</show><status>online</
            status><device>mobile</device></presence>";
        communication.send(query);
    }

    public void sendPresence(String status, String context){
        String presence = "<presence>";
        if (status.length() > 0){
            presence = presence.concat("<show>" + status +"</show>");
        }
        if (context.length() > 0){
            presence = presence.concat("<status>" + context + "</status>");
        }
        presence = presence.concat("</presence>");
        communication.send(presence);
    }

    public void receiveMsg(String buf){
        try{

            int offset =0;
            //System.out.println("XML <---: " + buf);

            if (buf.charAt(1)=='?'){
```

```
        if (( offset = buf.indexOf('>', offset )) < buf.length() ){
            return;
        }else{
            offset = offset + 1;
            if ( offset + 6 > buf.length() ){
                sendInitailLogonMsg();
                return;
            }
        }
    }else if (buf.startsWith("<stream")){
        sendInitailLogonMsg();
        return;
    }else if (buf.startsWith("</stream")){
        communication.read = false;
        communication.disconnect();
        communication = null;
        return;
    }

    while( offset < buf.length() ){

        try{
            if (msg.length() > 0){
                msg.append(buf);
                buf = msg.toString();
                msg.delete (0, ( msg.length() ));
            }

            kXMLElement xmltree = new kXMLElement();
            offset += xmltree.parseString (buf, offset );
            name = null;
            name = xmltree.getTagName();

            if (name != null){
                if (name.equals("iq")){
                    // parseIqMsg(xmltree);
                    iqHandler.handleElement(xmltree);

                }else if (name.equals("presence")){
                    // parsePresenceMsg(xmltree);
                    presenceHandler.handleElement(xmltree);

                }else if (name.equals("message")){
                    // parseMessageMsg(xmltree);
                    messageHandler.handleElement(xmltree);
                }
            }
        }catch(XMLParseException e){
            if ( offset == 0){
                msg.append(buf);
                System.out.println("XML <---: "+ msg.toString());
                return;
            }else{
                msg.append(buf.substring( offset ));
                System.out.println("XML <---: "+ msg.toString());
                return;
            }
        }
    }
    }catch(Exception e){
        System.out.println(e);
    }
```

```
        }

        public void createConnection (){
            this .communication = new Communication(this);
            communication.connect();
            new Thread(communication). start () ;
            String str = "<stream:stream to=\"129.241.103.179\" xmlns=\"jabber:client\" xmlns:
                stream=\"http :// etherx . jabber . org/streams\">";
            communication.send( str );

        }

        public void sendLogonMsg() {
            // String str = "<iq type=\"set\" id=\"JCOM_5\"><query xmlns=\"jabber:iq:auth
                \"><username>"+ model.getUserName() +"</username><password>" + model.
                getPassword() + "</password><resource>alag_531</resource></query></iq>";
            String str = "<iq type=\"set\" id=\"JCOM_5\"><query xmlns=\"jabber:iq:auth\"><
                username>"+ model.getUserName() +"</username><password>"+ model.
                getPassword() +"</password><resource>mobile</resource></query></iq>";
            communication.send( str );
        }

        public void sendInitailLogonMsg() {
            // String str = "<iq type=\"get\" id=\"JCOM_40\"><query xmlns=\"jabber:iq:auth
                \"><username>"+ model.getUserName() +"</username></query></iq>";
            String str = "<iq type=\"get\" id=\"JCOM_40\"><query xmlns=\"jabber:iq:auth
                \"><username>"+ model.getUserName() +"</username></query></iq>";
            communication.send( str );
        }
}
```

Logonscreen.java

```
/*
 * LogoScreen.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;


/**
 * @version
 */
public class Logonscreen extends Form{

    private Controller controller;
    private Model model;
    private TextField nameField;
    private TextField passwordField;
    private ChoiceGroup savePasswordField;

    public Logonscreen(Controller c, Model m){
        super("Logon");
        this.controller = c;
        this.model = m;
        String[] elements = new String[1];
        elements[0] = "Save password";
        nameField = new TextField("Username: ", "", 20, TextField.ANY);
        passwordField = new TextField("Password: ", "", 20, TextField.PASSWORD);
        savePasswordField = new ChoiceGroup(null, ChoiceGroup.MULTIPLE, elements, null);
        append(nameField);
        append(passwordField);
        append(savePasswordField);

        addCommand(controller.commands.backCommand);
        addCommand(controller.commands.goCommand);
        setCommandListener(controller);
    }

    public void updateData(){
        model.setUserInfo(nameField.getString(), passwordField.getString(), savePasswordField.
            isSelected(0));
        //model.setUserInfo("nioto", "password", savePasswordField.isSelected(0));
    }

}
```

Message.java

```
/*
 * Message.java
 *
 */

package UniChat;


/**
 * @version
 */
public class Message {

    public String message;

    public Message(String msg){
        this.message = msg;
    }

}
```

MessageHandler.java

```
/*
 * MessageHandler.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;
import java.io.*;
import java.util.*;
import nanoxml.*;

/**
 * @version
 */
public class MessageHandler extends ElementHandler {

    private StringBuffer msg;

    public MessageHandler(JabberComm c, Model m){
        super(c, m);
        msg = new StringBuffer();

    }

    public void handleElement(kXMLElement node){

        String type = node.getProperty("type");
        String from = node.getProperty("from");
        String to = node.getProperty("to");
        String body = null;
        Enumeration enum = node.enumerateChildren();
        while(enum.hasMoreElements()){
            kXMLElement child = (kXMLElement)(enum.nextElement());
            if (child.getTagName().startsWith("body")) {
                body = child.getContent();
            }
        }
        model.newContactMsg(from, body);
    }
}
```

Messagescreen.java

```
/*
 * Messagescreen.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;

/**
 * @version
 */
public class Messagescreen extends javax.microedition.lcdui.List {
    private Controller controller;
    private Model model;
    private Message m;
    private Vector v;


    public Messagescreen(Controller c, Model m){
        super("Messages", javax.microedition.lcdui.List.IMPLICIT);
        this.controller = c;
        this.model = m;

        addCommand(controller.commands.backCommand);
        addCommand(controller.commands.writeCommand);
        setCommandListener(controller);
    }

    public void updateData(){

        int listsize = this.size();
        for(int num = 0; num<listsize; num++){
            this.delete(0);
        }

        v = model.getMessages();
        int size = v.size();
        if(size > model.displaynumber){
            size = model.displaynumber;
        }
        for(int i=0; i<size; i++){
            m = (Message)v.elementAt(i);
            append(m.message, null);
        }
        model.setMessageRead();
    }
}
```

Model.java

```java
/*
 * Model.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;
import java.io.*;
import java.util.*;

/**
 * @version
 */
public class Model {

    private JabberComm adapter;
    private Vector contacts;
    private Contact contact;
    private int count;
    private String infomsg;
    private String errormsg;
    private User theUser;
    private Controller controller;
    private String currentcontact;
    public int displaynumber;
    public boolean showall;
    public boolean sound;
    private RecordStore recordstore;
    private RecordEnumeration enum;
    private boolean store;
    public Hashtable deviceMap;

    public Model(Controller c) {
        contacts = new Vector();
        adapter = new JabberComm(this);
        theUser = new User();
        count = 0;
        infomsg = null;
        this.controller = c;
        displaynumber = 5;
        showall = true;
        sound = false;
        store = false;
        deviceMap = new Hashtable(3);
        deviceMap.put(new Integer(1), "Mobile");
        deviceMap.put(new Integer(2), "PDA");
        deviceMap.put(new Integer(3), "Desktop");

        /*
        try{
            recordstore = recordstore.openRecordStore("Store", true);
        }catch(RecordStoreException e){
            // return;
        }

        try{
            enum = recordstore.enumerateRecords(null, null, false);
        }catch(RecordStoreNotOpenException e){
            // return;
        }

        try{
            theUser.passWord = new String(enum.nextRecord());
            theUser.userName = new String(enum.nextRecord());
            store = true;
        }catch(InvalidRecordIDException e){
            // return;

        }catch(RecordStoreNotOpenException e){
            // return;

        }catch(RecordStoreException e){
            // return;
        }
        */

        theUser.userName = "jill";
        theUser.passWord = "jill";
    }


    public void newContact(String id, String name, String subs){
        if (findContact(id) == null){
            Contact c = new Contact(id, name, subs);
            contacts.addElement(c);
            count++;
        }
    }

    public void newContact(){
        controller.setMainScreen();
    }

    public void newContactMsg(String user, String body){
        contact = null;
        contact = findContact(user);
        if (contact != null){
            contact.addMessage(body);
            contact.unread++;
            controller.newContactEvent();
        }
    }


    private Contact findContact(String id){
        for(int i=0; i<contacts.size(); i++){
            if ((((Contact)contacts.elementAt(i)).id).equals(id)){
                return (Contact)contacts.elementAt(i);
            }
        }
        return null;
    }

    private Contact findContactByName(String name){
        for(int i=0; i<contacts.size(); i++){
            if ((((Contact)contacts.elementAt(i)).name).equals(name)){
                return (Contact)contacts.elementAt(i);
            }
        }
        return null;
    }

    public void setUserInfo(String userName, String passWord, boolean save) {
        theUser.userName = userName;
        theUser.passWord = passWord;

        ///*
        if(save){

            try{
                enum = recordstore.enumerateRecords(null, null, false);
            }catch(RecordStoreNotOpenException e){
                return;
            }

            try{
                recordstore.deleteRecord((enum.nextRecordId()));
                recordstore.deleteRecord((enum.nextRecordId()));
            }catch(InvalidRecordIDException e){
                // return;

            }catch(RecordStoreNotOpenException e){
                // return;

            }catch(RecordStoreException e){
                // return;
            }

            try{
                recordstore.addRecord(userName.getBytes(), 0, userName.length());
                recordstore.addRecord(passWord.getBytes(), 0, passWord.length());
            }catch(RecordStoreNotOpenException e){

            }catch(RecordStoreException e){

            }
            store = true;
        }
        //*/

    }

    public void setUserStatus(boolean b){
        theUser.status = b;
        if(b == false){
            setUserPresence("Offline");
        }else{
            setUserPresence("Online");
        }
    }

    public void setUserPresence(String presence){
        theUser.presence = presence;
    }

    public String getUserPresence(){
        return theUser.presence;
    }

    public String getUserStatus(){
            String s;
            if (theUser.status == true){
                s = "Online";
                setUserPresence("Online");
            }else{
                s = "Offline";
                setUserPresence("Offline");
            }
            return s;
        }

        public String getUserName(){
            return theUser.userName;
        }

        public String getPassword(){
            return theUser.passWord;
        }

        public boolean logon() {
            // if (true){
                adapter.createConnection();
                return false;
            //} else{
              //    return true;
           // }
        }

        public void logoff() {
            removeAllContacts();
            adapter.sendLogoffMsg();
            setUserStatus(false);

        }

        public Vector getContacts() {
            return contacts;
        }


        public void setCurrentContact(String name) {
            Contact c = findContactByName(name);
            currentcontact = c.id;
        }

        public String getShow(){
            Contact c = findContact(currentcontact);
            return c.show;
        }

        public Contact getCurrentContact(){
            Contact c = findContact(currentcontact);
            return c;
        }

        public void writtenMsg(String msg){
            adapter.sendContactsMsg(currentcontact, msg);
        }

        public Vector getMessages(){
            Contact c = findContact(currentcontact);
            return c.messages;
```

```
    }

    public void setMessageRead(){
        Contact c = findContact( currentcontact );
        c.unread = 0;
    }

    private void removeAllContacts(){
        int size = contacts.size();
        for(int i=0; i<size; i++){
            contacts.removeElementAt(0);
            count = 0;
        }
    }

    public void newPresenceMsg(String from, String to, String show, String status, String
          device){
        Contact contact = findContact(from);
        if(contact != null){
            if(status.startsWith("Online")){
                contact.status = true;
                contact.show = null;
            }

            if(device != null && device.startsWith("mobile")){
                contact.device = 1;
            }else if(device != null && device.startsWith("pda")){
                contact.device = 2;
            }else if(device != null && device.startsWith("desktop")){
                contact.device = 3;
            }else{
                contact.device = 3;
            }

            if(show != null){
                contact.show = show;
            }
            if(status != null){
                contact.context = status;
            }

            controller.newContactEvent();
        }

    }

    public void contactOffline(String id){
        Contact contact = findContact(id);
        if(contact != null){
            contact.status = false;
            contact.show = null;
            contact.context = null;
            contact.device = -1;
            controller.newContactEvent();
        }
    }

    public void sendPresence(String status, String context){
        adapter.sendPresence(status, context);
        if(status.length() > 0){
            setUserPresence(status);
        }
    }
}
```

PresenceHandler.java

```
/*
 * PresenceHandler.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.microedition.rms.*;
import java.io.*;
import java.util.*;
import nanoxml.*;

/**
 * @version
 */
public class PresenceHandler extends ElementHandler {

    private StringBuffer msg;

    public PresenceHandler(JabberComm c, Model m){
        super(c, m);
        msg = new StringBuffer();

    }

    public void handleElement(kXMLElement node){

        String type = node.getProperty("type");
        Enumeration enum = node.enumerateChildren();

        if(type != null && type.startsWith("unavailable")){
            String from = node.getProperty("from");
            model.contactOffline(from);
        }else if(type != null && type.startsWith("subscribe")){
            String from = node.getProperty("from");
            com.communication.send("<presence from=\"nioto@jabber.org\" to=\"audun@jabber.org
                \" type=\"subscribed\"/>");
        }else{
            String show = null;
            String device = null;
            String status = null;
            String from = node.getProperty("from");
            String to = node.getProperty("to");
            while(enum.hasMoreElements()){
                kXMLElement child = (kXMLElement)(enum.nextElement());
                if(child.getTagName().startsWith("show")){
                    show = child.getContent();
                }else if(child.getTagName().startsWith("status")){
                    status = child.getContent();
                }else if(child.getTagName().startsWith("device")){
                    device = child.getContent();
                }
            }
            model.newPresenceMsg(from, to, show, status, device);
        }
    }
}
```

Presencescreen.java

```
/*
 * Presencescreen.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;

/**
 * @version
 */
public class Presencescreen extends Form{

    private Controller controller;
    private Model model;
    private ChoiceGroup presenceChoice;
    private ChoiceGroup autoReplyBox;
    private ChoiceGroup opaqueBox;
    private TextField sizeField;
    private TextField autoReplyField;
    private TextField contextField;
    private String[] displayTypes;
    private String[] elements;

    public Presencescreen(Controller c, Model m){
        super("Presence");
        this.controller = c;
        this.model = m;

        elements = new String[1];
        elements[0] = "Opaque";

        displayTypes = new String[3];
        displayTypes[0] = "Online";
        displayTypes[1] = "Do not disturb";
        displayTypes[2] = "Free for chat";
        presenceChoice = new ChoiceGroup("Presence", Choice.EXCLUSIVE, displayTypes, null);


        contextField = new TextField("Context", "", 20, TextField.ANY);


        sizeField = new TextField("Max size", "", 3, TextField.NUMERIC);


        opaqueBox = new ChoiceGroup(null, ChoiceGroup.MULTIPLE, elements, null);


        elements[0] = "Auto reply";
        autoReplyBox = new ChoiceGroup(null, ChoiceGroup.MULTIPLE, elements, null);


        autoReplyField = new TextField("", "", 20, TextField.ANY);


        this.append(presenceChoice);
        this.append(contextField);
        this.append( sizeField );
        this.append(opaqueBox);
        this.append(autoReplyBox);
        this.append(autoReplyField );

        addCommand(controller.commands.backCommand);
        addCommand(controller.commands.setPresenceCommand);
        setCommandListener( controller );
    }


    public void updateData(){
        String context = new String( contextField.getString() );
        String status = null;
        int index = presenceChoice.getSelectedIndex();
        if (index == 0){
            status = "Online";
        }else if (index == 1){
            status = "dnd";
        }else if (index == 2){
            status = "chat";
        }

        if ( context.length() > 0 || index != -1){
            model.sendPresence( status , context );
        }

        String size = sizeField.getString();
        if ( size.length() > 0){
            // model.sendMaxSizePref( size );
        }

        if (autoReplyBox.isSelected(0)){
            String reply = autoReplyField.getString();
            // model.sendAutoReplyPref( reply );
        }

        if (opaqueBox.isSelected(0)){
            // model.sendOpaquePref();
        }

    }

}
```

Sendscreen.java

```
/*
 * Sendscreen.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;
/**
 * @version
 */
public class Sendscreen extends Form {

    private Controller controller;
    private Model model;
    private TextField messagefield;

    public Sendscreen(Controller c, Model m){
        super("Message");
        this.controller = c;
        this.model = m;
        messagefield = new TextField("", "", 100, TextField.ANY);
        append(messagefield);

        addCommand(controller.commands.backCommand);
        addCommand(controller.commands.sendCommand);
        setCommandListener(controller);
    }

    public void updateData(){
        model.writtenMsg(messagefield.getString());
    }
}
```

Settingsscreen.java

```
/*
 * Settingsscreen.java
 *
 */

package UniChat;

import javax.microedition.io.*;
import javax.microedition.lcdui.*;


/**
 * @version
 */
public class Settingsscreen extends Form{

    private Controller controller;
    private Model model;
    private ChoiceGroup sortChoice;
    private ChoiceGroup soundChoice;
    private TextField messages;
    private String [] displayTypes;
    private String [] soundTypes;


    public Settingsscreen (Controller c, Model m){
        super("Display settings");
        this.controller = c;
        this.model = m;

        displayTypes = new String [2];
        displayTypes [0] = "Show All";
        displayTypes [1]= "Show Online";
        sortChoice = new ChoiceGroup("Display Types", Choice.EXCLUSIVE, displayTypes, null);

        soundTypes = new String [2];
        soundTypes[1] = "Sound ON";
        soundTypes[0] = "Sound OFF";
        soundChoice = new ChoiceGroup("Sound", Choice.EXCLUSIVE, soundTypes, null);

        messages = new TextField("# Messages", "", 2, TextField.NUMERIC);

        this.append(messages);
        this.append(sortChoice);
        this.append(soundChoice);

        addCommand(controller.commands.backCommand);
        addCommand(controller.commands.setCommand);
        setCommandListener(controller);
    }

    public void updateData(){
        Integer integer = new Integer(Integer.parseInt(messages.getString()));
        model.displaynumber = integer.intValue();

        int index = sortChoice.getSelectedIndex();
        if(index == 0){
            model.showall = true;
        }
        else{
                model.showall = false ;
            }

            index = soundChoice. getSelectedIndex () ;
            if (index == 0) {
                model.sound = false ;
            }
            else {
                model.sound = false ;
            }

        }

    }
```

UniChat.java

```
/*
 * MobileIMS.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;


public class UniChat extends javax.microedition.midlet.MIDlet {
    public Controller controller;

    public UniChat() {
        controller = new Controller(this);
    }

    public void startApp() {
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```

User.java

```
/*
 * User.java
 *
 */

package UniChat;

import javax.microedition.midlet.*;

/**
 * @version
 */
public class User {
    public String userName;
    public String passWord;
    public boolean status;
    public String presence = "Offline";

    public User(){
    }
}
```

UserDetailsscreen.java

```
/*
 * UserDetails.java
 */

package UniChat;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;
/**
 * @version
 */
public class UserDetailsscreen extends javax.microedition.lcdui.List{

    private Controller controller;
    private Model model;

    public UserDetailsscreen(Controller c, Model m){
        super("User", javax.microedition.lcdui.List.IMPLICIT);
        this.controller = c;
        this.model = m;

        addCommand(controller.commands.backCommand);
        addCommand(controller.commands.writeCommand);
        setCommandListener(controller);
    }

    public void updateData(){

        Contact c = model.getCurrentContact();

        setTitle("User - " + c.name);

        int listsize = this.size();
        for(int num = 0; num<listsize; num++){
            this.delete(0);
        }

        String text = "Device: ";
        if(c.device < 0){
            text = text.concat((String)model.deviceMap.get(new Integer(c.device)));
        }
        append(text, null);

        text = "Status: ";
        if(c.show != null){
            text = text.concat(c.show);
        }
        append(text, null);

        text = "Context: ";
        if(c.context != null){
        text = text.concat(c.context);
        }
        append(text, null);
    }
}
```

C.2 PDA client

Contact.java

```
package jabber;

import java.util.*;

public class Contact {

  public final static int MOBILE = 0;
  public final static int DESKTOP = 1;
  public final static int PDA = 2;

  public final static int ONLINE = 0;
  public final static int OFFLINE = 1;
  public final static int DND = 2;
  public final static int AWAY = 3;
  public final static int XA = 4;

  private String username;
  private String jid;
  private String context = "";
  int device;
  int show;
  Vector unreadMessages;

  public Contact(String jid, String name){
    this.username = name;
    this.jid = jid;
    show = OFFLINE;
    device = DESKTOP;
    unreadMessages = new Vector();
  }

  public void setPresence(int show, String status, int device){
    this.show = show;
    this.context = status;
    this.device = device;
  }
  public void addMessage(String msg){
    unreadMessages.add(msg);
  }
  public Enumeration getUnreadMessages(){
    return unreadMessages.elements();
  }
  public void removeUnreadMessages(){
    unreadMessages.removeAllElements();
  }
  public String getName(){
    return username;
  }
  public String getJid(){
    return jid;
  }
  public String getContext(){
    return context;
  }
}
```

ContactPanel.java

```
package jabber;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ContactPanel extends Panel implements ActionListener {

  private final static int TOP = 22;
  private final static int ROW_HEIGHT = 20;
  private final static int ICON_MARGIN = 1;
  private final static int NAME_MARGIN = ICON_MARGIN + 18;

  JabberFrame frame;
  static Font bigFont;
  static Font smallFont;
  Vector rows = new Vector();
  Row selectedRow = null;
  MenuItem popupSend;


  final static Image header = Toolkit.getDefaultToolkit().getImage("border.gif");
  Image icons[][] = new Image[3][3];

  public ContactPanel(JabberFrame frame) {
    this.frame = frame;
    popupSend = new MenuItem("Read/Send message");
    popupSend.addActionListener(this);

    MouseListener ml = new MouseAdapter() {
            public void mouseReleased(MouseEvent e){
              Enumeration enum = rows.elements();
              while (enum.hasMoreElements()) {
                Row row = (Row)enum.nextElement();
                if (row.rect.contains(e.getX(), e.getY())){
                  selectedRow = row;
                  showPopupMenu(row, e);
                }
              }
            }
    };
    addMouseListener(ml);

    bigFont = new Font("times", Font.BOLD, 14);
    smallFont = new Font("times", Font.PLAIN, 12);

    icons[Contact.DESKTOP][Contact.ONLINE] = Toolkit.getDefaultToolkit().getImage("
         desktop_online.gif");
    icons[Contact.DESKTOP][Contact.OFFLINE] = Toolkit.getDefaultToolkit().getImage("
         desktop_offline.gif");
    icons[Contact.DESKTOP][Contact.DND] = Toolkit.getDefaultToolkit().getImage("desktop_dnd.gif
         ");
    icons[Contact.MOBILE][Contact.ONLINE] = Toolkit.getDefaultToolkit().getImage("mobile_online.
         gif");
    icons[Contact.MOBILE][Contact.OFFLINE] = Toolkit.getDefaultToolkit().getImage("
         mobile_offline.gif");
    icons[Contact.MOBILE][Contact.DND] = Toolkit.getDefaultToolkit().getImage("mobile_dnd.gif");
    icons[Contact.PDA][Contact.ONLINE] = Toolkit.getDefaultToolkit().getImage("pda_online.gif");
    icons[Contact.PDA][Contact.OFFLINE] = Toolkit.getDefaultToolkit().getImage("pda_offline.gif
         ");
    icons[Contact.PDA][Contact.DND] = Toolkit.getDefaultToolkit().getImage("pda_dnd.gif");
    MediaTracker tracker = new MediaTracker(this);
    int id=0;
    for(int r=0; r<3; r++){
      for(int c=0; c<3; c++){
        tracker.addImage(icons[r][c], id);
        id++;
      }
    }
    try{
      tracker.waitForAll();
    } catch (InterruptedException ie){
    }
  }

  private void showPopupMenu(Row row, MouseEvent e){
    PopupMenu pm = new PopupMenu();
    add(pm);
    MenuItem jid = new MenuItem(row.contact.getJid());
    jid.setEnabled(false);
    jid.setFont(new Font("times", Font.BOLD, 12));
    pm.add(jid);
    pm.add("-");
    pm.add(popupSend);
    pm.add(new MenuItem("Remove " + row.contact.getName()));
    pm.show(this, e.getX(), e.getY());
  }

  public void addContact(Contact c){
    rows.insertElementAt(new Row(c), 0);
    validate();
    repaint();
  }

  public void clearContacts(){
    selectedRow = null;
    rows.removeAllElements();
    validate();
    repaint();
  }

  public void paint(Graphics g){
    int y = TOP;
    int iconHeight = 10; // mobile_online.getHeight(null)-2;
    Image buffer = createImage(getSize().width, getSize().height);
    Graphics bg = buffer.getGraphics();
    FontMetrics fm = bg.getFontMetrics(bigFont);

    Enumeration enum = rows.elements();
    while (enum.hasMoreElements()) {
      Row row = (Row)enum.nextElement();
      Image image = icons[row.contact.device][row.contact.show];
      bg.drawImage(image, ICON_MARGIN, y-image.getHeight(null)+3, null);
      bg.setFont(bigFont);
      bg.setColor(Color.blue);
      bg.drawString(row.displayString, NAME_MARGIN, y);
      if(row.contact.unreadMessages.size() > 0){
        bg.setFont(smallFont);
        bg.setColor(Color.black);
        bg.drawString("(" + String.valueOf(row.contact.unreadMessages.size()) + ")",
              NAME_MARGIN + fm.stringWidth(row.displayString)+2, y);
      }
```

```
            row.rect = new Rectangle(NAME_MARGIN-2,y-fm.getMaxAscent(),fm.stringWidth(row.
                    displayString)+5, fm.getMaxAscent()+3);
            y += ROW_HEIGHT;
        }
        g.drawImage(buffer, 0, 0, null);
        bg.dispose();
        buffer.flush();
    }

    public void actionPerformed(ActionEvent e){
        Object o = e.getSource();
        if(o == popupSend){
            frame.setMessagePanel(selectedRow.contact);
        }
    }

    private class Row {

        public Row(Contact c){
            contact = c;
            displayString = c.getName();
        }
        public Contact contact;
        public Rectangle rect;
        public Image icon;
        public String displayString;
    }
}
```

ContextDialog.java

```
package jabber;

import java.awt.*;
import java.awt.event.*;

public class ContextDialog extends Dialog implements ActionListener{

  TextField tfContext = new TextField("", 10);
  Label labelContext = new Label("Enter context", Label.LEFT);
  Button ok = new Button("OK");
  Button cancel = new Button("Cancel");
  boolean okPressed = false;


  public ContextDialog(JabberFrame frame) {
    super(frame, true);

    ok.addActionListener(this);
    cancel.addActionListener(this);

    Panel p = new Panel();
    p.add(labelContext);
    p.add(tfContext);
    p.add(ok);
    p.add(cancel);
    add(p);
    setSize(100, 150);
  }

  public void actionPerformed(ActionEvent e){
    if(e.getSource() == ok){
      okPressed = true;
      dispose();
    } else if(e.getSource() == cancel){
      dispose();
    }
  }
}
```

ElementHandler.java

```
package jabber;

import nanoxml.*;

public abstract class ElementHandler {

   protected JabberFrame gui;
   protected JabberComm comm;

   public ElementHandler(JabberFrame gui, JabberComm comm) {
      this.gui = gui;
      this.comm = comm;
   }

   public abstract void handleElement(XMLElement xml);

   public static String trimJid(String jid){
      int endIndex = jid.indexOf('/');
      if (endIndex > 0)
         return jid.substring(0, endIndex);
      else
         return jid;
   }

}
```

HeaderPanel.java

```java
package jabber;

import java.awt.*;
import java.awt.image.*;

public class HeaderPanel extends Panel {

  private Image border;
  private String username = null;
  private String status = "Offline";
  private static Font nameFont = new Font("times", Font.BOLD, 14);
  private static Font statusFont = new Font("times", Font.PLAIN, 12);

  public HeaderPanel() {
    border = Toolkit.getDefaultToolkit().getImage("border.gif");
    MediaTracker mt = new MediaTracker(this);
    mt.addImage(border, 0);
    try{
      mt.waitForID(0);
    } catch (InterruptedException ie){}
  }

  public void setUsername(String username){
    this.username = username;
  }

  public void setStatus(String status){
    this.status = status;
    repaint();
  }

  public void paint(Graphics g){
    int fontBaseline = 15;
    if(username != null){
      int statusX = g.getFontMetrics(nameFont).stringWidth(username);
      Dimension dim = getSize();
      g.setFont(nameFont);
      g.drawString(username, 2, fontBaseline);
      g.setFont(statusFont);
      g.drawString(" (" + status + ")", statusX, fontBaseline);
    } else{
      g.setFont(nameFont);
      g.drawString("<No User>", 2, fontBaseline);
    }
    g.drawImage(border, 0, 22, null);
  }
}
```

IqHandler.java

```java
package jabber;

import java.util.*;

import nanoxml.*;

 public class IqHandler extends ElementHandler {

  private final static String ONLINE_MESSAGE = "<presence><status>Online</status></
       presence>";

  public IqHandler(JabberFrame gui, JabberComm comm) {
    super(gui, comm);
  }

  public void handleElement(XMLElement xml) {
     String type;
     String id;

     type = xml.getProperty("type");
     id = xml.getProperty("id");

     if (type.equalsIgnoreCase("result") || type.equalsIgnoreCase("set")){
       if (id.equalsIgnoreCase("logon")){
         gui.println("Logon OK");
         // gui.connected(true);
         // gui. setTitle ();
         comm.send(ONLINE_MESSAGE);
         comm.send(comm.ROSTER);
       } else if (id.equalsIgnoreCase("roster")){
         XMLElement child = (XMLElement)xml.getChildren().elementAt(0);
         for(Enumeration e = child.enumerateChildren(); e.hasMoreElements(); ){
           XMLElement contact = (XMLElement)e.nextElement();
           String nick = contact.getProperty("name");
           String jid = contact.getProperty("jid");
           gui.addContact(new Contact(jid, nick));
         }
       }
     } else if (type.equalsIgnoreCase("error")){
       gui.setConnected(false, "Logon failed");
     } else {
       gui.println("Handle me:\n" + xml.toString());
     }
  }
}
```

Jabber.java

```
import jabber.*;

public class Jabber {

  public static void main(String[] args) {
    new JabberFrame();
  }
}
```

JabberComm.java

```
package jabber;

import java.io.*;
import java.net.*;
import nanoxml.*;

public class JabberComm implements Runnable {

  private final static int PORT = 5222;
  private final static String DEVICE = "pda";

  private JabberUser user;
  private JabberFrame frame;
  private Socket socket;
  private InputStream in;
  private OutputStream out;
  private boolean doRead = true;
  private ElementHandler iqHandler;
  private ElementHandler presenceHandler;
  private ElementHandler messageHandler;

  final static String ROSTER = "<iq id=\"roster\" type=\"get\">" +
                               "<query xmlns=\"jabber:iq:roster\"/>" +
                               "</iq>";
  final static String UNAVAILABLE = "<presence type=\"unavailable\"/>";
  final static String DISCONNECT = "</stream:stream>";

  public JabberComm(JabberFrame frame, JabberUser user) {
    this.frame = frame;
    this.user = user;
    iqHandler = new IqHandler(this.frame, this);
    presenceHandler = new PresenceHandler(this.frame, this);
    messageHandler = new MessageHandler(this.frame, this);
  }

  public void connect(String adr) throws Exception {
    try {
      socket = new Socket(adr, PORT);
      in = socket.getInputStream();
      out = socket.getOutputStream();
      (new Thread(this)).start();
      send(getConnectMessage(adr));
    } catch (Exception e){
      doRead = false;
      throw e;
    }
  }

  public void disconnect(){
    send(UNAVAILABLE);
    send(DISCONNECT);
    try{
      Thread.sleep(500);
    } catch (InterruptedException ie){}
    doRead = false;
  }

  public void processMessage(String s){
    System.out.println(s);
    int offset = 0;
```

```
        int pos;
        XMLElement m;
        if(s.charAt(1) == '?'){ // begin document
           offset = s.indexOf('>', offset);
           offset +=1;
           if(offset + 6 >= s.length())
              return;
        }
        String temp = s.substring(offset+1, offset+7);
        if(temp.equals("stream")){
           offset = s.indexOf('>', offset);
           send(getLogonMessage(user.username, user.password));
           frame.setConnected(true, "Connected to server");
           return;
        }

        while(offset < s.length()){
           try{
              m = new XMLElement();
              offset += m.parseString(s, offset);
              String elementName = m.getTagName();
              if(elementName.equals("iq")){
                 iqHandler.handleElement(m);
              } else if(elementName.equals("presence")){
                 presenceHandler.handleElement(m);
              } else if(elementName.equalsIgnoreCase("message")){
                 messageHandler.handleElement(m);
              } else{
                 frame.println(m.toString());
              }
           }catch(XMLParseException xmle){
              frame.println(xmle.getMessage());
              break;
           }
        }
     }

     public void send(String s){
        try {
           out.write(s.getBytes());
           out.flush();
        } catch (IOException ioe){
           doRead = false;
           frame.println("Disconnected");
        }
     }

     public void sendPresence(String context, int show){
        if(context.equals("hide")){
           send("<presence type=\"unavailable\"/>");
           return;
        }
        StringBuffer buf = new StringBuffer();
        buf.append("<presence from=\"" + user.getJid() + "\" type=\"available\"><show>");
        switch(show){
        case Contact.ONLINE:
           buf.append("online");
           break;
        case Contact.AWAY:
           buf.append("away");
           break;
        case Contact.DND:
            buf.append("dnd");
            break;
        }
        buf.append("</show>");
        buf.append("<status>" + context + "</status>");
        buf.append("<device>" + DEVICE + "</device>");
        buf.append("</presence>");
        send(buf.toString());
    }

    public void sendMessage(String toUserJid, String text){
        String msg = "<message id=\"12\" to=\"" + toUserJid + "\">" +
                     "<body>" + text + "</body>" +
                     "</message>";
        send(msg);
    }

    private String getConnectMessage(String server){
        String temp = "<stream:stream to=\"" + server + "\" " +
                  "xmlns=\"jabber:client\" " +
                  "xmlns:stream=\"http://etherx.jabber.org/streams\">";
        return temp;
    }

    private String getLogonMessage(String username, String pwd){
        String temp = "<iq type=\"set\" id=\"logon\"><query xmlns=\"jabber:iq:auth\">" +
                 "<username>" + username + "</username>" +
                 "<password>" + pwd + "</password>" +
                 "<resource>pda</resource></query></iq>";
        return temp;
    }

    public void run(){
        byte[] buf = new byte[1000];
        int bytesRead = 0;

        while(doRead){
            try {
                bytesRead = in.read(buf);
                if (!doRead)
                    return;
                if (bytesRead == -1){
                    frame.setConnected(false, "Connection closed by peer");
                    return;
                }
                processMessage(new String(buf).substring(0, bytesRead));
            } catch (IOException ioe){
                frame.setConnected(false, "Connection closed" + ioe.getMessage());
                doRead = false;
            }
        }
    }
}
```

JabberFrame.java

```
package jabber;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class JabberFrame extends Frame implements ActionListener {

  private MenuBar menubar;
  private MenuItem exit;
  private MenuItem connect;
  private MenuItem disconnect;
  private MenuItem setup;

  private Menu presence;
  private MenuItem online;
  private MenuItem busy;
  private MenuItem away;
  private MenuItem presenceShow;
  private MenuItem presenceHide;

  private MenuItem presenceContext;

  TextArea output;
  ContactPanel contactPanel;
  MessagePanel sendPanel;
  HeaderPanel header;
  JabberComm comm;
  private Hashtable contacts;

  JabberUser user = null;
  String server = null;
  boolean connected = false;

  public JabberFrame() {
    super("Jabber");
    contacts = new Hashtable();
    initGUI();
    show();

    // Closable window
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        if (connected)
          comm.disconnect();
        System.exit(0);
      }
    });
  }

  private void initGUI(){
    setSize(240, 320);
    setResizable(false);

    menubar = new MenuBar();
    Menu file = new Menu("File");
    connect = new MenuItem("Connect");
    connect.addActionListener(this);
    file.add(connect);
    setup = new MenuItem("Setup");
```

```
      setup.addActionListener(this);
      file.add(setup);
      file.add(new MenuItem("-"));
      exit = new MenuItem("Exit");
      exit.addActionListener(this);
      file.add(exit);
      menubar.add(file);

      presence = new Menu("Presence");
      online = new MenuItem("Online");
      online.addActionListener(this);
      presence.add(online);
      busy = new MenuItem("Do not disturb");
      busy.addActionListener(this);
      presence.add(busy);
      away = new MenuItem("Away");
      away.addActionListener(this);
      presence.add(away);
      presence.add(new MenuItem("-"));
      presenceContext = new MenuItem("Context...");
      presence.add(presenceContext);
      presenceContext.addActionListener(this);
      presence.add(new MenuItem("-"));
      presenceShow = new MenuItem("Show presence");
      presenceShow.addActionListener(this);
      presenceHide = new MenuItem("Hide presence");
      presenceHide.addActionListener(this);
      presence.add(presenceHide);
      menubar.add(presence);

      setMenuBar(menubar);

      header = new HeaderPanel();
      header.setSize(100, 30);
      contactPanel = new ContactPanel(this);
      contactPanel.setSize(100, 100);
      sendPanel = new MessagePanel(this);
      output = new TextArea("", 4, 1);
      output.setEditable(false);
      output.setBackground(Color.white);

      setLayout(new GridBagLayout());
      addComponents(contactPanel);
   }

   private void addComponents(Panel panel){
      removeAll();
      setLayout(new GridBagLayout());
      GridBagConstraints gc = new GridBagConstraints();
      gc.fill = GridBagConstraints.BOTH;
      gc.gridx = 0;
      gc.gridy = 0;
      gc.gridwidth = 1;
      gc.gridheight = 1;
      gc.weightx = 100;

      if(panel == contactPanel)
         gc.weighty = 1;
      else
         gc.weighty = 4;
      add(header, gc);

```

```
    gc.gridy = 1;
    gc.gridheight = 2;
    gc.weighty = 10;
    add(panel, gc);

    gc.gridy = 3;
    gc.gridheight = 1;
    gc.weighty = 0;
    add(output, gc);
    doLayout();
    header.invalidate();
    header.repaint();
  }

  public void println(String s){
    output.append(s +"\n");
  }

  public void setConnected(boolean connected, String message){
    this.connected = connected;
    if(connected){
      header.setStatus("Online");
    } else {
      header.setStatus("Offline");
      println(message);
    }
  }

  public void addContact(Contact c){
    contacts.put(c.getJid(), c);
    contactPanel.addContact(c);
  }

  public void clearContacts(){
    contacts.clear();
    contactPanel.clearContacts();
  }

  public void setPresence(String jid, int show, String status, int device){
    Contact c = (Contact) contacts.get(jid);
    if(c == null)
        return;
    c.setPresence(show, status, device);
    contactPanel.repaint(); /* @todo repaint region only */
  }

  public void newMessage(String jid, String header, String body){
    Contact c = (Contact) contacts.get(jid);
    if(c != null){
      if(sendPanel.isShowing() && c == sendPanel.contact){
        sendPanel.displayMessage(c.getName(), header+body);
      } else {
        c.addMessage(header + body);
        contactPanel.repaint(); /* @todo repaint region only */
      }
    }
  }

  public void setContactPanel(){
    addComponents(contactPanel);
    contactPanel.invalidate();
    contactPanel.repaint();
    }

    public void setMessagePanel(Contact c){
      addComponents(sendPanel);
      sendPanel.doLayout();
      sendPanel.setContact(c);
      //sendPanel.invalidate();
      //sendPanel.repaint();
    }

    private boolean showSetupDialog(){
      SetupDialog dlg = new SetupDialog(this);
      dlg.show();
      if(dlg.okPressed){
        String name = dlg.tfUsername.getText();
        header.setUsername(name);
        server = dlg.tfServer.getText();
        user = new JabberUser(name, dlg.tfPassword.getText(), name + "@" + server);
        header.repaint();
        return true;
      }
      return false;
    }

    public JabberUser getUser(){
      return user;
    }

    public void connect(){
      clearContacts();
      try {
        if(user == null){
          if(!showSetupDialog())
            return;
        }
        println("Connecting ...");
        comm = new JabberComm(this, user);
        comm.connect(server);
      }catch(Exception e){
        output.setText("Connection failed:\n" + e.getMessage());
      }
    }

    public void actionPerformed(ActionEvent e){
      Object source = e.getSource();
      if(source == exit){
        if(connected)
          comm.disconnect();
        System.exit(0);
      } else if(source == connect){
        connect();
      } else if(source == setup){
        showSetupDialog();
      } else if(source == online){
        user.presence = Contact.ONLINE;
        if(comm != null){
          header.setStatus("Online");
          comm.sendPresence("Online", Contact.ONLINE);
        }
      } else if(source == away){
        user.presence = Contact.AWAY;
        if(comm != null){
            header. setStatus ("Away");
            comm.sendPresence("Away", Contact.AWAY);
          }
        } else if (source == busy){
          user.presence = Contact.DND;
          if (comm != null){
            header. setStatus ("Busy");
            comm.sendPresence("Busy", Contact.DND);
          }
        } else if (source == presenceHide){
          presence.remove(presenceHide);
          presence.add(presenceShow);
          comm.sendPresence("hide", 0);
        } else if (source == presenceShow){
          presence.remove(presenceShow);
          presence.add(presenceHide);
          comm.sendPresence("Online", Contact.ONLINE);
        } else if (source == presenceContext){
          ContextDialog dlg = new ContextDialog(this);
          dlg.show();
          if (dlg.okPressed){
            String context = dlg.tfContext.getText();
            comm.sendPresence(context, user.presence);
          }

        }

    }
}
```

JabberUser.java

```
package jabber;

public class JabberUser {

   String username;
   String jid;
   String password;
   int presence = Contact.ONLINE;

   public JabberUser(String name, String pwd, String jid) {
      this.username = name;
      this.password = pwd;
      this.jid = jid;
   }

   public void setUsername(String username){
      this.username = username;
   }

   public void setPassword(String pwd){
      this.password = pwd;
   }

   public void setJid(String jid){
      this.jid = jid;
   }

   public String getJid(){
      return jid;
   }

   public String toString(){
      return username;
   }
}
```

MessagePanel.java

```
package jabber;

import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;

public class MessagePanel extends Panel implements ActionListener {

  JabberFrame frame;
  TextArea output;
  TextArea write;
  Button send;
  Button cancel;
  Contact contact = null;

  public MessagePanel(JabberFrame parent){
    frame = parent;
    initGui();
  }

  public void initGui(){
    output = new TextArea("", 4, 10, TextArea.SCROLLBARS_VERTICAL_ONLY);
    output.setEditable(false);
    output.setBackground(Color.white);
    write = new TextArea("", 2, 10, TextArea.SCROLLBARS_NONE);
    write.requestFocus();
    send = new Button("Send");
    send.addActionListener(this);
    cancel = new Button("Cancel");
    cancel.addActionListener(this);

    setLayout(new GridBagLayout());
    GridBagConstraints gc = new GridBagConstraints();
    gc.fill = GridBagConstraints.BOTH;
    gc.gridx = 0;
    gc.gridy = 0;
    gc.gridwidth = 4;
    gc.gridheight = 4;
    gc.weightx = 100;
    gc.weighty = 10;
    add(output, gc);

    gc.gridy = 4;
    gc.gridwidth = 3;
    gc.gridheight = 2;
    gc.weightx = 100;
    gc.weighty = 0;
    add(write, gc);

    gc.fill = GridBagConstraints.BOTH;
    //gc.anchor = GridBagConstraints.CENTER;
    gc.gridx = 3;
    gc.gridy = 4;
    gc.gridwidth = 1;
    gc.gridheight = 1;
    gc.weightx = 0;
    gc.weighty = 0;
    add(send, gc);
    gc.gridy = 5;
    add(cancel, gc);

    }

    public void setContact (Contact c){
        if ( contact != c){
            output . setText (""") ;
            contact = c;
        }
        Enumeration enum = c.getUnreadMessages();
        while (enum.hasMoreElements()) {
            displayMessage(c.getName() , ( String )enum.nextElement()) ;
        }
        c.removeUnreadMessages();
    }

    public void displayMessage( String author , String msg){
        output . setForeground (Color. blue ) ;
        output .append(author + ":\n") ;
        output . setForeground (Color. black ) ;
        output .append(msg + "\n");
    }

    public void actionPerformed (ActionEvent e){
        Object o = e . getSource () ;
        if (o == send){
            if ( write . getText () . length () == 0)
                return;
            displayMessage(frame.getUser () . toString () , write . getText () ) ;
            frame.comm.sendMessage(contact.getJid () , write . getText () ) ;
            write . setText (""") ;
        } else if (o == cancel ){
            frame. setContactPanel () ;
        }
    }
}
```

PresenceHandler.java

```
package jabber;

import nanoxml.*;
import java.util.Enumeration;

public class PresenceHandler extends ElementHandler {

  public PresenceHandler(JabberFrame gui, JabberComm comm) {
    super(gui, comm);
  }
  public void handleElement(XMLElement xml) {
     String temp;
     String status = "";
     String jid = null;
     int device = Contact.DESKTOP;
     String type = null;
     int show = Contact.ONLINE;

     jid = trimJid(xml.getProperty("from"));
     type = xml.getProperty("type");
     if(type != null && type.equalsIgnoreCase("unavailable"))
       show = Contact.OFFLINE;
     if(type != null && type.equalsIgnoreCase("available"))
       show = Contact.ONLINE;

     for(Enumeration children = (Enumeration)xml.enumerateChildren(); children.hasMoreElements()
           ;){
       XMLElement node = (XMLElement)children.nextElement();
       String tagName = node.getTagName();
       if(tagName.equals("status")){
           status = node.getContents();
           if( status.equalsIgnoreCase("online"))
              show = Contact.ONLINE;
       }else if(tagName.equals("show")){
          temp = node.getContents();
          if(temp != null){
             if(temp.equalsIgnoreCase("dnd"))
               show = Contact.DND;
             else if(temp.equalsIgnoreCase("away"))
               show = Contact.AWAY;
             else if(temp.equalsIgnoreCase("xa"))
               show = Contact.XA;
             else if(temp.equalsIgnoreCase("online"))
               show = Contact.ONLINE;
          }
       } else if(tagName.equals("device")){
            temp = node.getContents();
            if(temp != null){
               if(temp.equals("mobile"))
                  device = Contact.MOBILE;
               else if(temp.equals("pda"))
                  device = Contact.PDA;
               else if(temp.equals("desktop")){
                  device = Contact.DESKTOP;
               }
            }
       }
     }
     gui.setPresence(jid, show, status, device);
 }}
```

SetupDialog.java

```
package jabber;

import java.awt.*;
import java.awt.event.*;

public class SetupDialog extends Dialog implements ActionListener {

  public final static String SERVER = "hauk02.idi.ntnu.no";

  TextField tfUsername = new TextField(10);
  Label lUsername = new Label("Username:");
  TextField tfPassword = new TextField(10);
  Label labelPassword = new Label("Password:");
  Checkbox storePwd = new Checkbox("Store password", false);
  TextField tfServer = new TextField(SERVER, 10);
  Label labelServer = new Label("Server:");
  Button cancel = new Button("Cancel");
  Button ok = new Button("OK");
  boolean okPressed = false;

  public SetupDialog(JabberFrame frame) {
    super(frame, "Setup", true);
    // setSize(Toolkit.getDefaultToolkit().getScreenSize());
    setSize(240, 320);

    ok.addActionListener(this);
    ok.setSize(cancel.getSize());
    cancel.addActionListener(this);

    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        dispose();
      }
    });
    initGui();
  }

  private void initGui(){
    Panel p = new Panel();
    p.setLayout(new GridBagLayout());
    GridBagConstraints gc = new GridBagConstraints();
    gc.fill = GridBagConstraints.HORIZONTAL;
    gc.gridheight = 1;
    gc.gridwidth = 1;
    gc.weightx = 100;
    gc.weighty = 0;
    gc.gridx = 0;
    gc.gridy = 0;
    gc.anchor = GridBagConstraints.EAST;
    p.add(lUsername, gc);

    gc.gridx = 1;
    gc.gridy = 0;
    gc.gridwidth = 2;
    gc.anchor = GridBagConstraints.WEST;
    p.add(tfUsername, gc);

    gc.gridx = 0;
    gc.gridy = 1;
    gc.gridwidth = 1;
```

```
        gc.anchor = GridBagConstraints.EAST;
        p.add(labelPassword, gc);

        gc.gridx = 1;
        gc.gridy = 1;
        gc.gridwidth = 2;
        gc.anchor = GridBagConstraints.WEST;
        p.add(tfPassword, gc);

        gc.gridx = 1;
        gc.gridy = 2;
        gc.gridwidth = 1;
        gc.anchor = GridBagConstraints.CENTER;
        p.add(storePwd, gc);

        gc.gridwidth = 1;
        gc.gridx = 0;
        gc.gridy = 3;
        gc.anchor = GridBagConstraints.EAST;
        p.add( labelServer , gc);

        gc.gridx = 1;
        gc.gridy = 3;
        gc.gridwidth = 2;
        gc.anchor = GridBagConstraints.WEST;
        p.add( tfServer , gc);

        gc.fill = GridBagConstraints.NONE;
        gc.gridx = 0;
        gc.gridy = 4;
        gc.gridwidth = 1;
        gc.anchor = GridBagConstraints.EAST;
        p.add(ok, gc);

        gc.gridx = 1;
        gc.gridy = 4;
        gc.anchor = GridBagConstraints.WEST;
        p.add(cancel, gc);

        add(p, BorderLayout.NORTH);
    }

    public void actionPerformed(ActionEvent e){
        Object o = e.getSource();
        if(o == cancel){
            dispose();
        } else if(o == ok){
            okPressed = true;
            dispose();
        }
    }
}
```

MessageHandler.java

```
package jabber;

import nanoxml.*;
import java.util.Enumeration;

public class MessageHandler extends ElementHandler {

  public MessageHandler(JabberFrame gui, JabberComm comm) {
    super(gui, comm);
  }

  public void handleElement(XMLElement xml) {
     String jid = trimJid(xml.getProperty("from"));
     String id = xml.getProperty("id");
     String subject = "";
     String body = "";

    for(Enumeration e = xml.enumerateChildren(); e.hasMoreElements(); ){
      XMLElement child = (XMLElement)e.nextElement();
      String tagName = child.getTagName();
      if (tagName.equals("subject")){
        subject = child.getContents();
      } else if (tagName.equals("body")){
        body = child.getContents();
      }
    }
    gui.newMessage(jid, subject, body);
  }
}
```

C.3 Desktop client

ConnectDialog.java

```
package jabberclient ;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ConnectDialog extends JDialog implements ActionListener{

  public ConnectDialog() {
  }

  public final static String SERVER = "hauk02.idi.ntnu.no";

  TextField tfUsername = new TextField(10);
  Label lUsername = new Label("Username:");
  TextField tfPassword = new TextField(10);
  Label labelPassword = new Label("Password:");
  Checkbox storePwd = new Checkbox("Store password", false);
  TextField tfServer = new TextField(SERVER, 10);
  Label labelServer = new Label("Server:");
  Button cancel = new Button("Cancel");
  Button ok = new Button("OK");
  boolean okPressed = false;

  public ConnectDialog(JabberFrame frame) {
    super(frame, "Connect", true);
    // setSize(Toolkit.getDefaultToolkit().getScreenSize());
    setSize(240, 240);

    ok.addActionListener(this);
    ok.setSize(cancel.getSize());
    cancel.addActionListener(this);

    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        dispose();
      }
    });
    initGui();
  }

  private void initGui(){
    JPanel p = new JPanel();
    p.setLayout(new GridBagLayout());
    GridBagConstraints gc = new GridBagConstraints();
    gc.fill = GridBagConstraints.HORIZONTAL;
    gc.gridheight = 1;
    gc.gridwidth = 1;
    gc.weightx = 100;
    gc.weighty = 0;
    gc.gridx = 0;
    gc.gridy = 0;
    gc.anchor = GridBagConstraints.EAST;
    p.add(lUsername, gc);

    gc.gridx = 1;
    gc.gridy = 0;
```

```
      gc.gridwidth = 2;
      gc.anchor = GridBagConstraints.WEST;
      p.add(tfUsername, gc);

      gc.gridx = 0;
      gc.gridy = 1;
      gc.gridwidth = 1;
      gc.anchor = GridBagConstraints.EAST;
      p.add(labelPassword, gc);

      gc.gridx = 1;
      gc.gridy = 1;
      gc.gridwidth = 2;
      gc.anchor = GridBagConstraints.WEST;
      p.add(tfPassword, gc);

      gc.gridx = 1;
      gc.gridy = 2;
      gc.gridwidth = 1;
      gc.anchor = GridBagConstraints.CENTER;
      p.add(storePwd, gc);

      gc.gridwidth = 1;
      gc.gridx = 0;
      gc.gridy = 3;
      gc.anchor = GridBagConstraints.EAST;
      p.add(labelServer, gc);

      gc.gridx = 1;
      gc.gridy = 3;
      gc.gridwidth = 2;
      gc.anchor = GridBagConstraints.WEST;
      p.add(tfServer, gc);

      gc.fill = GridBagConstraints.NONE;
      gc.gridx = 0;
      gc.gridy = 4;
      gc.gridwidth = 1;
      gc.anchor = GridBagConstraints.EAST;
      p.add(ok, gc);

      gc.gridx = 1;
      gc.gridy = 4;
      gc.anchor = GridBagConstraints.WEST;
      p.add(cancel, gc);

      getContentPane().add(p, BorderLayout.NORTH);
    }

    public void actionPerformed(ActionEvent e){
      Object o = e.getSource();
      if(o == cancel){
        dispose();
      } else if(o == ok){
        okPressed = true;
        dispose();
      }
    }
  }
```

Contact.java

```
package jabberclient;

import java.util.Enumeration;
import java.util.Vector;

public class Contact {

  public final static int MOBILE = 0;
  public final static int DESKTOP = 1;
  public final static int PDA = 2;

  public final static int ONLINE = 0;
  public final static int OFFLINE = 1;
  public final static int DND = 2;
  public final static int AWAY = 3;
  public final static int XA = 4;

  private String username;
  private String jid;
  private String status = "Offline";
  int device;
  int show;
  Vector unreadMessages;

  public Contact(String jid, String name){
    this.username = name;
    this.jid = jid;
    show = OFFLINE;
    device = DESKTOP;
    unreadMessages = new Vector();
  }

  public void setPresence(int show, String status, int device){
      this.show = show;
      this.status = status;
      this.device = device;
  }

  public void addMessage(String msg){
    int end = unreadMessages.size();
    unreadMessages.insertElementAt(msg, end);
  }

  public Enumeration getUnreadMessages(){
    return unreadMessages.elements();
  }

  public void removeUnreadMessages(){
    unreadMessages.removeAllElements();
  }

  public String getName(){
    return username;
  }

  public String getJid(){
    return jid;
  }

  public int getShow(){
        return show;
    }

    public int getDevice(){
        return device;
    }

    public String getStatus(){
        return status;
    }

    public String toString(){
        return username;
    }
}
```

ContactDialog.java

```
package jabberclient ;

import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.event.*;

public class ContactDialog extends JDialog {
  // Gui
  GridBagLayout gridBagLayout1 = new GridBagLayout();
  JButton buttonOk = new JButton();
  JButton buttonCancel = new JButton();
  TitledBorder titledBorder1 ;
  JTabbedPane jTabbedPane1 = new JTabbedPane();
  JPanel panelMobile = new JPanel();
  GridBagLayout gridBagLayout2 = new GridBagLayout();
  JList listAllMobile = new JList();
  JList listSelectedMobile = new JList();
  DefaultListModel lmAllMobile = new DefaultListModel();
  DefaultListModel lmSelectedMobile = new DefaultListModel();
  TitledBorder titledBorder2 ;
  TitledBorder titledBorder3 ;
  JButton buttonAddMobile = new JButton();
  JButton buttonRemoveMobile = new JButton();
  JList listSelectedPda = new JList();
  JList listAllPda = new JList();
  DefaultListModel lmAllPda = new DefaultListModel();
  DefaultListModel lmSelectedPda = new DefaultListModel();
  JButton buttonAddPda = new JButton();
  JPanel panelPDA = new JPanel();
  GridBagLayout gridBagLayout3 = new GridBagLayout();
  JButton buttonRemovePda = new JButton();

  // Member variables
  JabberFrame frame;
  Vector contacts = new Vector();
  boolean okPressed = false ;

  public ContactDialog(JabberFrame frame){
    this.frame = frame;
    try {
      jbInit ();
    }
    catch(Exception e) {
      e.printStackTrace ();
    }
    Enumeration enum = frame.contactsHash.elements();
    while (enum.hasMoreElements()){
      Object o = enum.nextElement();
      lmAllMobile.addElement(o);
      lmAllPda.addElement(o);
    }
  }

  public ContactDialog() {
    try {
      jbInit ();
    }
    catch(Exception e) {
      e.printStackTrace();
    }

  }
  private void jbInit() throws Exception {
    titledBorder1 = new TitledBorder(BorderFactory.createEtchedBorder(Color.white,new Color
          (178, 178, 178)),"Contacts");
    titledBorder2 = new TitledBorder(BorderFactory.createEtchedBorder(Color.white,new Color
          (178, 178, 178)),"Selected");
    titledBorder3 = new TitledBorder(BorderFactory.createEtchedBorder(Color.white,new Color
          (178, 178, 178)),"Available");
    this.getContentPane().setLayout(gridBagLayout1);
    buttonOk.setText("OK");
    buttonOk.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonOk_actionPerformed(e);
      }
    });
    buttonCancel.setText("Cancel");
    buttonCancel.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonCancel_actionPerformed(e);
      }
    });
    panelMobile.setLayout(gridBagLayout2);
    listSelectedMobile.setBorder(titledBorder2);
    listSelectedMobile.setModel(lmSelectedMobile);
    listAllMobile.setBorder(titledBorder3);
    listAllMobile.setModel(lmAllMobile);
    buttonAddMobile.setText("-->");
    buttonAddMobile.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonAddMobile_actionPerformed(e);
      }
    });
    buttonRemoveMobile.setText("<--");
    buttonRemoveMobile.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonRemoveMobile_actionPerformed(e);
      }
    });
    listSelectedPda.setBorder(titledBorder2);
    listSelectedPda.setModel(lmSelectedPda);
    listAllPda.setBorder(titledBorder3);
    listAllPda.setModel(lmAllPda);
    buttonAddPda.setText("-->");
    buttonAddPda.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonAddPda_actionPerformed(e);
      }
    });
    panelPDA.setLayout(gridBagLayout3);
    buttonRemovePda.setText("<--");
    buttonRemovePda.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonRemovePda_actionPerformed(e);
      }
    });
    this.getContentPane().add(buttonOk,           new GridBagConstraints
          (0, 4, 1, 1, 0.0, 0.0
            ,GridBagConstraints.EAST, GridBagConstraints.NONE, new Insets(0, 50, 0, 0), 0, 0));
    this.getContentPane().add(buttonCancel,                         new GridBagConstraints
            (2, 4, 1, 1, 0.0, 0.0
            , GridBagConstraints.EAST, GridBagConstraints.NONE, new Insets (0, 0, 0, 50)   , 0, 0) );
    this.getContentPane().add(jTabbedPane1,     new GridBagConstraints    (0, 0, 3, 3, 1.0, 1.0
            , GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0)   , 0, 0) );
    jTabbedPane1.add(panelMobile,   "Mobile_phone");
    panelMobile.add( listAllMobile ,       new GridBagConstraints    (0, 0, 2, 4, 1.0, 1.0
            , GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0)   , 0, 0) );
    panelMobile.add( listSelectedMobile ,        new GridBagConstraints    (3, 0, 2, 4, 1.0, 1.0
            , GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0)   , 0, 0) );
    panelMobile.add(buttonAddMobile,             new GridBagConstraints    (2, 0, 1, 1, 0.0, 0.0
            , GridBagConstraints.SOUTH, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0)   , 0, 0) );
    panelMobile.add(buttonRemoveMobile,          new GridBagConstraints    (2, 3, 1, 1, 0.0, 0.0
            , GridBagConstraints.NORTH, GridBagConstraints.NONE, new Insets (0, 0, 0, 0)   , 0, 0) );
    jTabbedPane1.add(panelPDA,   "PDA");
    panelPDA.add( listAllPda , new GridBagConstraints    (0, 0, 2, 4, 1.0, 1.0
            , GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0)   , 0, 0) );
    panelPDA.add( listSelectedPda , new GridBagConstraints    (3, 0, 2, 4, 1.0, 1.0
            , GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0)   , 0, 0) );
    panelPDA.add(buttonAddPda, new GridBagConstraints    (2, 2, 1, 1, 0.0, 0.0
            , GridBagConstraints.SOUTH, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0)   , 0, 0) );
    panelPDA.add(buttonRemovePda, new GridBagConstraints    (2, 3, 1, 1, 0.0, 0.0
            , GridBagConstraints.NORTH, GridBagConstraints.NONE, new Insets (0, 0, 0, 0)   , 0, 0) );
  }

  void buttonAddMobile_actionPerformed(ActionEvent e) {
    Object item = listAllMobile . getSelectedValue ();
    if (item == null)
      return;
    lmAllMobile.removeElement(item);
    lmSelectedMobile.addElement(item);
  }

  void buttonRemoveMobile_actionPerformed(ActionEvent e) {
    Object item = listSelectedMobile . getSelectedValue ();
    if (item == null)
      return;
    lmAllMobile.addElement(item);
    lmSelectedMobile.removeElement(item);
  }

  void buttonAddPda_actionPerformed(ActionEvent e) {
    Object item = listAllPda . getSelectedValue ();
    if (item == null)
      return;
    lmSelectedPda.addElement(item);
    lmAllPda.removeElement(item);
  }

  void buttonRemovePda_actionPerformed(ActionEvent e) {
    Object item = listSelectedPda . getSelectedValue ();
    if (item == null)
      return;
    lmAllPda.addElement(item);
    lmSelectedPda.removeElement(item);
  }

  void buttonOk_actionPerformed(ActionEvent e) {
    okPressed = true;
    dispose ();
  }

  void buttonCancel_actionPerformed (ActionEvent e) {
    dispose () ;
  }
}
```

ElementHandler.java

```
package jabberclient ;

import nanoxml.*;

public abstract class ElementHandler {

   protected JabberFrame gui;
   protected JabberComm comm;

   public ElementHandler(JabberFrame gui, JabberComm comm) {
      this.gui = gui;
      this.comm = comm;
   }

   public abstract void handleElement(XMLElement xml);

   public static String trimJid (String jid){
      int endIndex = jid.indexOf('/');
      if (endIndex > 0)
         return jid.substring (0, endIndex);
      else
         return jid;
   }
}
```

IqHandler.java

```
package jabberclient;

import java.util.Vector;
import java.util.Enumeration;
import nanoxml.*;

 public class IqHandler extends ElementHandler {

   private final static String onlineMessage = ”<presence><status>Online</status></presence>”
       ;
   private final static String CONTACT_LIST_NAMESPACE = ”\”jabber:ext:iq:rosterconfig\””;
   private final static String PREFERENCES_NAMESPACE = ”\”jabber:ext:iq:preferences\””;

   public IqHandler(JabberFrame gui, JabberComm comm) {
     super(gui, comm);
   }

   public void handleElement(XMLElement xml) {
      String type;
      String id;

      type = xml.getProperty(”type”);
      id = xml.getProperty(”id”);

      if (type.equalsIgnoreCase(”result”) || type.equalsIgnoreCase(”set”)){
        if (id.equalsIgnoreCase(”logon”)){
          gui.println(”Logon␣OK”);
          gui.connected(true);
          gui.setTitle();
          comm.send(onlineMessage);
          comm.sendRoster();
        } else if (id.equalsIgnoreCase(”roster”)){
          XMLElement child = (XMLElement)xml.getChildren().elementAt(0);
          for(Enumeration e = child.enumerateChildren(); e.hasMoreElements(); ){
            XMLElement contact = (XMLElement)e.nextElement();
            String nick = contact.getProperty(”name”);
            String jid = contact.getProperty(”jid”);
            gui.addContact(new Contact(jid, nick));
          }
        }

      } else if (type.equalsIgnoreCase(”error”)){
        gui.println(”Logon␣failed”);
      } else {
        gui.println(”Handle␣me:\n” + xml.toString());
      }
   }

   public static String getContactListMsg(Enumeration mobileContacts, Enumeration pdaContacts){
      StringBuffer buf = new StringBuffer();

      buf.append(”<iq␣type=\”set\”>”);
      buf.append(”<query␣xmlns=” + CONTACT_LIST_NAMESPACE + ”>”);

      buf.append(”<mobile>”);
      while(mobileContacts.hasMoreElements()){
          String jid = ((Contact)mobileContacts.nextElement()).getJid();
          buf.append(”<item␣jid=\”” + jid + ”\”␣/>”);
      }
      buf.append(”</mobile>”);

      buf.append("<pda>");
      while(pdaContacts.hasMoreElements()){
            String jid = ((Contact)pdaContacts.nextElement()).getJid();
            buf.append("<item jid=\"" + jid + "\" />");
      }
      buf.append("</pda>");

      buf.append("</query>");
      buf.append("</iq>");
      return buf.toString();
   }

   public static String getPreferencesMessage(PreferencesInterface pref){
      StringBuffer buf = new StringBuffer();

      buf.append("<iq type=\"set\">");
      buf.append("<query xmlns=" + PREFERENCES_NAMESPACE + ">");

      buf.append("<mobile>");
      if(pref.getMobileBlock())
         buf.append("<block/>");
      String size = pref.getMobileMaxSize();
      if(size != null){
         buf.append("<size max=\"");
         buf.append(size);
         buf.append("\"/>");
      }
      buf.append("</mobile>");

      buf.append("<pda>");
      if(pref.getPdaBlock())
         buf.append("<block/>");
      size = pref.getPdaMaxSize();
      if(size != null){
         buf.append("<size max=\"");
         buf.append(size);
         buf.append("\"/>");
      }
      buf.append("</pda>");

      if(pref.getDesktopBlock()){
         buf.append("<desktop>");
         buf.append("<block/>");
         buf.append("</desktop>");
      } else {
         buf.append("<desktop/>");
      }

      buf.append("</query>");
      buf.append("</iq>");
      return buf.toString();
   }
}
```

JabberClient.java

```
package jabberclient ;

import javax.swing.UIManager;

public class JabberClient {

  public JabberClient () {
  }

  public static void main(String [] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e) {
      e. printStackTrace () ;
    }
    JabberFrame frame = new JabberFrame();
    frame. setVisible (true);
  }
}
```

JabberComm.java

```
package jabberclient;

import java.net.*;
import java.io.*;
import nanoxml.*;
import tools.NonBlockingInputStream;

public class JabberComm implements Runnable {

  // Socket variables
  int serverPort = 5222;
  String serverAddress;
  Socket socket;
  OutputStream out;
  InputStream in;
  boolean read = true;

  // Member variables
  JabberFrame gui;
  JabberUser user;

  // Element handlers
  ElementHandler iqHandler;
  ElementHandler messageHandler;
  ElementHandler presenceHandler;

  // XML strings
  String connectString = "<stream:stream to=\"hauk02.idi.ntnu.no\" " +
                         "xmlns=\"jabber:client\" " +
                         "xmlns:stream=\"http://etherx.jabber.org/streams\">";

  String rosterString = "<iq id=\"roster\" type=\"get\">" +
                          "<query xmlns=\"jabber:iq:roster\"/>" +
                        "</iq>";

  public JabberComm(JabberFrame gui, JabberUser user) throws Exception {
    this.gui = gui;
    this.user = user;
    iqHandler = new IqHandler(gui, this);
    messageHandler = new MessageHandler(gui, this);
    presenceHandler = new PresenceHandler(gui, this);
    // createParser();
  }

  public void connect(String adr) throws Exception {
    try{
      socket = new Socket(adr, serverPort);
      out = socket.getOutputStream();
      in = socket.getInputStream();
      (new Thread(this)).start();
      send(getConnectMessage(adr));
    } catch (Exception e){
      read = false;
      throw e;
    }
  }

  public void disconnect(){
    send("<presence type=\"unavailable\"/>");
    send("</stream:stream>");
```

```
      gui.clearContacts();
      gui.connected(false);
   }

   public void sendPresence(String context, int show){
      if(context.equals(”hide”)){
         send(”<presence type=\”unavailable\”/>”);
         return;
      }
      StringBuffer buf = new StringBuffer();
      buf.append(”<presence from=\”” + user.getJid() + ”\” type=\”available\”><show>”);
      switch(show){
      case Contact.ONLINE:
         buf.append(”online”);
         break;
      case Contact.XA:
         buf.append(”xa”);
         break;
      case Contact.AWAY:
         buf.append(”away”);
         break;
      case Contact.DND:
         buf.append(”dnd”);
         break;
      }
      buf.append(”</show>”);
      buf.append(”<status>” + context + ”</status>”);
      buf.append(”</presence>”);
      send(buf.toString());
   }/

   public void sendMessage(String toUserJid, String msg){
      String xml = ”<message id=\”12\” to=\”” + toUserJid + ”\”>” +
                   ”<body>” + msg + ”</body>” +
                   ”</message>”;
      send(xml);
   }

   public void run(){
      String s;
      byte[] buf;
      int bytesRead = 0;
      while(read)
      try {
         int available = in.available();
         if(available > 0){
            buf = new byte[available];
            bytesRead = in.read(buf, 0, available);
            String temp = new String(buf);
            //System.out.println(”<-- ” + temp.substring(0, bytesRead));
            processMessage(temp.substring(0, bytesRead));
         } else {
            try{
               Thread.sleep(200);
            } catch (InterruptedException ie){}
         }

      } catch(IOException ioe){
         gui.println(”Error reading”);
         read = false;
      }
   }
```

```

    private void processMessage(String s){
      int offset = 0;
      int pos;
      XMLElement m;
      System.out.println("<-- " + s);
      if(s.charAt(1) == '?'){ // begin document
         offset = s.indexOf('>', offset);
         offset +=1;
         if(offset + 6 >= s.length())
           return;
      }
      String temp = s.substring(offset +1, offset +7);
      if(temp.equals("stream")){
         offset += s.indexOf('>', offset);
         sendLogon();
         gui.connected(true);
         return;
      }


      while(offset < s.length()){
         try{
           m = new XMLElement();
           offset += m.parseString(s, offset);
           String elementName = m.getTagName();
           if(elementName.equals("iq")){
             iqHandler.handleElement(m);
           } else if(elementName.equals("presence")){
             presenceHandler.handleElement(m);
           } else if(elementName.equalsIgnoreCase("message")){
             messageHandler.handleElement(m);
           }
         }catch(XMLParseException xmle){
           gui.println(xmle.getMessage());
           break;
         }
      }
    }

    public void send(String s) {
      try {
         out.write(s.getBytes());
         out.flush();
         System.out.println("--> " + s);
      }catch (IOException ioe){
         gui.println("Error sending:\n" + ioe.getMessage());
         ioe.printStackTrace(System.err);
      }
    }

    private String getConnectMessage(String server){
      String temp = "<stream:stream to=\"" + server + "\" " +
                 "xmlns=\"jabber:client\" " +
                 "xmlns:stream=\"http://etherx.jabber.org/streams\">";
      return temp;
    }

    private String getLogonMessage(){
    return "<iq type=\"set\" id=\"logon\">" +
           "<query xmlns=\"jabber:iq:auth\">" +
           "<username>" + user.toString() + "</username>" +
            "<password>" + user.password + "</password>" +
            "<resource>test</resource>" +
            "</query>" +
            "</iq>";
  }

  public void sendLogon(){
    send(getLogonMessage());
  }

  public void sendRoster(){
    send( rosterString );
  }
}
```

JabberFrame.java

```
package jabberclient;

import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.util.Enumeration;
import java.util.Hashtable;

import nanoxml.*;
import java.awt.Font;

public class JabberFrame extends JFrame implements ActionListener, TreeSelectionListener {

  // GUI elements
  JPanel contentPane;
  JTextArea output;
  JSplitPane splitPane;
  JTree tree;
  DefaultTreeModel treeModel;
  DefaultMutableTreeNode rootNode;

  // Menues
  JMenuBar menuBar = new JMenuBar();
  JMenu menuFile = new JMenu();
  JMenuItem menuFileConnect = new JMenuItem("Connect", 'c');
  JMenuItem menuFileDisconnect = new JMenuItem("Disconnect", 'd');
  JMenuItem menuFilePreferences = new JMenuItem("Preferences", 'p');
  JMenuItem menuFileExit = new JMenuItem("Exit", 'e');

  JMenu menuPresence = new JMenu();
  JMenuItem menuPresenceOnline = new JMenuItem("Online", 'o');
  JMenuItem menuPresenceDnd = new JMenuItem("Do not disturb", 'b');
  JMenuItem menuPresenceAway = new JMenuItem("Away", 'a');
  JMenuItem menuPresenceXA = new JMenuItem("Extended away", 'x');
  JMenuItem menuPresenceHide = new JMenuItem("Hide presence");
  JMenuItem menuPresenceShow = new JMenuItem("Show presence");

  JMenu menuContacts = new JMenu("Contacts");
  JMenuItem menuContactsAdd = new JMenuItem("Add contact");
  JMenuItem menuContactsRemove = new JMenuItem("Remove contact");
  JMenuItem menuContactsAdm = new JMenuItem("Edit contact lists");

  //Popup menu
  JMenuItem popupSend = new JMenuItem("Send message");
  JMenuItem popupRemove = new JMenuItem();

  // Member variables
  JabberComm comm;
  Hashtable contactsHash = new Hashtable();
  Hashtable contactWindowHash = new Hashtable();
  JabberUser user;
  Contact selectedContact = null;
  String serverAddress;
  String context;

  public JabberFrame() {
    super("Jabber");
    try{
        jbInit();
    } catch (Exception e){
    }
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }});

    ImageIcon icon = new ImageIcon("bulb_online.gif");

    setIconImage(icon.getImage());
    setSize(250,350);
}

private void jbInit() throws Exception{
    Border etched = BorderFactory.createEtchedBorder();
    Border border = BorderFactory.createTitledBorder(etched, "Output");
    JScrollPane scrollOutput;
    JScrollPane scrollTree;

    // Output area
    output = new JTextArea();
    output.setLineWrap(true);
    output.setBorder(border);
    scrollOutput = new JScrollPane(output);
    scrollOutput.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

    // Tree area
    rootNode = new DefaultMutableTreeNode("Contacts");
    treeModel = new DefaultTreeModel(rootNode);
    treeModel.setAsksAllowsChildren(true);
    tree = new JTree(treeModel);
    tree.putClientProperty("JTree.lineStyle", "Angled");
    tree.addTreeSelectionListener(this);
    tree.setCellRenderer(new TreeRenderer());
    ToolTipManager.sharedInstance().registerComponent(tree);
    scrollTree = new JScrollPane(tree);

    MouseListener ml = new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        TreePath path = tree.getPathForLocation(e.getX(), e.getY());
        if (path == null)
            return;
        DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)path.getLastPathComponent()
            ;
        if (selectedNode == rootNode)
            return;
        Contact c = (Contact)selectedNode.getUserObject();
        selectedContact = c;
        showPopupMenu(c, e);
     }};
    tree.addMouseListener(ml);

    splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT, scrollTree, scrollOutput);
    splitPane.setDividerLocation(200);

    contentPane = (JPanel)getContentPane();
    contentPane.add(splitPane);

    // Menus
    setJMenuBar(menuBar);
```

```

        menuFile.setText("File");
        menuFile.setMnemonic('f');
        menuFileConnect.addActionListener(this);
        menuFile.add(menuFileConnect);
        menuFileDisconnect.addActionListener(this);
        menuFileDisconnect.setEnabled(false);
        menuFile.add(menuFileDisconnect);
        menuFilePreferences.addActionListener(this);
        menuFile.add(menuFilePreferences);
        menuFile.addSeparator();
        menuFileExit.addActionListener(this);
        menuFile.add(menuFileExit);
        menuBar.add(menuFile);

        menuPresence.setText("Presence");
        menuPresence.setMnemonic('p');
        menuPresenceOnline.addActionListener(this);
        menuPresence.add(menuPresenceOnline);
        menuPresenceDnd.addActionListener(this);
        menuPresence.add(menuPresenceDnd);
        menuPresenceAway.addActionListener(this);
        menuPresence.add(menuPresenceAway);
        menuPresenceXA.addActionListener(this);
        menuPresence.add(menuPresenceXA);
        menuPresence.add(new JSeparator());
        menuPresenceShow.addActionListener(this);
        menuPresenceHide.addActionListener(this);
        menuPresence.add(menuPresenceHide);
        menuBar.add(menuPresence);

        menuContacts.setMnemonic('c');
        menuContactsAdd.addActionListener(this);
        menuContacts.add(menuContactsAdd);
        menuContactsRemove.addActionListener(this);
        menuContacts.add(menuContactsRemove);
        menuContacts.addSeparator();
        menuContactsAdm.addActionListener(this);
        menuContacts.add(menuContactsAdm);
        menuBar.add(menuContacts);

        // Popup menu
        popupSend.addActionListener(this);
        popupRemove.addActionListener(this);
    }

    public void newMessage(String jid, String subject, String body){
        Contact c = (Contact)contactsHash.get(jid);
        if (c == null)
            return; // Do something!!!
        MessageFrame frame = showMessageDialog(c);
        frame.displayMessage(c.getName(), body);
    }

    public void setPresence(String jid, int show, String status, int device){
        Contact contact = (Contact)contactsHash.get(jid);
        if (contact != null){
            contact.setPresence(show, status, device);
        }
        treeModel.nodeChanged(rootNode);
    }

    public void addContact(Contact contact){
        contactsHash.put(contact.getJid(), contact);
        DefaultMutableTreeNode node = new DefaultMutableTreeNode(contact, false);
        treeModel.insertNodeInto(node, rootNode, rootNode.getChildCount());
    }

    public void removeContact(String name){

    }

    public void clearContacts(){
        rootNode.removeAllChildren();
        treeModel.reload();
    }



    public void print(char c){
        output.append((new Character(c)).toString());
    }

    public void print(String s){
        output.append(s);
    }

    public void println(String s){
        output.append(s + "\n");
    }

    public void setTitle(){
        setTitle("Jabber␣-␣" + user.getJid());
    }

    private boolean showConnectDialog(){
        ConnectDialog dlg = new ConnectDialog(this);
        dlg.show();
        if(dlg.okPressed){
            String userName = dlg.tfUsername.getText();
            serverAddress = dlg.tfServer.getText();
            user = new JabberUser(userName, dlg.tfPassword.getText(), userName + "@" + serverAddress)
                ;
            return true;
        }
        return false;
    }

    private MessageFrame showMessageDialog(Contact contact){
        if(contact == null)
            return null; // Notify user
        MessageFrame currentFrame = (MessageFrame)contactWindowHash.get(contact.getJid());
        if(currentFrame == null){
            currentFrame = new MessageFrame(contact, this);
            currentFrame.setSize(250,350);
            currentFrame.setVisible(true);
            contactWindowHash.put(contact.getJid(), currentFrame);
        } else {
            currentFrame.setVisible(true);
        }
        return currentFrame;
    }

    private void showContactsDialog(){
```

```
    ContactDialog dlg = new ContactDialog(this);
    dlg.setSize(300,300);
    dlg.setModal(true);
    dlg.show();
    if(dlg.okPressed){
      String msg = IqHandler.getContactListMsg(dlg.lmSelectedMobile.elements(), dlg.
           lmSelectedPda.elements());
      comm.send(msg);
      System.out.println("-->␣" + msg);
    }
  }

  private void showPreferencesDialog(){
    PreferencesDialog dlg = new PreferencesDialog();
    dlg.setSize(300, 400);
    dlg.setModal(true);
    dlg.show();
    if(dlg.okPressed){
      String msg = IqHandler.getPreferencesMessage(dlg);
      System.out.println(msg);
      if(comm != null && msg != null)
        comm.send(msg);
    }
  }

  private void connect(){
    try {
      if(user == null){
        if(!showConnectDialog())
          return;
      }
      println("Connecting ...");
      comm = new JabberComm(this, user);
      comm.connect(serverAddress);
    }catch(Exception e){
      output.setText("Connection␣failed:\n" + e.getMessage());
    }
  }

  private void showPopupMenu(Contact c, MouseEvent e){
    JPopupMenu pm = new JPopupMenu();
    JMenuItem jid = new JMenuItem(c.getJid());
    jid.setEnabled(false);
    pm.add(jid);
    pm.addSeparator();
    pm.add(popupSend);
    popupRemove.setText(c.getName());
    pm.add(popupRemove);
    pm.show(tree, e.getX(), e.getY());
  }

  public void actionPerformed(ActionEvent event){
    JMenuItem item = (JMenuItem)event.getSource();
    if(item == menuFileExit){
      System.exit(0);
    } else if(item == menuFileConnect){
      connect();
    } else if(item == menuFileDisconnect){ /** @todo check that comm != null*/
      comm.disconnect();
    } else if(item == menuPresenceDnd){
      if(context == null)
        comm.sendPresence("dnd", Contact.DND);
```

```
            else
              comm.sendPresence(context, Contact.DND);
        } else if (item == menuPresenceOnline){
            if (context == null)
              comm.sendPresence("online", Contact.ONLINE);
            else
              comm.sendPresence(context, Contact.ONLINE);
        } else if (item == menuPresenceAway){
            if (context == null)
              comm.sendPresence("away", Contact.AWAY);
            else
              comm.sendPresence(context, Contact.AWAY);
        }else if (item == menuPresenceXA){
            if (context == null)
              comm.sendPresence("xa", Contact.XA);
            else
                comm.sendPresence(context, Contact.XA);
        } else if (item == popupSend){
            showMessageDialog(selectedContact);
        } else if (item == menuContactsAdm){
            showContactsDialog();
        } else if (item == menuFilePreferences){
              showPreferencesDialog();
        } else if (item == menuPresenceHide){
              comm.sendPresence("hide", 0);
              menuPresence.remove(menuPresenceHide);
              menuPresence.add(menuPresenceShow);
              menuPresence.validate();
        } else if (item == menuPresenceShow){
              comm.sendPresence("online", Contact.ONLINE);
              menuPresence.remove(menuPresenceShow);
              menuPresence.add(menuPresenceHide);
              menuPresence.validate();
        }
    }

    public void valueChanged(TreeSelectionEvent e){
        TreePath path = tree.getSelectionPath();
        if (path == null)
              return;
    }

    public void connected(boolean b){
        menuFileConnect.setEnabled(!b);
        menuFileDisconnect.setEnabled(b);
        menuFile.validate();
    }
}
```

JabberUser.java

```
package jabberclient ;

public class JabberUser {

   String username;
   String jid ;
   String password;

   public JabberUser( String name, String pwd, String jid ) {
      this .username = name;
      this .password = pwd;
      this . jid = jid ;
   }

   public void setUsername(String username){
      this .username = username;
   }

   public void setPassword( String pwd){
      this .password = pwd;
   }

   public void setJid ( String jid ){
      this . jid = jid ;
   }

   public String getJid (){
      return jid ;
   }

   public String toString (){
      return username;
   }
}
```

MessageFrame.java

```
package jabberclient;

import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;
import java.awt.event.*;

public class MessageFrame extends JFrame {

  // GUI elements
  TitledBorder titledBorder1;
  JTextArea inputArea = new JTextArea();
  GridBagLayout gridBagLayout1 = new GridBagLayout();
  JLabel headerPanel = new JLabel();
  JButton buttonSend = new JButton();

  // Member variables
  Contact contact;
  JabberFrame frame;
  JTextArea otuputArea = new JTextArea();

  public MessageFrame(Contact contact, JabberFrame frame) {
    super(contact.getName());
    this.contact = contact;
    this.frame = frame;
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  private void jbInit() throws Exception {
    headerPanel.setFont(new Font("Times", Font.BOLD, 14));
    headerPanel.setForeground(Color.blue);
    headerPanel.setText("␣Til:␣" + contact.getName() + "␣" + contact.getStatus());
    titledBorder1 = new TitledBorder("");
    JScrollPane scroll = new JScrollPane(otuputArea);
    scroll.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
    scroll.setAutoscrolls(true);

    inputArea.setMaximumSize(new Dimension(10, 10));
    inputArea.setMinimumSize(new Dimension(10, 10));
    inputArea.setLineWrap(true);
    inputArea.setRows(4);

    this.getContentPane().setLayout(gridBagLayout1);

    buttonSend.setToolTipText("Send␣message␣to␣" + contact.getName());
    buttonSend.setText("Send");
    buttonSend.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonSend_actionPerformed(e);
      }
    });
    otuputArea.setBorder(BorderFactory.createEtchedBorder());
    otuputArea.setPreferredSize(new Dimension(40, 20));
    otuputArea.setCaretPosition(0);
    otuputArea.setEditable(false);
    otuputArea.setColumns(20);
```

```
        otuputArea.setLineWrap(true);
        otuputArea.setRows(500);
        this.getContentPane().add( scroll , new GridBagConstraints (0, 1, 5, 3, 1.0, 1.0
                , GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets (0, 0, 1, 0) , 92, 36)
                    );
        scroll .getViewport().add(otuputArea , null);
        this.getContentPane().add(inputArea , new GridBagConstraints (0, 4, 3, 2, 1.0, 0.0
                , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL, new Insets(1, 0, 2, 0)
                    , 162, 40));
        this.getContentPane().add(headerPanel , new GridBagConstraints (0, 0, 2, 1, 0.0, 0.0
                , GridBagConstraints.CENTER, GridBagConstraints.NONE, new Insets (0, 0, 13, 0) , 0, 0)
                    );
        this.getContentPane().add(buttonSend , new GridBagConstraints (4, 4, 1, 2, 0.0, 0.0
                , GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets (0, 0, 0, 0) , 6, 0) );
    }

    public void displayMessage( String user , String msg){
        otuputArea.setForeground(Color.blue);
        otuputArea.append(user + ":\n");
        otuputArea.setForeground(Color.black);
        otuputArea.append(msg + "\n");
    }

    void buttonSend_actionPerformed(ActionEvent e) {
            String s = inputArea.getText();
            if (s.length() == 0)
              return;
            inputArea.setText("");
            displayMessage(frame.user.toString(), s);
            if (frame.comm != null)
              frame.comm.sendMessage(contact.getJid(), s);
    }
}
```

MessageHandler.java

```
package jabberclient;

import nanoxml.*;
import java.util.Enumeration;

public class MessageHandler extends ElementHandler {

  public MessageHandler(JabberFrame gui, JabberComm comm) {
    super(gui, comm);
  }

  public void handleElement(XMLElement xml) {
    String jid = trimJid(xml.getProperty("from"));
    String id = xml.getProperty("id");
    String subject = "";
    String body = "";

    for(Enumeration e = xml.enumerateChildren(); e.hasMoreElements();){
      XMLElement child = (XMLElement)e.nextElement();
      String tagName = child.getTagName();
      if(tagName.equals("subject")){
        subject = child.getContents();
      } else if(tagName.equals("body")){
        body = child.getContents();
      }
    }
    gui.newMessage(jid, subject, body);
  }
}
```

PreferencesDialog.java

```
package jabberclient ;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import com.borland.jbcl.layout.*;
import java.awt.event.*;
import javax.swing.event.*;

public class PreferencesDialog extends JDialog implements PreferencesInterface {

    // Member variables
    boolean okPressed = false ;
    JabberFrame frame;

    boolean blockDesktop = false ;
    boolean blockMobile = false ;
    boolean blockPda = false ;
    boolean filterMobile = false ;
    String filterMobileValue = null;
    boolean filterPda = false ;
    String filterPdaValue = null;

    boolean forwardSMS = false;
    boolean forwardEmail = false ;
    String forwardParameter = null;


    // Gui elements
    IntegerVerifier iv = new IntegerVerifier ( this );
    JTabbedPane tabPane = new JTabbedPane();
    JPanel tabPaneFilter = new JPanel() ;
    JPanel blockPanel = new JPanel() ;
    JPanel sizePanel = new JPanel() ;
    TitledBorder borderBlock;
    TitledBorder borderSize ;
    JCheckBox cbBlockMobile = new JCheckBox();
    JCheckBox cbBlockDesktop = new JCheckBox();
    JCheckBox cbBlockPda = new JCheckBox();
    VerticalFlowLayout verticalFlowLayout1 = new VerticalFlowLayout() ;
    JCheckBox cbFilterMobile = new JCheckBox();
    GridBagLayout sizeGridBagLayout = new GridBagLayout();
    JTextField tfFilterMobile = new JTextField () ;
    JCheckBox cbFilterPda = new JCheckBox();
    JTextField tfFilterPda = new JTextField () ;
    VerticalFlowLayout verticalFlowLayout2 = new VerticalFlowLayout() ;
    JPanel buttonPanel = new JPanel() ;
    JButton buttonOK = new JButton();
    JButton buttonCancel = new JButton() ;
  JPanel tabPaneForward = new JPanel() ;
  ButtonGroup buttonGroupForward = new ButtonGroup();
  JPanel jPanel1 = new JPanel() ;
  JTextField tfSMS = new JTextField () ;
  JRadioButton rbEmail = new JRadioButton();
  JTextField tfEmail = new JTextField () ;
  JRadioButton rbForwardSms = new JRadioButton();
  BorderLayout borderLayout1 = new BorderLayout();
  GridBagLayout gridBagLayout1 = new GridBagLayout();
  TitledBorder titledBorder1 ;
  Component component1;
```

```
JCheckBox jCheckBox1 = new JCheckBox();

  public PreferencesDialog () {
    try {
      jbInit () ;
    }
    catch(Exception e) {
      e. printStackTrace () ;
    }
    setTitle ("Preferences") ;
  }
  private void jbInit () throws Exception {
    borderBlock = new TitledBorder(BorderFactory . createEtchedBorder (Color.white ,new Color
        (148, 145, 140) ) ,"Blocking␣rules") ;
    borderSize = new TitledBorder(BorderFactory . createEtchedBorder (Color.white ,new Color
        (148, 145, 140) ) ,"Size␣filter ") ;
    titledBorder1 = new TitledBorder(BorderFactory . createEtchedBorder (Color.white ,new Color
        (148, 145, 140) ) ,"Forward␣to");
  component1 = Box. createVerticalStrut (8) ;
  tabPaneFilter . setLayout( verticalFlowLayout2 ) ;
    blockPanel . setBorder(borderBlock) ;
    blockPanel . setLayout( verticalFlowLayout1 ) ;
    sizePanel . setBorder ( borderSize ) ;
    sizePanel . setLayout (sizeGridBagLayout);
    cbBlockMobile.setText ("Mobile␣phone");
    cbBlockMobile.addItemListener(new java .awt.event . ItemListener () {
      public void itemStateChanged(ItemEvent e) {
        cbBlockMobile_itemStateChanged(e);
      }
    });
    cbBlockDesktop.setMnemonic('0');
    cbBlockDesktop.setText ("Desktop");
    cbBlockDesktop.addItemListener(new java .awt.event . ItemListener () {
      public void itemStateChanged(ItemEvent e) {
        cbBlockDesktop_itemStateChanged(e);
      }
    });
    cbBlockPda.setText ("PDA");
    cbBlockPda.addItemListener(new java .awt.event . ItemListener () {
      public void itemStateChanged(ItemEvent e) {
        cbBlockPda_itemStateChanged(e);
      }
    });
    cbFilterMobile . setToolTipText ("") ;
    cbFilterMobile . setText ("Mobile␣Phone");
    cbFilterMobile . addItemListener (new java .awt.event . ItemListener () {
      public void itemStateChanged(ItemEvent e) {
        cbFilterMobile_itemStateChanged (e) ;
      }
    });
    tfFilterMobile .setBackground(Color. lightGray ) ;
    tfFilterMobile .setEnabled ( false ) ;
    tfFilterMobile . setToolTipText ("Truncate␣message␣to␣this␣size") ;
    tfFilterMobile . setInputVerifier (iv ) ;
    cbFilterPda . setText ("PDA");
    cbFilterPda . addItemListener (new java .awt.event . ItemListener () {
      public void itemStateChanged(ItemEvent e) {
        cbFilterPda_itemStateChanged (e) ;
      }
    });
    tfFilterPda .setBackground(Color. lightGray ) ;
    tfFilterPda .setEnabled ( false ) ;
```

```
            tfFilterPda . setToolTipText ("Truncate message to this size");
            tfFilterPda . setInputVerifier (iv);
            buttonOK.setText("OK");
            buttonOK.addActionListener(new java.awt.event.ActionListener () {
                public void actionPerformed(ActionEvent e) {
                    buttonOK_actionPerformed(e);
                }
            });
            buttonCancel . setText ("Cancel");
            buttonCancel . addActionListener (new java.awt.event.ActionListener () {
                public void actionPerformed(ActionEvent e) {
                    buttonCancel_actionPerformed (e);
                }
            });
            borderSize . setTitle ("Maximum message size");
        rbEmail. setText ("E-mail");
        tfEmail . setMaximumSize(new Dimension(60, 21));
        tfEmail . setToolTipText ("E-mail address");
        rbForwardSms.setText("SMS");
        jPanel1 . setLayout (gridBagLayout1);
        tabPaneForward.setLayout(borderLayout1);
        tfSMS.setMaximumSize(new Dimension(40, 21));
        tfSMS.setToolTipText("Mobile phone numer");
        tfSMS.setColumns(20);
        jPanel1 . setBorder ( titledBorder1 );
        jCheckBox1.setText("Send copy to desktop client");
        this . getContentPane () . add(tabPane , BorderLayout.CENTER);
            tabPane . add( tabPaneFilter ,          " Filter ");
            tabPaneFilter . add(blockPanel , null);
            blockPanel . add(cbBlockDesktop, null);
            blockPanel . add(cbBlockMobile, null);
            blockPanel . add(cbBlockPda, null);
            tabPaneFilter . add( sizePanel , null);
            sizePanel . add( cbFilterMobile ,          new GridBagConstraints   (0, 0, 1, 1, 0.0, 0.0
                , GridBagConstraints . EAST, GridBagConstraints.NONE, new Insets (5, 5, 0, 5) , 99, 0) );
            sizePanel . add( tfFilterMobile ,        new GridBagConstraints   (0, 1, 1, 1, 0.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (0, 30, 0, 0) , 50, 0) );
            sizePanel . add( cbFilterPda ,     new GridBagConstraints   (0, 2, 1, 1, 0.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (5, 5, 0, 0) , 0, 0) );
            sizePanel . add( tfFilterPda ,      new GridBagConstraints   (0, 3, 1, 1, 0.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (0, 30, 0, 0) , 50, 0) );
            tabPaneFilter . add( buttonPanel , null);
            buttonPanel . add(buttonOK, null);
            buttonPanel . add(buttonCancel , null);
        tabPane . add(tabPaneForward,    "Forward");
        tabPaneForward.add(jPanel1 ,   BorderLayout.NORTH);
        jPanel1 . add(rbForwardSms,      new GridBagConstraints   (0, 0, 1, 1, 0.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (5, 10, 0, 5) , 0, 0) );
        jPanel1 . add(tfSMS,          new GridBagConstraints   (0, 1, 1, 1, 1.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (0, 20, 0, 5) , 100, 0)
                    );
        jPanel1 . add(rbEmail ,        new GridBagConstraints   (0, 2, 1, 1, 0.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (0, 10, 0, 5) , 0, 0) );
        jPanel1 . add( tfEmail ,         new GridBagConstraints   (0, 3, 1, 1, 1.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (0, 20, 0, 5) , 100, 0)
                    );
        buttonGroupForward.add(rbForwardSms);
        buttonGroupForward.add(rbEmail);
        jPanel1 . add(component1, new GridBagConstraints   (0, 4, 1, 1, 0.0, 0.0
                , GridBagConstraints . CENTER, GridBagConstraints.NONE, new Insets (0, 0, 0, 0) , 0, 0) );
        jPanel1 . add(jCheckBox1,     new GridBagConstraints   (0, 5, 1, 1, 0.0, 0.0
                , GridBagConstraints . WEST, GridBagConstraints.NONE, new Insets (0, 10, 0, 0) , 0, 0) );
```

```
    }

    void cbBlockDesktop_itemStateChanged(ItemEvent e) {
      blockDesktop = (e.getStateChange() == ItemEvent.SELECTED);
    }

    void cbBlockMobile_itemStateChanged(ItemEvent e) {
      blockMobile = (e.getStateChange() == ItemEvent.SELECTED);
    }

    void cbBlockPda_itemStateChanged(ItemEvent e) {
      blockPda = (e.getStateChange() == ItemEvent.SELECTED);
    }

    void cbFilterMobile_itemStateChanged(ItemEvent e) {
      boolean selected = (e.getStateChange() == ItemEvent.SELECTED);
      filterMobile = selected;
      tfFilterMobile.setEnabled(selected);
      if (selected){
        tfFilterMobile.setBackground(Color.white);
      } else {
        tfFilterMobile.setBackground(Color.lightGray);
      }
    }

    void cbFilterPda_itemStateChanged(ItemEvent e) {
      boolean selected = (e.getStateChange() == ItemEvent.SELECTED);
      filterPda = selected;
      tfFilterPda.setEnabled(selected);
      if (selected){
        tfFilterPda.setBackground(Color.white);
      } else {
        tfFilterPda.setBackground(Color.lightGray);
      }
    }

    void buttonOK_actionPerformed(ActionEvent e) {
      // Get input
      if (filterMobile)
        filterMobileValue = tfFilterMobile.getText();
      if (filterPda)
        filterPdaValue = tfFilterPda.getText();

      okPressed = true;
      dispose();
    }

    void buttonCancel_actionPerformed(ActionEvent e) {
      dispose();
    }

    // PreferenceInterface implementations
    public boolean getMobileBlock(){
      return blockMobile;
    }

    public boolean getPdaBlock(){
      return blockPda;
    }

    public boolean getDesktopBlock(){
      return blockDesktop;
    }

    public String getMobileMaxSize(){
      if (filterMobile)
        return filterMobileValue;
      else
        return null;
    }

    public String getPdaMaxSize(){
      if (filterPda)
        return filterPdaValue;
      else
        return null;
    }


    // Verifies that input values are integers.
    private class IntegerVerifier extends InputVerifier {

      private Component parent;

      public IntegerVerifier(Component parent){
        this.parent = parent;
      }
      public boolean verify(JComponent input) {
        if (!(input instanceof JTextField))
          return true;
        JTextField tf = (JTextField) input;
        String text = tf.getText();
        if (text.length() == 0)
          return true;
        try{
          int i = Integer.parseInt(text);
          return true;
        } catch (NumberFormatException nfe){
          JOptionPane.showMessageDialog(parent, "Input must be an integer value.");
          return false;
        }
      }
    }

}
```

PreferencesInterface.java

```
package jabberclient ;

public interface PreferencesInterface {

  public boolean getMobileBlock();
  public boolean getPdaBlock();
  public boolean getDesktopBlock();
  public String getMobileMaxSize();
  public String getPdaMaxSize();
}
```

PresenceHandler.java

```
package jabberclient ;

import nanoxml.*;
import java . util .Enumeration;

public class PresenceHandler extends ElementHandler {

  public PresenceHandler(JabberFrame gui , JabberComm comm) {
    super(gui , comm);
  }
  public void handleElement(XMLElement xml) {
    String status = "";
    String jid = null;
    String type = null;
    int device = Contact.DESKTOP;
    int show = Contact.ONLINE;
    jid = trimJid (xml. getProperty ("from"));
    type = xml. getProperty ("type");
    if (type != null && type.equalsIgnoreCase (" unavailable "))
      show = Contact.OFFLINE;
    if (type != null && type.equalsIgnoreCase (" available "))
      show = Contact.ONLINE;

    for(Enumeration children = ( Enumeration)xml.enumerateChildren () ; children .hasMoreElements()
          ; ) {
      XMLElement node = (XMLElement)children.nextElement();
      String tagName = node.getTagName();
      if (tagName.equals(" status ")) {
        status = node. getContents () ;
        if ( status . equalsIgnoreCase ("online "))
          show = Contact.ONLINE;
      }else if (tagName.equals("show")){
        String temp = node. getContents () ;
        if (temp != null){
          if (temp.equalsIgnoreCase ("dnd")){
            show = Contact.DND;
          } else if (temp.equalsIgnoreCase ("away")){
            show = Contact.AWAY;
          } else if (temp.equalsIgnoreCase ("xa")){
            show = Contact.XA;
          } else if (temp.equalsIgnoreCase ("online ")){
            show = Contact.ONLINE;
          }
        }
      } else if (tagName.equals("device ")){
          String temp = node. getContents () ;
          if (temp != null){
            if (temp. equals ("mobile"))
              device = Contact.MOBILE;
            else if (temp. equals ("pda"))
              device = Contact.PDA;
            else if (temp. equals ("desktop")){
              device = Contact.DESKTOP;
            }
          }
      }
    }
    gui . setPresence ( jid , show, status , device ) ;
  }
}
```

TreeRenderer.java

```
package jabberclient;

import javax.swing.tree.*;
import javax.swing.*;
import java.awt.Component;

public class TreeRenderer extends DefaultTreeCellRenderer {

  ImageIcon[][] icons = new ImageIcon[3][3];

  public TreeRenderer() {
    icons[Contact.DESKTOP][Contact.ONLINE] = new ImageIcon("desktop_online.gif");
    icons[Contact.DESKTOP][Contact.OFFLINE] = new ImageIcon("desktop_offline.gif");
    icons[Contact.DESKTOP][Contact.DND] = new ImageIcon("desktop_dnd.gif");

    icons[Contact.MOBILE][Contact.ONLINE] = new ImageIcon("mobile_online.gif");
    icons[Contact.MOBILE][Contact.OFFLINE] = new ImageIcon("mobile_offline.gif");
    icons[Contact.MOBILE][Contact.DND] = new ImageIcon("mobile_dnd.gif");

    icons[Contact.PDA][Contact.ONLINE] = new ImageIcon("pda_online.gif");
    icons[Contact.PDA][Contact.OFFLINE] = new ImageIcon("pda_offline.gif");
    icons[Contact.PDA][Contact.DND] = new ImageIcon("pda_dnd.gif");
  }

  public Component getTreeCellRendererComponent(
                                   JTree tree,
                                   Object value,
                                   boolean sel,
                                   boolean expanded,
                                   boolean leaf,
                                   int row,
                                   boolean hasFocus) {

    super.getTreeCellRendererComponent(
                                   tree, value, sel,
                                   expanded, leaf, row,
                                   hasFocus);
    if (!leaf)
      return super.getTreeCellRendererComponent(
                                   tree, value, sel,
                                   expanded, leaf, row,
                                   hasFocus);

    DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
    Contact contact = (Contact)node.getUserObject();
    setIcon(icons[contact.getDevice()][contact.getShow()]);

    setToolTipText(contact.getJid() + contact.getStatus());

    return this;
  }
}
```

C.4 Server

C.4.1 New classes

ProcessMessage.java

```
package org.ntnu.jabaext.process;

import org.novadeck.jabaserver.jabber.*;
import org.novadeck.jabaserver.users.*;
import org.novadeck.jabaserver.offline.*;
import org.novadeck.jabaserver.core.*;
import java.util.*;

public class ProcessMessage {

  private StreamParser streamparser;

  public ProcessMessage(StreamParser streamparser) {
      this.streamparser = streamparser;
  }

  // Added by Audun


public void handleMessage(Message message){
      // Check forwarding rules and forward
      // Get "best" userstream
      // Check if in contact list
      // Check filter rules
      // If not send to desktop

      String username = (String) streamparser.parameters.get(streamparser.USERNAME_PARAM);
      String to = message.getTo();
      User toUser = streamparser.USER_HOME.findUserByUsername(to);
      User fromUser = streamparser.USER_HOME.findUserByUsername(username);
      message.setTo( null );
      message.setFrom(username);

      if (toUser != null ){
        // Check forwarding rules
        /*
        if (toUser.getForwarding()){
             // forward();
        return;
        }
        */
        if (!toUser.getOnline()){
             sendOfflineMessage(fromUser, to, message);
             return;
        }

        // Deliver
        // Get best stream
        String toResource = message.getToResource();
        UserStream us = toUser.getUserStream(toResource);
        int device = -1;
        if (us.getResource().equalsIgnoreCase("mobile")){
             device = 1;
        } else if (us.getResource().equalsIgnoreCase("pda")){
             device = 2;
        }

        if (device == 1 || device == 2){
        // Check if filter on contacts and user is in contact list

            // Is user in contactlist ?
            if (! streamparser .USER_HOME.getBlockMessage(toUser.getId(), fromUser.getId(), device
                 )){
                System.out. println ("User in contact list :");
                int maxSize = streamparser .USER_HOME.getMaxSize(toUser.getId(), device);
                // If filter , filter send to mobile and dekstop
                if (maxSize != −1 && message.getBody().length() > maxSize)
                {
                      sendToDesktop(fromUser, to , message);
                      message = reduceMessageSize(message, maxSize);
                }
                // send to mobile
                // Check autoreply message
                String autoreply = null;
                if ( streamparser .USER_HOME.getAutoReply(toUser.getId(), device) != null){
                    UserStream fromstream = fromUser.getUserStream(null);
                    if (fromstream != null){
                      Message outmessage = new Message();
                      outmessage.setBody( autoreply );
                      outmessage.setFrom(username);
                      fromstream. postString (outmessage. toString () );
                    }
                }
                // send to mobile
                if (us != null){
                  us. postString (message. toString () );
                }
            }else{
               // No − Send to dekstop
                System.out. println ("Send to desktop:");
                sendToDesktop(fromUser, to , message);
            }

            }else{ // not mobile
                if (! streamparser .USER_HOME.getBlockMessage(toUser.getId(), fromUser.getId(), 0)
                     ){
                    if (us != null) {
                        us. postString (message. toString () );
                    }
                }else{ // Discard message
                }
            }
        }

    }// end handle Message

    public void sendToDesktop(User fromUser, String to , Message message){
        UserStream userstream = getDesktopResource(to);
        if ( userstream != null){
            System.out. println ("Hoyest prioritet : " + userstream .getResource () );
            userstream . postString (message. toString () );
        }else{ // Ingen desktop online
            // Send offline
            System.out. println ("Send offline :");
            sendOfflineMessage(fromUser, to , message);
        }
    }

    public Message reduceMessageSize(Message message, int maxSize){
        String body = message.getBody();
        message.setBody(body.substring (0, maxSize));
        return message;
    }

    public void sendOfflineMessage(User fromUser, String to, Message message){
        User toUser = streamparser.USER_HOME.findUserByUsername(to);
        OfflineMsg offMsg = new OfflineMsg();
        offMsg.setMsg(message.toString());
        offMsg.setToUserId((int)toUser.getId());
        offMsg.setFromUserId((int)fromUser.getId());
        streamparser.OFFLINE_MSG_HOME.storeOfflineMsg(offMsg);
    }

    public UserStream getDesktopResource(String username){

        UserStream us = null;
        User user = streamparser.USER_HOME.findUserByUsername(username);
        Integer pri = new Integer(-1);
        Enumeration enum = user.resource.keys();
        while(enum.hasMoreElements()){
            Integer priority = (Integer)enum.nextElement();
            us = user.getUserStream(priority.intValue());
            if (!(us.getResource().equalsIgnoreCase("mobile") || us.getResource().
                    equalsIgnoreCase("pda"))){
                if (pri.compareTo(priority) < 0){
                    pri = priority;
                }
            }
        }

        if (pri.compareTo(new Integer(-1)) > 0){
            us = user.getUserStream(pri.intValue());
            return us;
        }else{
            return null;
        }
    }//end getDesktopResource
}
```

ProcessPresence.java

```java
package org.ntnu.jabaext.process;

import java.util.*;
import org.novadeck.jabaserver.core.*;
import org.novadeck.jabaserver.jabber.*;
import org.novadeck.jabaserver.users.*;


public class ProcessPresence {

  private StreamParser streamparser;

  public ProcessPresence(StreamParser streamparser) {

      this.streamparser = streamparser;
  }

  public void handlePresence(Presence presence){

     System.out.println("handlePresence");


      if(presence.getPriority() != null){  //Added by Audun

         String username = (String) streamparser.parameters.get(streamparser.USERNAME_PARAM
               );
         System.out.print(username);
         UserStream stream = (UserStream)streamparser.parameters.get(streamparser.
               USERSTREAM_PARAM);
         String resource = stream.getResource();
         System.out.print(resource);
         User userObj = streamparser.USER_HOME.findUserByUsername(username);
         Integer pri = new Integer(presence.getPriority());
         //userObj.replaceUserStream(stream, pri.intValue());
      }

      if(presence.getTo() != null){
          return;
      }

         //For each friend
           //Get stream
             //If in contactlist
               //Post message
         User user = (User)streamparser.parameters.get(streamparser.USER_PARAM);
         List onlineFriend = user.getOnlineFriends();
         Iterator iter = onlineFriend.iterator();

         for( int i = 0; i<onlineFriend.size(); i++){
             User friend = (User) iter.next();
             String username = (String) streamparser.parameters.get(streamparser.
                   USERNAME_PARAM);
             presence.setFrom(username);
             UserStream[] us = friend.getUserStreams();
             if( us != null ){
                 for(int j = 0; j<us.length; j++){
                     String r = us[j].getResource();
                     //if in contactlist
                     us[j].postString(presence.toString());
                 }
```

```
                }
            }
        }//end handle presence


}
```

SQPreferences.java

```
package org.ntnu.jabaext.jabber;

import org.novadeck.jabaserver.jabber.Element;
import org.xml.sax.*;

public class SQPreferences extends Element {

  public static final String ELEMENT_NAME = "";

   private static final String MOBILE_ELEMENT = "mobile";
   private static final String PDA_ELEMENT = "pda";
   private static final String DESKTOP_ELEMENT = "desktop";
   private static final String BLOCK_ELEMENT = "block";
   private static final String SIZE_ELEMENT = "size";
   private static final String MAX_SIZE_ATTRIBUTE = "max";

   private String currentDeviceElement = null;

   private boolean blockMobile = false;
   private boolean blockDesktop = false;
   private boolean blockPda = false;

   private boolean filterMobile = false;
   private String filterMobileValue = null;
   private boolean filterPda = false;
   private String filterPdaValue = null;

   public SQPreferences() {
        super("");
   }

   public boolean getBlockMobile(){
        return blockMobile;
   }

   public boolean getBlockPda(){
        return blockPda;
   }

   public boolean getBlockDesktop(){
        return blockDesktop;
   }

   public String getFilterMobile (){
      if ( filterMobile )
           return filterMobileValue ;
      else
           return null;
   }

   public String getFilterPda (){
      if ( filterPda )
           return filterPdaValue ;
      else
           return null;
   }

   public void startElement ( String namespaceURI, String elementName, String qName, Attributes
             attributes ) throws SAXException{
        if (elementName.equals(DESKTOP_ELEMENT)){
            currentDeviceElement = DESKTOP_ELEMENT;
        } else if (elementName.equals(MOBILE_ELEMENT)){
            currentDeviceElement = MOBILE_ELEMENT;
        } else if (elementName.equals(PDA_ELEMENT)){
            currentDeviceElement = PDA_ELEMENT;
        }

        if (elementName.equals(BLOCK_ELEMENT)){
            if (currentDeviceElement . equals (DESKTOP_ELEMENT))
                blockDesktop = true;
            else if ( currentDeviceElement . equals (PDA_ELEMENT))
                blockPda = true;
            else if ( currentDeviceElement . equals (MOBILE_ELEMENT))
                blockMobile = true;
        } else if (elementName.equals(SIZE_ELEMENT)){
            if ( currentDeviceElement . equals (PDA_ELEMENT)){
                filterPda = true;
                filterPdaValue = attributes . getValue (MAX_SIZE_ATTRIBUTE);
            } else if ( currentDeviceElement . equals (MOBILE_ELEMENT)){
                filterMobile = true;
                filterMobileValue = attributes . getValue (MAX_SIZE_ATTRIBUTE);
            }
        }
    }

    public void endElement( String namespaceURI, String elementName, String qName ) throws
            SAXException{
        if (elementName.equals(DESKTOP_ELEMENT)){
            currentDeviceElement = null;
        } else if (elementName.equals(MOBILE_ELEMENT)){
            currentDeviceElement = null;
        } else if (elementName.equals(PDA_ELEMENT)){
            currentDeviceElement = null;
        }
    }
}
```

SQRosterConfig.java

```
package org.ntnu.jabaext.jabber;

import org.novadeck.jabaserver.jabber.Element;
import org.novadeck.jabaserver.jabber.SQRosterItem;
import org.xml.sax.*;

import java.util.Vector;
import java.util.List;

public class SQRosterConfig extends Element {

  public static final String ELEMENT_NAME = "";
  private static final String ITEM_ELEMENT = "item";

  private static final String MOBILE_ELEMENT = "mobile";
  private static final String PDA_ELEMENT = "pda";

  private String currentDeviceElement = null;

  private Vector pdaContacts = new Vector();
  private Vector mobileContacts = new Vector();

  public SQRosterConfig() {
    this(ELEMENT_NAME);
  }

  public SQRosterConfig(String elementName){
    super(elementName);
  }

  public void startElement ( String namespaceURI, String elementName, String qName, Attributes
        attributes ) throws SAXException{
    if ( child != null ){
        child . startElement ( namespaceURI, elementName, qName, attributes ) ;
        return;
    }

    if ( elementName.equals( ITEM_ELEMENT )){
        SQRosterItem sqRosterItem = new SQRosterItem();
        child = sqRosterItem;
        setupLogger( child );
        sqRosterItem . startElement ( namespaceURI, elementName, qName, attributes ) ;
    } else if (elementName.equals(MOBILE_ELEMENT)){
      currentDeviceElement = MOBILE_ELEMENT;
    } else if (elementName.equals(PDA_ELEMENT)){
      currentDeviceElement = PDA_ELEMENT;
    }
  }

    public void endElement( String namespaceURI, String elementName, String qName ) throws
        SAXException{
        if ( child != null ){
            child . endElement( namespaceURI, elementName, qName );
        }
        if (elementName.equals(ITEM_ELEMENT)){
            addItem( child );
            child = null;
            return;
        } else if (elementName.equals(MOBILE_ELEMENT)){
            currentDeviceElement = null;
        } else if (elementName.equals(PDA_ELEMENT)){
            currentDeviceElement = null;
        }
    }

    private void addItem(Element e){
        SQRosterItem item = (SQRosterItem)e;
        if (currentDeviceElement.equals(MOBILE_ELEMENT)){
            mobileContacts.add(item.getJid());
        } else if (currentDeviceElement.equals(PDA_ELEMENT)){
            pdaContacts.add(item.getJid());
        }
    }

    public List getMobileContacts(){
        return (List)mobileContacts;
    }

    public List getPdaContacts(){
        return (List)pdaContacts;
    }
}
```

C.4.2 Modified classes

UserHomeDB.java

```
/*
 * Copyright - JabaServer Project 2001 - Alexis Agahi
 */

package org.novadeck.jabaserver.users;

import org.novadeck.jabaserver.jabber.SQRosterItem;
import java.util.*;
import java.sql.*;
import org.apache.avalon.framework.logger.*;
import org.apache.avalon.framework.configuration.*;
import org.apache.avalon.framework.activity.*;

public class UserHomeDB extends AbstractLogEnabled implements UserHome, Configurable,
        Initializable {

    /* SIGN change */
    private static String FIELD_ROSTER_DEVICE_ID;
    private static String TABLE_PREFERENCES;
    private static String FIELD_PREF_USERS_ID;
    private static String FIELD_PREF_DEVICE_ID;
    private static String FIELD_PREF_RULE;
    private static String FIELD_PREF_PARAM;
    /* End SIGN change */

    /* SIGN change (~25 LOC)*/
    /**
     * A method to retreice device-specific rosters. The device id is 1 for
     * mobile phones and 2 for PDAs
     *
     * @param userId The users id in the datatbase
     * @param deviceId Device id to indicate which roster to retreive
     * @return List containing SQRosterItem objects
     */
    public List getRosterListById(long userId, int deviceId){
        List list = new ArrayList();
        Connection connection = null;

        try{
            connection = getConnection();
            String query = "SELECT "+TABLE_USERS+". * FROM " + TABLE_USERS+','+
                    TABLE_ROSTER +
                            " WHERE " + TABLE_ROSTER+'.'+FIELD_ROSTER_USERS_ID + "=? "+
                            " AND " + TABLE_ROSTER+'.'+FIELD_ROSTER_FRIEND_ID +'='+
                                    TABLE_USERS+'.'+FIELD_USER_ID +
                            " AND " + TABLE_ROSTER+'.' + FIELD_ROSTER_DEVICE_ID + "=?";
            PreparedStatement pst = connection.prepareStatement(query);
            pst.setLong(1, userId);
            pst.setInt(2, deviceId);

            ResultSet rs = pst.executeQuery();
            while(rs.next()){
                SQRosterItem roster = new SQRosterItem();
                String domain = findDomainNameById( rs.getLong( FIELD_USER_DOMAIN_ID ));
                roster.setJid( rs.getString( FIELD_USER_LOGIN )+'@'+ domain );
                roster.setName( rs.getString( FIELD_USER_NAME ));
                roster.setSubscription( "both" );
                roster.setGroup( "friends" );
                list .add( roster ) ;
            }
            rs . close () ;
            pst . close () ;

        } catch(SQLException e){
            e . printStackTrace () ;
        }
        return list ;
    }
    /* End SIGN change */

    /** SIGN change (˜30 LOC)
     * A method to store a device specific contact list . The list does not contain
     * any new roster items .
     *
     * The method uses a delete and insert policy : All roster items with the
     * specified user− and device id are deleted . Then roster items from the
     * list are inserted .
     *
     * @author Svein
     * @param userId The id of the user
     * @param roster List of jids
     * @param deviceId The id of the device to store the list for
     */
    public void setRosterList (long userId , int deviceId , List roster ){
        Connection connection = null ;

        String deleteQuery = ”DELETE FROM ” + TABLE ROSTER + ” WHERE ” +
                    FIELD ROSTER USERS ID + ”=? AND ” +
                    FIELD ROSTER DEVICE ID + ”=?”;

        String insertQuery = ”INSERT INTO ” + TABLE ROSTER + ” VALUES(?,?,?)”;

        try{
            connection = getConnection () ;

            // Delete current roster
            PreparedStatement pst = connection . prepareStatement (deleteQuery ) ;
            pst .setLong (1, userId ) ;
            pst . setInt (2, deviceId ) ;
            ResultSet rs = pst . executeQuery () ;
            pst . close () ;
            rs . close () ;

            // Insert new items
            long friendId = 0;
            Iterator iter = roster . iterator () ;
            while ( iter .hasNext() ) {
                String jid = ( String ) iter . next () ;
                User friend = findUserByUsername(jid);
                if ( friend != null ){
                    friendId = friend . getId () ;
                    pst = connection . prepareStatement ( insertQuery ) ;
                    pst .setLong (1, userId ) ;
                    pst .setLong (2, friendId ) ;
                    pst . setInt (3, deviceId ) ;

                    rs = pst . executeQuery () ;
                    pst . close () ;
                    rs . close () ;

                }
            }
        } catch(Exception e){
            System.out.println("DB exception in store UserHomeDB::storeRosterList()");
        }
    }

    /** SIGN change (~20 LOC)
     *
     * @param userId The user id
     * @param deviceId The device id 0=desktop, 1=mobile, 2=pda
     * @param preference Either 'filter' or 'block' or 'forward'
     * @param value Max msg. lengt for 'filter', 'sms' or 'mail' for forward
     *              null otherwise
     */
    public void setPreference(long userId, int deviceId, String preference, String value){
      Connection connection = null;
      try{
        connection = getConnection();

        String insertQuery = "INSERT INTO " + TABLE_PREFERENCES + " VALUES(?,?,?,?)";
        PreparedStatement pst = connection.prepareStatement(insertQuery);
        pst.setLong(1, userId);
        pst.setInt(2, deviceId);
        pst.setString(3, preference);
        if(value == null)
          pst.setString(4, "(null)");
        else
          pst.setString(4, value);
        ResultSet rs = pst.executeQuery();
        rs.close();
        pst.close();
      } catch(Exception e){
          System.out.println("DB exception in store UserHomeDB::storeRosterList()\n");
          System.out.println(e.getMessage());
      }
    }
    /* End SIGN change */


    /* SIGN change (~10 LOC) */
    public void deletePreferences(long userId){
      Connection connection = null;
      try{
        connection = getConnection();
        String query = "DELETE FROM " + TABLE_PREFERENCES + " WHERE " +
                     FIELD_ROSTER_USERS_ID+"=?";
        PreparedStatement pst = connection.prepareStatement(query);
        pst.setLong(1, userId);
        ResultSet rs = pst.executeQuery();
        rs.close();
        pst.close();
      } catch(Exception e){
        System.out.println("Exception in deletePreferences():\n");
        System.out.println(e.getMessage());
      }
    }
    /* End SIGN change */

    /* SIGN change (~15 LOC) */
    private boolean isUserInContactList(long userId, int deviceId){
        System.out.println("Userid: " + userId + " Device: " + deviceId);
        Connection connection = null;
        boolean result = false;
        try{
            connection = getConnection();

            String query = "SELECT * FROM " + TABLE_ROSTER + " WHERE " +
                           FIELD_ROSTER_USERS_ID + "=? AND " +
                                FIELD_ROSTER_DEVICE_ID +
                           "=?";
            PreparedStatement pst = connection.prepareStatement(query);
            pst.setLong(1, userId);
            pst.setInt(2, deviceId);

            ResultSet rs = pst.executeQuery();
            result = rs.next();
            rs.close();
            pst.close();
            System.out.println("isUser ... " + result);
            return result;
        } catch (Exception e){
          System.out.println("Exception in isUserInContactList()\n");
          System.out.println(e.getMessage());
          return false;
        }
    }
    /* End SIGN change */

    /* SIGN change (~20 LOC) */
    public boolean getBlockMessage(long userId, long friendId, int deviceId){
        Connection connection = null;
        boolean doFilter = false;
        try{
          connection = getConnection();
          String query = "SELECT * FROM " + TABLE_PREFERENCES + " WHERE " +
                         FIELD_PREF_USERS_ID + "=? AND " + FIELD_PREF_DEVICE_ID +
                         "=? AND " + FIELD_PREF_RULE + "='block'";
          PreparedStatement pst = connection.prepareStatement(query);
          pst.setLong(1, userId);
          pst.setInt(2, deviceId);
          ResultSet rs = pst.executeQuery();

          doFilter = rs.next();
          rs.close();
          pst.close();

          if( doFilter )
            return ! isUserInContactList(friendId, deviceId);

        } catch (Exception e){
            System.out.println("Exception in doBlockMessage()\n");
            System.out.println(e.getMessage());
        }
        System.out.println("Block not set: " + userId + " " + deviceId);
        return false;
    }
    /* SIGN change */

    /* SIGN change (~20 LOC) */
    public int getMaxSize(long userId, int deviceId){
        Connection conn;
        int result = -1;
        try{
            conn = getConnection();
            String query = "SELECT parameter FROM " + TABLE_PREFERENCES + " WHERE "
                 +
                       FIELD_PREF_USERS_ID +"=? AND " + FIELD_PREF_DEVICE_ID +
                       "=? AND " + FIELD_PREF_RULE + "=?";
            PreparedStatement pst = conn.prepareStatement(query);
            pst.setLong(1, userId);
            pst.setInt(2, deviceId);
            pst.setString(3, "size");

            ResultSet rs = pst.executeQuery();
            if(rs.next()){
               result = rs.getInt("parameter");
            }
            rs.close();
            pst.close();
        } catch (Exception e){
           System.out.println("Exception in getMaxSize()\n");
           System.out.println(e.getMessage());
        }
        return result;
    }
    /* End SIGN change */

    /* SIGN change (~20 LOC)*/
    public String getForward(long userId){
        Connection conn = null;
        String result = null;
        try{
            conn = getConnection();
            String query = "SELECT parameter FROM " + TABLE_PREFERENCES + " WHERE "
                 +
                       FIELD_PREF_USERS_ID + "=? AND " + FIELD_PREF_RULE + "=?";
            PreparedStatement pst = conn.prepareStatement(query);
            pst.setLong(1, userId);
            pst.setString(2, "forward");

            ResultSet rs = pst.executeQuery();
            if(rs.next()){
               result = rs.getString("parameter");
            }
            rs.close();
            pst.close();

        } catch(Exception e){
           System.out.println("Exception in getForward()\n");
           System.out.println(e.getMessage());
        }
        return result;
    }

    /* SIGN change (~20 LOC) */
    public String getAutoReply(long userId, int deviceId){
      Connection conn = null;
      String result = null;
      try {
        conn = getConnection();
        String query = "SELECT parameter FROM " + TABLE_PREFERENCES + " WHERE " +
                   FIELD_PREF_USERS_ID + "=? AND " + FIELD_PREF_DEVICE_ID + "=?" +
                   " AND " + FIELD_PREF_RULE + "=?";
        PreparedStatement pst = conn.prepareStatement(query);
        pst.setLong(1, userId);
```

```
            pst . setInt  (2,  deviceId ) ;
            pst . setString  (3,  ”forward”) ;

            ResultSet   rs  =  pst . executeQuery () ;
            if ( rs . next () ){
               result   =  rs . getString (”parameter”) ;
            }
            rs . close () ;
            pst . close () ;

         } catch  ( Exception  e)  {
            System.out . println (”Exception_in_getAutoReply()\n”);
            System.out . println (e . getMessage()) ;
         }
         return   result ;
      }
      /∗ End SIGN ∗/


      public  void    initialize   (){

            if  ( ! INITIALIZED ){

                  try {
                        Configuration   conf  =  CONFIGURATION.getChild(”user−db”);
                        DATABASE_DRIVER = conf.getChild (”driver ”) . getValue () ;
                        DATABASE_URL       = conf . getChild (”url ”) . getValue () ;
                        DATABASE_LOGIN  = conf . getChild (”login ”) . getValue () ;
                        DATABASE_PASSWORD = conf.getChild(”password”).getValue( ””);
                        LOGIN_DOMAIN       = conf . getChild (”login −domain”).getValue();

                        Configuration   confUsers  =  conf . getChild (”tables ”) . getChild (”users ”) ;
                        TABLE_USERS           = confUsers . getChild (”name”).getValue () ;
                        FIELD_USER_ID         = confUsers . getChild (” fields ”) . getChild (”id”) . getValue () ;
                        FIELD_USER_NAME = confUsers.getChild (” fields ”) . getChild (”name”).getValue () ;
                        FIELD_USER_LOGIN = confUsers.getChild(” fields ”) . getChild (”login ”) . getValue () ;
                        FIELD_USER_PASSWORD = confUsers.getChild(”fields”).getChild(”password”).
                              getValue();
                        FIELD_USER_DOMAIN_ID = confUsers.getChild(”fields”).getChild(”domain−id”).
                              getValue();

                        Configuration   confDomains = conf . getChild (”tables ”) . getChild (”domains”);
                        TABLE_DOMAINS    = confDomains.getChild(”name”).getValue () ;
                        FIELD_DOMAIN_ID = confDomains.getChild(” fields ”) . getChild (”id”) . getValue () ;
                        FIELD_DOMAIN_NAME = confDomains.getChild(”fields”).getChild(”name”).getValue
                              ();

                        Configuration   confRoster  =  conf . getChild (”tables ”) . getChild (”roster ”) ;
                        TABLE_ROSTER        =    confRoster . getChild (”name”).getValue () ;
                        FIELD_ROSTER_USERS_ID = confRoster.getChild(”fields”).getChild(”user−id”).
                              getValue () ;
                        FIELD_ROSTER_FRIEND_ID = confRoster.getChild(”fields”).getChild(”friend−id”).
                              getValue () ;

                        /∗ SIGN Change (˜10 LOC) ∗/
                        FIELD_ROSTER_DEVICE_ID = confRoster.getChild(”fields”).getChild(”device−id”).
                              getValue() ;

                        Configuration    confPreferences   =  conf . getChild (”tables ”) . getChild (”preferences ”) ;
                        TABLE_PREFERENCES = confPreferences.getChild(”name”).getValue();
                        FIELD_PREF_USERS_ID = confPreferences.getChild(”fields”). getChild (”user−id”).
                              getValue () ;
```

```
                    FIELD_PREF_DEVICE_ID = confPreferences.getChild("fields").getChild("device-id")
                        .getValue();
                    FIELD_PREF_RULE = confPreferences.getChild("fields").getChild("rule").getValue()
                        ;
                    FIELD_PREF_PARAM = confPreferences.getChild("fields").getChild("parameter").
                        getValue();
                    /* End SIGN Change */

                } catch ( ConfigurationException ce ){
                    ce.printStackTrace();
                }

                INITIALIZED = true;
            }
        }

    }
```

StreamParser.java

```
/*
 * Copyright − JabaServer Project 2001 − Alexis Agahi
 */

package org.novadeck.jabaserver.users;

import org.novadeck.jabaserver.jabber.SQRosterItem;
import java.util.*;
import java.sql.*;
import org.apache.avalon.framework.logger.*;
import org.apache.avalon.framework.configuration.*;
import org.apache.avalon.framework.activity.*;

public class UserHomeDB extends AbstractLogEnabled implements UserHome, Configurable,
        Initializable {

    /* SIGN change */
    private static String FIELD_ROSTER_DEVICE_ID;
    private static String TABLE_PREFERENCES;
    private static String FIELD_PREF_USERS_ID;
    private static String FIELD_PREF_DEVICE_ID;
    private static String FIELD_PREF_RULE;
    private static String FIELD_PREF_PARAM;
    /* End SIGN change */

    /* SIGN change (~25 LOC)*/
    /**
     * A method to retreice device−specific rosters. The device id is 1 for
     * mobile phones and 2 for PDAs
     *
     * @param userId The users id in the datatbase
     * @param deviceId Device id to indicate which roster to retreive
     * @return List containing SQRosterItem objects
     */
    public List getRosterListById(long userId, int deviceId){
        List list = new ArrayList();
        Connection connection = null;

        try{
            connection = getConnection();
            String query = "SELECT "+TABLE_USERS+".* FROM "+ TABLE_USERS+','+
                    TABLE_ROSTER +
                            " WHERE "+ TABLE_ROSTER+'.'+FIELD_ROSTER_USERS_ID + "=? "+
                            " AND "+ TABLE_ROSTER+'.'+FIELD_ROSTER_FRIEND_ID +'='+
                                    TABLE_USERS+'.'+FIELD_USER_ID +
                            " AND "+ TABLE_ROSTER+'.' + FIELD_ROSTER_DEVICE_ID + "=?";
            PreparedStatement pst = connection.prepareStatement(query);
            pst.setLong(1, userId);
            pst.setInt(2, deviceId);

            ResultSet rs = pst.executeQuery();
            while(rs.next()){
                SQRosterItem roster = new SQRosterItem();
                String domain = findDomainNameById(rs.getLong(FIELD_USER_DOMAIN_ID));
                roster.setJid(rs.getString(FIELD_USER_LOGIN)+'@'+ domain);
                roster.setName(rs.getString(FIELD_USER_NAME));
                roster.setSubscription("both");
                roster.setGroup("friends");
                list.add(roster);
            }
            rs.close();
            pst.close();

        } catch(SQLException e){
            e.printStackTrace();
        }
        return list;
    }
    /* End SIGN change */

    /** SIGN change (~30 LOC)
     * A method to store a device specific contact list. The list does not contain
     * any new roster items.
     *
     * The method uses a delete and insert policy: All roster items with the
     * specified user- and device id are deleted. Then roster items from the
     * list are inserted.
     *
     * @author Svein
     * @param userId The id of the user
     * @param roster List of jids
     * @param deviceId The id of the device to store the list for
     */
    public void setRosterList(long userId, int deviceId, List roster){
        Connection connection = null;

        String deleteQuery = "DELETE FROM " + TABLE_ROSTER + " WHERE " +
                          FIELD_ROSTER_USERS_ID + "=? AND " +
                          FIELD_ROSTER_DEVICE_ID + "=?";

        String insertQuery = "INSERT INTO " + TABLE_ROSTER + " VALUES(?,?,?)";

        try{
            connection = getConnection();

            // Delete current roster
            PreparedStatement pst = connection.prepareStatement(deleteQuery);
            pst.setLong(1, userId);
            pst.setInt(2, deviceId);
            ResultSet rs = pst.executeQuery();
            pst.close();
            rs.close();

            // Insert new items
            long friendId = 0;
            Iterator iter = roster.iterator();
            while (iter.hasNext()) {
                String jid = (String) iter.next();
                User friend = findUserByUsername(jid);
                if (friend != null){
                    friendId = friend.getId();
                    pst = connection.prepareStatement(insertQuery);
                    pst.setLong(1, userId);
                    pst.setLong(2, friendId);
                    pst.setInt(3, deviceId);

                    rs = pst.executeQuery();
                    pst.close();
                    rs.close();

                }
            }
        } catch(Exception e){
            System.out. println ("DB exception in store UserHomeDB::storeRosterList()");
        }
    }

    /** SIGN change (~20 LOC)
     *
     * @param userId The user id
     * @param deviceId The device id 0=desktop, 1=mobile, 2=pda
     * @param preference Either 'filter' or 'block' or 'forward'
     * @param value Max msg. lengt for 'filter', 'sms' or 'mail' for forward
     *              null otherwise
     */
    public void setPreference (long userId , int deviceId , String preference , String value){
      Connection connection = null;
      try{
        connection = getConnection();

        String insertQuery = "INSERT INTO " + TABLE_PREFERENCES + " VALUES(?,?,?,?)";
        PreparedStatement pst = connection.prepareStatement(insertQuery);
        pst.setLong(1, userId);
        pst.setInt(2, deviceId);
        pst.setString(3, preference);
        if (value == null)
          pst.setString(4, "(null)");
        else
          pst.setString(4, value);
        ResultSet rs = pst.executeQuery();
        rs.close();
        pst.close();
      } catch(Exception e){
          System.out. println ("DB exception in store UserHomeDB::storeRosterList()\n");
          System.out. println (e.getMessage());
      }
    }
    /* End SIGN change */


    /* SIGN change (~10 LOC) */
    public void deletePreferences (long userId){
      Connection connection = null;
      try{
        connection = getConnection();
        String query = "DELETE FROM " + TABLE_PREFERENCES + " WHERE " +
                        FIELD_ROSTER_USERS_ID+"=?";
        PreparedStatement pst = connection.prepareStatement(query);
        pst.setLong(1, userId);
        ResultSet rs = pst.executeQuery();
        rs.close();
        pst.close();
      } catch(Exception e){
        System.out. println ("Exception in deletePreferences ():\n");
        System.out. println (e.getMessage());
      }
    }
    /* End SIGN change */

    /* SIGN change (~15 LOC) */
    private boolean isUserInContactList (long userId , int deviceId){
        System.out. println ("Userid: " + userId + " Device: " + deviceId);
        Connection connection = null;
        boolean result = false;
        try{
            connection = getConnection();

            String query = "SELECT * FROM " + TABLE_ROSTER + " WHERE " +
                                FIELD_ROSTER_USERS_ID + "=? AND " +
                                    FIELD_ROSTER_DEVICE_ID +
                                "=?";
            PreparedStatement pst = connection.prepareStatement(query);
            pst.setLong(1, userId);
            pst.setInt(2, deviceId);

            ResultSet rs = pst.executeQuery();
            result = rs.next();
            rs.close();
            pst.close();
            System.out.println("isUser ... " + result);
            return result;
        } catch (Exception e){
          System.out.println("Exception in isUserInContactList()\n");
          System.out.println(e.getMessage());
          return false;
        }
    }
    /* End SIGN change */

    /* SIGN change (~20 LOC) */
    public boolean getBlockMessage(long userId, long friendId, int deviceId){
        Connection connection = null;
        boolean doFilter = false;
        try{
          connection = getConnection();
          String query = "SELECT * FROM " + TABLE_PREFERENCES + " WHERE " +
                            FIELD_PREF_USERS_ID + "=? AND " + FIELD_PREF_DEVICE_ID +
                            "=? AND " + FIELD_PREF_RULE + "='block'";
          PreparedStatement pst = connection.prepareStatement(query);
          pst.setLong(1, userId);
          pst.setInt(2, deviceId);
          ResultSet rs = pst.executeQuery();

          doFilter = rs.next();
          rs.close();
          pst.close();

          if (doFilter)
            return !isUserInContactList(friendId, deviceId);

        } catch (Exception e){
            System.out.println("Exception in doBlockMessage()\n");
            System.out.println(e.getMessage());
        }
        System.out.println("Block not set: " + userId + " " + deviceId);
        return false;
    }
    /* SIGN change */

    /* SIGN change (~20 LOC) */
    public int getMaxSize(long userId, int deviceId){
        Connection conn;
        int result = -1;
        try{
            conn = getConnection();
```

```
                String query = "SELECT parameter FROM " + TABLE_PREFERENCES + " WHERE "
                    +
                                FIELD_PREF_USERS_ID +"=? AND " + FIELD_PREF_DEVICE_ID +
                                "=? AND " + FIELD_PREF_RULE + "=?";
                PreparedStatement pst = conn.prepareStatement(query);
                pst.setLong(1, userId);
                pst.setInt(2, deviceId);
                pst.setString(3, "size");

                ResultSet rs = pst.executeQuery();
                if(rs.next()){
                    result = rs.getInt("parameter");
                }
                rs.close();
                pst.close();
            } catch (Exception e){
                System.out.println("Exception in getMaxSize()\n");
                System.out.println(e.getMessage());
            }
            return result;
        }
        /* End SIGN change */

        /* SIGN change (~20 LOC)*/
        public String getForward(long userId){
            Connection conn = null;
            String result = null;
            try{
                conn = getConnection();
                String query = "SELECT parameter FROM " + TABLE_PREFERENCES + " WHERE "
                    +
                                FIELD_PREF_USERS_ID + "=? AND " + FIELD_PREF_RULE + "=?";
                PreparedStatement pst = conn.prepareStatement(query);
                pst.setLong(1, userId);
                pst.setString(2, "forward");

                ResultSet rs = pst.executeQuery();
                if(rs.next()){
                    result = rs.getString("parameter");
                }
                rs.close();
                pst.close();

            } catch(Exception e){
                System.out.println("Exception in getForward()\n");
                System.out.println(e.getMessage());
            }
            return result;
        }

        /* SIGN change (~20 LOC) */
        public String getAutoReply(long userId, int deviceId){
          Connection conn = null;
          String result = null;
          try {
            conn = getConnection();
            String query = "SELECT parameter FROM " + TABLE_PREFERENCES + " WHERE " +
                              FIELD_PREF_USERS_ID + "=? AND " + FIELD_PREF_DEVICE_ID + "=?" +
                              " AND " + FIELD_PREF_RULE + "=?";
            PreparedStatement pst = conn.prepareStatement(query);
            pst.setLong(1, userId);
            pst.setInt(2, deviceId);
            pst . setString (3, "forward");

            ResultSet rs = pst.executeQuery();
            if (rs . next () ){
               result = rs . getString ("parameter");
            }
            rs . close () ;
            pst . close () ;

         } catch (Exception e) {
            System.out. println ("Exception in getAutoReply()\n");
            System.out. println (e.getMessage());
         }
         return result ;
      }
      /* End SIGN */


      public void initialize (){

            if ( ! INITIALIZED ){

                  try {
                        Configuration conf = CONFIGURATION.getChild("user−db");
                        DATABASE_DRIVER = conf.getChild ("driver ") . getValue () ;
                        DATABASE_URL     = conf . getChild ("url ") . getValue () ;
                        DATABASE_LOGIN  = conf . getChild ("login ") . getValue () ;
                        DATABASE_PASSWORD = conf.getChild("password").getValue( "" );
                        LOGIN_DOMAIN     = conf . getChild ("login−domain").getValue();

                        Configuration confUsers = conf . getChild ("tables ") . getChild ("users ") ;
                        TABLE_USERS        = confUsers . getChild ("name").getValue () ;
                        FIELD_USER_ID      = confUsers . getChild (" fields ") . getChild ("id") . getValue () ;
                        FIELD_USER_NAME = confUsers.getChild (" fields ") . getChild ("name").getValue () ;
                        FIELD_USER_LOGIN = confUsers.getChild(" fields ") . getChild ("login ") . getValue () ;
                        FIELD_USER_PASSWORD = confUsers.getChild("fields").getChild("password").
                              getValue();
                        FIELD_USER_DOMAIN_ID = confUsers.getChild("fields").getChild("domain−id").
                              getValue();

                        Configuration confDomains = conf.getChild ("tables ") . getChild ("domains");
                        TABLE_DOMAINS   = confDomains.getChild("name").getValue () ;
                        FIELD_DOMAIN_ID = confDomains.getChild(" fields ") . getChild ("id") . getValue () ;
                        FIELD_DOMAIN_NAME = confDomains.getChild("fields").getChild("name").getValue
                              ();

                        Configuration confRoster = conf . getChild ("tables ") . getChild ("roster ") ;
                        TABLE_ROSTER      =  confRoster . getChild ("name").getValue () ;
                        FIELD_ROSTER_USERS_ID = confRoster.getChild("fields").getChild("user−id").
                              getValue() ;
                        FIELD_ROSTER_FRIEND_ID = confRoster.getChild("fields").getChild("friend−id").
                              getValue () ;

                        /* SIGN Change (~10 LOC) */
                        FIELD_ROSTER_DEVICE_ID = confRoster.getChild("fields").getChild("device−id").
                              getValue();

                        Configuration confPreferences = conf . getChild ("tables ") . getChild ("preferences ") ;
                        TABLE_PREFERENCES = confPreferences.getChild("name").getValue();
                        FIELD_PREF_USERS_ID = confPreferences.getChild("fields"). getChild ("user−id").
                              getValue () ;
```

```
                FIELD_PREF_DEVICE_ID = confPreferences.getChild("fields").getChild("device-id")
                    .getValue();
                FIELD_PREF_RULE = confPreferences.getChild("fields").getChild("rule").getValue()
                    ;
                FIELD_PREF_PARAM = confPreferences.getChild("fields").getChild("parameter").
                    getValue();
                /* End SIGN Change */

            } catch (ConfigurationException ce) {
                ce.printStackTrace();
            }

            INITIALIZED = true;
        }
    }

}
```

Query.java

```
/*
 * Copyright - JabaServer Project 2001 - Alexis Agahi
 */

package org.novadeck.jabaserver.jabber;

import org.xml.sax.*;
import org.apache.avalon.framework.logger.*;

/* SIGN change */
import org.ntnu.jabaext.jabber.SQRosterConfig;
import org.ntnu.jabaext.jabber.SQPreferences;
import org.ntnu.jabaext.process.ProcessIqExt;
/* End SIGN change */

public class Query extends Element{

    public static final String ELEMENT_NAME = "query";
    public static final String NAMESPACE_ATTRIB = "xmlns";
    public static String NAMESPACE_AUTH = "jabber:iq:auth";
    public static String NAMESPACE_AGENTS = "jabber:iq:agents";
    public static String NAMESPACE_ROSTER = "jabber:iq:roster";
    public static String NAMESPACE_OOB = "jabber:iq:oob";


    String namespace;
    Element subQuery;

    public Query(){
        this( ELEMENT_NAME );
    }

    public Query( String elementName ){
        super( elementName );
    }

    public void setNamespace( String s ){
        namespace = s;
    }

    public String getNamespace(){
        return namespace;
    }

    public void setSubQuery( Element e ){
        subQuery = e;
    }

    public Element getSubQuery(){
        return subQuery;
    }

    // processing
    public void startElement ( String namespaceURI, String elementName, String qName, Attributes
            attributes ) throws SAXException{
        if ( child != null ){
            child.startElement ( namespaceURI, elementName, qName, attributes );
            return;
        }

        if ( elementName.equals( ELEMENT_NAME )&& namespaceURI != null ){
            setNamespace( namespaceURI );
            // Auth
            if ( namespaceURI.equals( NAMESPACE_AUTH )){
                SQAuth sqAuth = new SQAuth();

                child = sqAuth;
                setupLogger( child ) ;
                sqAuth. startElement ( namespaceURI, elementName, qName, attributes ) ;
            } // Agents
            else if ( namespaceURI.equals( NAMESPACE_AGENTS )){
                SQAgents sqAgents = new SQAgents();

                child = sqAgents;
                setupLogger( child ) ;
                sqAgents. startElement ( namespaceURI, elementName, qName, attributes ) ;
            } // Roster
            else if ( namespaceURI.equals( NAMESPACE_ROSTER )){
                SQRoster sqRoster = new SQRoster();

                child = sqRoster;
                setupLogger( child ) ;
                sqRoster. startElement ( namespaceURI, elementName, qName, attributes ) ;
            } // OOB
            else if ( namespaceURI.equals( NAMESPACE_OOB )){
                SQOob sqOob = new SQOob();

                child = sqOob;
                setupLogger( child ) ;
                sqOob.startElement ( namespaceURI, elementName, qName, attributes ) ;
            }
            /* SIGN change */
            else if (namespaceURI.equals(ProcessIqExt.NAMESPACE_ROSTERCONFIG)){
                SQRosterConfig sqRosterConf = new SQRosterConfig();
                child = sqRosterConf;
                sqRosterConf. startElement (namespaceURI, elementName, qName, attributes);
            } else if (namespaceURI.equals(ProcessIqExt.NAMESPACE_PREFERENCES)){
                SQPreferences sqPreferences = new SQPreferences();
                child = sqPreferences ;
                sqPreferences . startElement (namespaceURI, elementName, qName, attributes);
            }
            /* End SIGN change */

        }
    }

    public void endElement( String namespaceURI, String elementName, String qName ) throws
        SAXException{
        if ( child != null ) {
            child .endElement( namespaceURI, elementName, qName );
        }
        if ( elementName.equals( ELEMENT_NAME )&& namespaceURI != null ){
            if ( namespaceURI.equals(NAMESPACE_AUTH)
                || namespaceURI.equals(NAMESPACE_ROSTER)
                || namespaceURI.equals(NAMESPACE_OOB)
                || namespaceURI.equals(ProcessIqExt.NAMESPACE_ROSTERCONFIG)
                || namespaceURI.equals(ProcessIqExt.NAMESPACE_PREFERENCES) ){
                subQuery = child ;
                child = null;
                return;
            }
        }
    }

    public String toString(){
        String s = new String();

        s += "<query";
        // Attributes
        if ( namespace != null )
            s += " xmlns=\"" + namespace + "\"";
        s += ">";
        if ( subQuery != null )
            s += subQuery.toString();
        s += "</query>";
        return s;
    }
}
```

References

[1] DJ Adams. *Programming Jabber*. O'Reilly, 2002.

[2] AOL Instant Messenger Web Page. http://www.aim.com/.

[3] AIM Wireless Web Page. http://www.aol.com/aim/wireless/index.htm.

[4] Donald Baker et al. Providing Customized Process and Situation Awareness in the Collaboration Management Infrastructure. CoopIS, 1999.

[5] Chediak, Mark. Instant messaging gets serious. *Red Herring*, 103, Septemper 2001.

[6] M. Day, S. Aggarwal, G. Mohr, and J. Vincent. Instant Messaging/Presence Protocol Requirements. RFC 2779, IETF, Feb 2000.

[7] M. Day, J. Rosenberg, and H. Sugano. A Model for Presence and Instant Messaging. RFC 2778, IETF, Feb 2000.

[8] P. Dourish and V. Bellotti. Awareness and Coordination in Shared Workspaces. *ACM CHI'92*, pages 107–114, 1992.

[9] Federal Communications Commision. Conditoned Approval of AOL - Time Warner Merger, January 2001.

[10] Fran Bushmann and others. *Pattern-oriented Software Architecture*, volume 1. November 1995.

[11] Tom Gross. Towards Ubiquitous Awareness: The PRAVATA Prototype. Parallel and Distributed Processing, 2001 Proceedings, Ninth Euro Workshop, Febuary 2001.

[12] HSQLDB. http://sourceforge.net/projects/hsqldb/.

[13] ICQ Web Page. http://www.icq.com.

[14] Jabber Central Web Page. http://www.jabbercentral.com/.

[15] Engin Kirda, Pascal Fenkamp, Gerald Reif, and Harald Gall. A service Architecture for Mobile Teamwork, 2002.

[16] Birgit Kreller et al. UMTS:A Middleware Architecture and Mobile API Approach, April 1998.

[17] Svein Løvland and Audun Mathisen. Java 2 Micro Edition - Technolgy driven architecture work. NTNU, Nov 2001.

[18] David Mandato et al. CAMP: A context-Aware Mobile Portal. IEEE Communications Magazine, January 2002.

[19] eMbedded Visual Tools Web Page. http://www.microsoft.com/mobile/developer/.

[20] J. Miller et al. Jabber RFC. http://www.ietf.org/internet-drafts/draft-miller-jabber-00.txt, Feb 2002.

[21] Madoka Mitsuoka et al. Instant Messaging with Mobil Phones to Support Awareness. IEEE, 2001.

[22] Morisio, Maurizi and Torchiano, Marco. Definition and classification of COTS: a proposal. In *1st International Conference on COTS Based Software Systems*, pages 165–175, February 2002.

[23] MSN Messenger Web Page. http://messenger.msn.no/.

[24] Microsoft Press Release, April 2001. http://www.microsoft.com/PressPass-/press/2001/Mar01/03-16MSNMessengerPR.asp.

[25] NanoXml and kNanoXml. http://nanoxml.sourceforge.net/index.html.

[26] Bonnie A. Nardi, Steve Whittaker, and Erin Bradner. Interaction and Outeraction: Instant Messaging in Action. CSCW, 2000.

[27] PersonalJava Runtime Environment. http://java.sun.com/products/personaljava/.

[28] B. Raman, R.H. Katz, and A.D. Joseph. Universal Inbox. IEEE, 2000.

[29] Recinto, Ronald. Shooting the messenger? *Red Herring*, 93, March 2001.

[30] J.C. Tang, N. Yankelovic, J. Begole, M.V. Kleek, F. Li, and J. Bhalodia. ConNexus to Awarenex. SIGCHI, April 2001.

[31] Yahoo! Messenger Web Page. http://messenger.yahoo.com/.