Jabber/J2ME based Instant Messaging Client for Presence Service

A Master Thesis in Media Computer Science

submitted to the University of Applied Science Stuttgart, Hochschule der Medien

UNIVERSITY OF APPLIED SCIENCES
HOCHSCHULE DER MEDIEN

Maiko Kozakai

June 4, 2004

First Supervisor: Prof. Dr. Martin Goik
Professor at the Faculty of Media Computer Science
University of Applied Science Stuttgart – Hochschule der Medien

Second Supervisor: MSc. Rodolfo López Aladros
Senior Consultant in Network Solutions
Alcatel SEL AG

Jabber/J2ME based Instant Messaging Client for Presence Service:
A Master Thesis in Media Computer Science

submitted to the University of Applied Science Stuttgart, Hochschule der Medien

UNIVERSITY OF APPLIED SCIENCES
HOCHSCHULE DER MEDIEN

by Maiko Kozakai

June 4, 2004

First Supervisor: Prof. Dr. Martin Goik
Professor at the Faculty of Media Computer Science
University of Applied Science Stuttgart – Hochschule der Medien

Second Supervisor: MSc. Rodolfo López Aladros
Senior Consultant in Network Solutions
Alcatel SEL AG

Table of Contents

List of Figures

List of Tables

Abstract

The Instant Messaging (IM) system together with its presence feature is a quite useful messaging platform providing various communication way such as text-based chat, conferencing and email-like message. The presence feature means the ability to see the online availability of the person whom we are willing to contact. This made the IM system to become so powerful and popular on the PC environment. Today, some companies offer mobile IM systems enhancing its form of communication to the voice-based nature. But only a few of those products are based on the open IM standard technology Jabber.

The basic idea of this thesis was given by Alcatel SEL AG in Stuttgart. Alcatel's main goal is to achieve a presence service system which allows user to be informed about online availability of his/her IM mates immediately after turning on his/her mobile phone. At the same time, depending on the user's preference setting, his/her presence would be forwarded to those who have subscription to it. Thus the participants of this service can get always accurate presence information about each other, as long as their mobile phones are turned on. This is an innovation, because so far there was no possibility for phone caller to know callee's presence.

But how the system can know that a user just turned on his/her mobile phone? The key lies in the Home Location Register (HLR), to which every mobile phone automatically sends a signal when it is on. Therefore such IM system requires a *presence server* which gets the mobile user's presence information from the HLR through a gateway and interacts with the client IM application on wireless devices. And the interaction should preferably use the today's IM standard Jabber protocol. This thesis deals with a Jabber based IM client application for mobile phone devices, which can be deployed with any Jabber server.

Chapter 1 is devoted to the general introduction of this thesis. The following two chapters introduce a minimum set of information about Jabber and J2ME technologies, making the following chapter easier to understand. Chapter 4 describes the selection procedure of the development environment. In Chapter 5, the software design and the requirement analysis are explained using unified modeling language (UML) and design patterns. Chapter 6 shows the process of actual implementation of the client application in detail and also the difficulties I confronted during the implementation. The software testing process is covered here, too. The last chapter gives the conclusion of my work and forecasts the future direction in the mobile IM market.

Declaration

I hereby declare to have written this master thesis entirely by myself and used no other sources than those listed in the resources part at the end of this thesis. Furthermore, I affirm that this thesis has never been published in Germany or in any other countries.

All the registered trademarks, names, logos and icons appearing in this thesis are the property of their respective owners.

_____

Date, Signature

Maiko Kozakai

Chapter 1. Introduction

1.1. Background

Instant Messaging (IM) is booming worldwide. Begun as a fun toy for consumers just to chat with friends, IM system has grown to be an important and attractive communication platform not only for the Internet community but also for the business world. The presence awareness offered by IM system can be coupled with any kinds of communication, in other words, possibilities with IM system are unlimited and still to be explored. The only obstacle is that there is no definitive standard in instant messaging yet. And this is why many developers are striving hard to create an open standard for today's and tomorrow's Instant Messaging. This effort embodies what is known as *Jabber*. Jabber is admittedly an ongoing work but its core protocol, Extensible Messaging and Presence Protocol (XMPP) is already approved by the Internet Engineering Task Force (IETF) as a Proposed Standard. Today, there are already a lot of IM applications based on Jabber. However, the number of those written for mobile phones are, as of this writing, still limited.

Realization of a Jabber-based IM client for mobile phone is a part of Alcatel's *Presence Service* project. The presence service should enable mobile phone users to enjoy instant messaging and to be presence-aware, anytime and anywhere. But here, to be present means that the user has turned on his/her mobile device, rather than to have logged onto an IM server as is often the case with the usual IM applications on PC. To understand how this can work, we need to look into the mechanism of cellular mobile network. The followings happen when you turn on your mobile phone:

1. The device sends a signal to the wireless network of a particular network carrier with which the user has a contract, telling that it turned to the active state. As long as it is on, the device sends this signal at regular intervals.

2. The nearest base transceiver station catches the signal and forwards it to the Mobile Switching Center (MSC), which is connected to the Home Location Register (HLR). The HLR contains the identities of mobile subscribers, their service parameters and their location information.

3. The HLR gets the information about the user's active state and location from the MSC and save this data in itself. Even if the user travels around here and there, this information always arrives to the HLR and is managed by it. In other words, the HLR "knows" your presence in the cellular mobile network automatically whenever you turned your device on or off.

The idea is to use this HLR's accurate knowledge about mobile user's presence for IM. Figure 1.1 roughly illustrates the presence service concept.

In order to realize this, we need several network instances such as the gateway to the HLR, the presence server that can communicate with the gateway, and naturally the client on mobile device that can get the other participants' presence information immediately after the device turns on. Creation of this mobile IM client using Jabber protocol is the subject of this thesis.

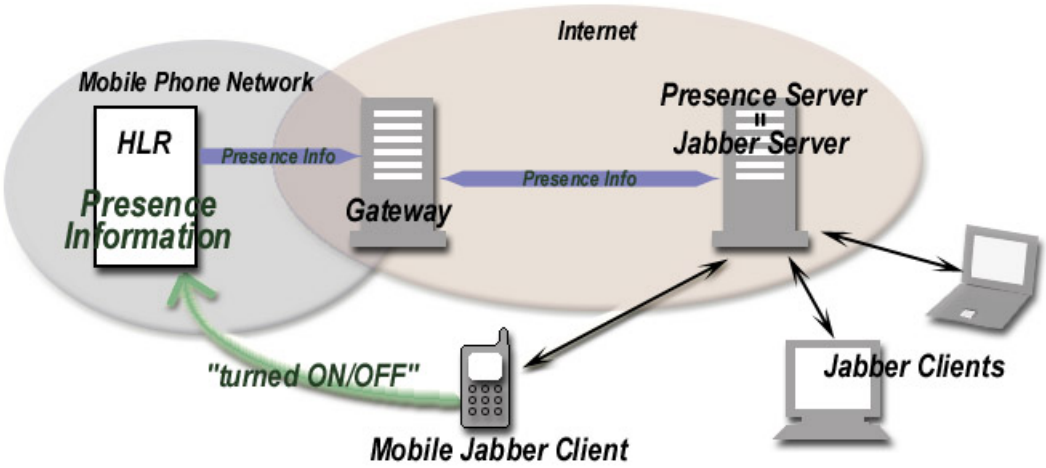

Figure 1.1. Presence service system to be constructed

1.2. Primary Objective

The existence of the presence server, which gets the presence information from the HLR and interacts with the client, is the prerequisite to this client application. Unlike normal client-server systems, the client should be in a position to get the sever push (that is, the server-initiated incoming communication request) as long as the device is turned on; the client should be invoked if it is not in active state. This can be done only by an arrangement between the two entities. However, the presence server was at the beginning of this work not existent. Hence I decided to create a generic Jabber IM client first so that this distinctiveness can be added later when the presence server is ready. In any case, both the client and the server must be able to "speak" in the Jabber protocol.

The requirements of a basic instant messaging and presence application are defined by RFC[1] 2779 ("Instant Messaging / Presence Protocol Requirements"), which at a high level stipulates that a user must be able to complete the following use cases:

- Exchange messages with other users
- Exchange presence information with other users
- Manage subscriptions to and from other users
- Manage items in a roster
- Block communications to or from specific other users

All these use cases are covered in Jabber protocol (or XMPP). This client application aims the complete implementation of XMPP to fulfill the requirements above.

[1]RFCs ("Request for Comments") are standard documents produced by the Internet Engineering Task Force (IETF).

Chapter 2. Instant Messaging with Jabber

The remarkable popularity of instant messaging deserves a closer examination. Beginning with the general overview of IM, this chapter goes further into IM using Jabber in particular.

2.1. What is Instant Messaging?

Thinking about messaging over the Internet, email might come up first in everyone's mind. Ever since the early days of the Internet, it has played an important role. Still today it enjoys an increasing percentage in the daily network traffic. On the other hand, instant messaging is catching up with email at a great speed in the last few years. First I would like to clarify the differences between email and instant messaging.

An instant message, as the name implies, is a message that can be instantaneously sent and received over the Internet, it means the communication session goes on in almost real time. First of all, the significant merit of IM is its versatility in the form of message. No matter short or long, email, one-to-one chat, group chat,[2] SMS or even telephony, all these can be accommodated by a single IM address that resembles an email address – while your email address is merely for email and nothing more than that. Secondly, IM combines this universality together with the concept of *Presence*. The combination results in the unique convenience what fascinates millions of users. Thus, Instant Messaging differs from email or simple chat systems by its ability

- to manage the various message types
- in a "near-real-time" fashion, offering the synchronous way of communication,
- and the ability for users to show and detect each other's presence.

2.2. Presence Awareness – the Key to Success

IM systems offers the "instant" delivery of messages and real time interaction between clients. These interactions can vary from simple text chats till complex collaborative applications such as groupware and online games. It means that both you and your IM partner(s) must be online at the same time. In most of IM systems, you can see whether your partner(s) are online and available in a so-called roster,[3] which makes the session initiation distinctively easier - one can be sure to catch the target person(s) for an online rendezvous, and there is no need to bother someone who is busy and unavailable (as is often the case with the phone). This is named presence feature. Presence information usually contains not only the online presence of the users but also the readiness for communication, for instance in the form of free configurable text describing one's status, such as "In a meeting", "Ready to chat :)", "Away to lunch" or "I'm at home" and so on. No matter how far the physical distance between the users might be, IM keeps track of each other's presence. This boosts up the efficiency in communication and the productivity of collaborative work. Many people can profit from this useful presence feature, in fact,

[2]The group chat facility is often called as "chat room" or "conference room" representing a place where participants can freely join and leave.
[3]A contact list showing presence state of your IM partner(s), widely known as buddy list as is named in the AIM.

numerous open source communities and enterprises utilize IM for their internal communication.

However, enabling users to see the presence of each other is not unproblematic. Imagine, what if everyone you know is constantly sending you an instant message? What about your privacy, if your presence information would fall into the wrong hands? We do not want the whole world to know when we go online or go to lunch. IM presence system must provide a way for us to determine who is given permission to see our presence, and a way to block unwanted incoming messages.

All the common IM systems on the market have dealt with these issues and established basic presence technology. And certainly presence technology will evolve further with the increasing demand for presence information in the online world. But now let us remind how this IM phenomenon has ever begun.

2.3. A Brief History of IM

The first IM application was created by three Israeli programmers at Mirabilis Ltd. in 1996. It was named ICQ ("I seek you")[4] and offered Internet users the ability to locate and communicate with other users in a near real time environment. Within a year, almost a million Internet users had registered – and currently ICQ has over 150 million registered users who exchange approximately 800 million messages a day.[5] After two years of amazing growth, Mirabilis and their ICQ technology were acquired by America Online (AOL) that has recognized the popularity and the importance of IM as a communication tool. Microsoft, Yahoo! and others also began the race to develop their own proprietary IM software utilities.

2.3.1. Typical Flow of Consumer IM Application

To briefly describe how IM works, we can take ICQ as a model. New users must first download and install the ICQ client software on their computers. The software is freely available and is so constructed that it connects to the ICQ server when it starts. On the first time the new user runs it, he/she is prompted for user name and password in order to set up an ICQ account. This account allows users to log on the ICQ server and exchange messages with other users. Each user's computer sends information (an IP address and the port number assigned to the ICQ client) to the ICQ server to establish a unique connection with the server. The client also allows users to create personal contact lists (called also "buddy lists" or "rosters") of other registered ICQ users with whom they would like to communicate. Once an initial roster is created, the server automatically checks if anyone on the list is currently online. Since the server identifies users by their IP addresses and port numbers, online users can exchange messages with other users on their rosters directly without server intervention. The server is responsible for maintaining a global user roster and updating each list whenever a user logs in or out.

[4] [ICQ]
[5] [Cox04]

2.3.2. Current Situation of the IM Market

Although the rapid development of competing IM utilities contributed to the prevalence of IM, it has complicated attempts to establish a ubiquitous IM standard. AIM (AOL Instant Messenger), ICQ, Yahoo! Instant Messenger and MSN Messenger and other IM applications each use proprietary communication protocol, utterly closed in their respective networks. For instance, although AIM and ICQ are now owned by the same company, their clients cannot communicate directly with each other. For their users, this means less members to contact and less flexibility. To surmount this shortcoming in interoperability and to make a truly open standard in the IM system, the Jabber project has started in 1998. Next section describes the Jabber IM technology in depth.

2.4. What is Jabber?

According to the *Jabber Software Foundation*,[6] Jabber is a freely available set of streaming XML (Extensible Markup Language) protocols and technologies that enable any two entities on the Internet to exchange messages, presence, and other structured information in close to real time. So Jabber is NOT a fixed piece of software, it is there for developers to create open, standard-conform IM applications. The first Jabber application is an IM network that offers functionality similar to legacy IM services such as AIM, ICQ, etc. However, the Jabber technologies offer several key advantages:

- Open – the Jabber protocols are free, open, public, and easily understandable; in addition, multiple implementations exist for clients, servers, components, and code libraries.

- Standard – the Internet Engineering Task Force (IETF) has formalized the core XML streaming protocols as an approved instant messaging and presence technology under the name of XMPP (Extensible Messaging and Presence Protocol), and the XMPP specifications are moving forward rapidly within the IETF's standards process.

- Proven – the first Jabber technologies were developed by Jeremie Miller in 1998 and are now quite stable; hundreds of developers are working on Jabber technologies, there are tens of thousands of Jabber servers running on the Internet today, and millions of people use Jabber for IM. In fact, it has been estimated that the number of Jabber IM users has recently surpassed the number of ICQ users.[7]

- Decentralized – the architecture of the Jabber network is similar to email; as a result, anyone can run their own Jabber server, enabling individuals and organizations to take control of their IM experience.

- Secure – any Jabber server may be isolated from the public Jabber network (e.g., on a company intranet), and robust security using SASL (Simple Authentication and Security Layer protocol) and TLS (Transport Layer Security protocol) has been built into the core XMPP specifications.

[6] [Jabber]
[7] [Cox04]

- Extensible – using the power of XML namespaces, anyone can build custom functionality on top of the core protocols; to maintain interoperability, common extensions are managed by the Jabber Software Foundation (JSF).

- Flexible – Jabber applications beyond IM include network management, content syndication, collaboration tools, file sharing, gaming, and remote systems monitoring.

- Diverse – a wide range of companies and open-source projects use the Jabber protocols to build and deploy real-time applications and services; you will never get "locked in" when you use Jabber technologies.

To use XML as the base means an immense technical advantage, for XML is a well-accepted open W3C (World Wide Web Consortium) standard. Besides, XML is fairly simple to read or parse, flexible for the customization, portable to almost any OS and can be handled in diverse programming languages. All these merits are naturally inherited to Jabber. In consideration of internationalization, Jabber's XML stream requires to be encoded in UTF-8.

The text-based nature of XML is indeed a benefit, but at the same time it is a drawback in terms of the bandwidth. Compared to binary data format, there certainly are inefficiencies caused by larger overhead. However, the XML-related benefits mentioned above outbalance this drawback. You will see in the later section how simple the Jabber protocol is structured. This open simplicity encourages more and more developers to join the Jabber community and to contribute, whereas renowned proprietary IM systems just remain as black boxes.

To solve the privacy problem mentioned earlier in Section 2.2, Jabber uses a *subscription* mechanism for the presence information. In addition, there is the *privacy list* concept in order to block communications from particular other (unwanted) users.

2.5. Jabber's Architecture

The Jabber messaging model follows the simple client/server architecture model similar to the email system. Jabber clients only communicate with Jabber servers in their Jabber domain. Jabber domains are the partial network units in the entire IM universe, each supported and managed by a separate, distributed Jabber server. And those different domains can interact through the server-to-server communication. This is in contrast with most of the commercial IM systems that use one centralized server for the whole IM system. In Jabber, messages pass from the sender's client to the sender's server, then to the recipient's server and from there finally to the recipient's client. Figure 2.1 shows this message flow over Jabber's client/server architecture.[8]

Advantages. Jabber's distributed client/server messaging model is simple to use and well-understood. Many messaging systems are built upon this type of architecture, such as email or enterprise message-oriented middleware (MOM), so it is proven to be stable. In case of company-wide (intranet) Jabber network, the distributed client/server messaging model reduces to an even simpler single set of client/server protocols. In comparison with peer-to-peer model, client privacy and security are much more improved because clients only talk to their server. Attackers on the network cannot know about the client's location.

[8] [Shigeoka02]

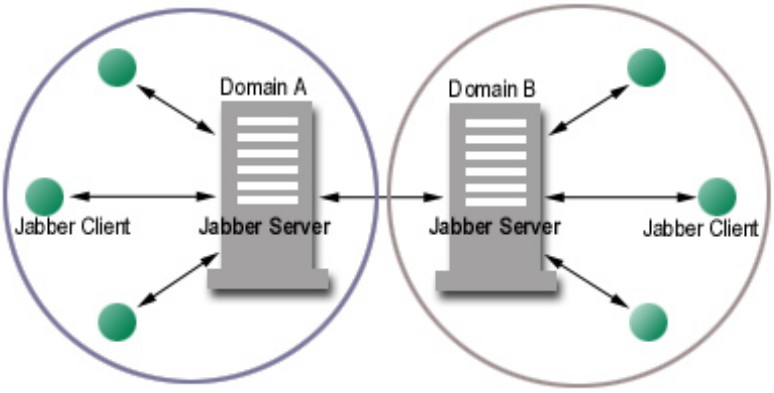

Figure 2.1. Jabber's client/server architecture with distributed servers

Eliminating the need for clients to act as a server enables clients to be lightweight. User specific data such as roster and presence subscription information are stored on servers, not on clients. Hence Jabber client applications can be developed quickly at a relatively low cost.

In addition, Jabber servers allow you centralized control over a Jabber domain. You can create policies and enforce them on the server without having to modify clients.

Disadvantages. In most cases, server-managed privacy and security are convenient and efficient. However, if you cannot trust your server, a centralized powerful server can be a major problem. A Jabber server is a typical single point of failure. Once the server is compromised, there is nothing preventing attackers from recording and reading your messages without your permission. But Jabber users have the possibility to send encrypted messages. Several commonly available encryption algorithms can protect data from unauthorized reading and/or editing of the message contents.

A server failure disables an entire server domain so it must be treated like a mission-critical server. Building reliable, highly scalable server software is a difficult but achievable goal.

2.6. Fundamental Concepts of Jabber

Before we can look at the Jabber protocols, we must understand some of the very basic concepts of Jabber. Fortunately, they are quite straightforward and intuitive.

2.6.1. Jabber Identifiers

An *entity* is anything that can be addressed in Jabber, for instance, a server, a component, a user connected with a client. Every entity is identifiable by a *Jabber ID* or for short, JID. These JIDs give the entities their *addressability*. Here is an example how JID addresses a user *juliet* connected to the domain *jabber.org* using her client on the machine (or resource) *Laptop*:

```
[Usage] username@hostname/resource
[Example] juliet@jabber.org/Laptop
```

A Jabber server is identified by a JID that doesn't contain a *username*, e.g. `jabber.org`. To address specific features of the server, a *resource* is often specified like `jabber.org/admin`. The JID *jabber.org/admin* is used by server administrator at *jabber.org* to obtain a list of online users. Administrators can send an announcement to all online users on the Jabber server *jabber.org* by sending a message to the JID: `jabber.org/announce/online`. In this case, the resource is *announce/online*, the second slash is just part of the *resource*.

2.6.1.1. A conference room

Jabber has a Conferencing component that provides group chat facilities akin to Internet Relay Chat (IRC). Whereas IRC has channels, the Jabber's conferencing component offers rooms. These rooms are addressed with JIDs in the form:

```
jab-dev@conference.jabber.org
```

The room name is specified in the *username* portion of the JID, and the *hostname* reflects the address of the conferencing component. Each room participant has an identity specific to that room, for example:

```
jab-dev@conference.jabber.org/bd05f766f98bd59d4c2d8a9d5a7bf5
```

The *resource* is a hexadecimal SHA-1 message digest of the participant's JID, designed to be unique and calculated and assigned by the conferencing component as a user enters the room. This is to shield the participant's real identity, which is the default setting for a conference room.

2.6.2. Resources and Priorities

We saw how differently the *resource* part of JID can be used. But the resource is mainly seen as a way of making a distinction between simultaneous connections by a user to the same Jabber server. For example, if you connect to a Jabber server using the same username and password on two different messaging endpoints, i.e. resources, the Jabber server will look at the resource part of the JID to determine which client to route messages to.

Now, what happens when someone sends you a message? To which client will the message be sent? This is where the concept of *priority* comes into play. Each Jabber client connection can be given a priority. XMPP defines the priority value to be any integer between -128 and +128, setting +128 as the highest possible priority: the higher the number, the higher the priority. If no priority is provided, a server should consider the priority to be zero. In practice though, often only positive integers are allowed.

Figure 2.2 shows priority in action. In this example, sender's message is forwarded to the recipient's Jabber client on the resource `Desktop`, as it has a higher priority than the other resource.

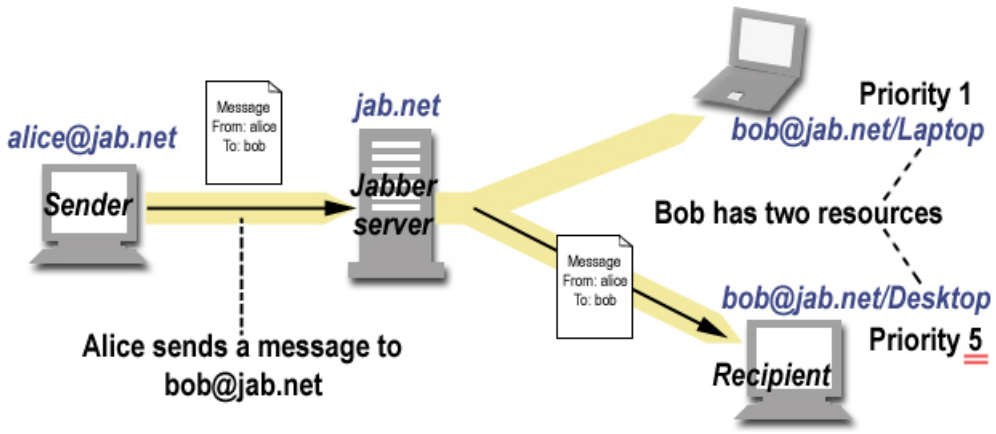

Figure 2.2. Resources, priority, and message delivery

2.6.3. Privacy

In order to protect the privacy of instant messaging users and any other entities, presence and availability information is disclosed only to other entities that the user has approved. When a user has agreed that another entity may view its presence, the entity is said to have a subscription to the user's presence information. A subscription lasts across sessions; indeed, it lasts until the subscriber unsubscribes or the subscribee cancels the previously-granted subscription.

Those entities, to whom a user has subscriptions, appear in the user's roster. A roster consists of any number of specific roster items, each of them being identified by a unique JID (usually of the form `contact@example.org`). A user's roster is stored by the user's server on the user's behalf so that the user can access his/her roster information from any resource.

2.6.4. XML Stream in Jabber Session

A connection between two Jabber endpoints, e.g. a client and a server, is made via a TCP socket, and XML is transferred between these endpoints. But it is not just random fragments of XML flowing back and forth, there is a clear structure imposed upon that flow. The entire conversation between two entities is embodied in a pair of XML documents in the following form:[9]

```
<?xml version="1.0"?>  --- xml declaration
<roottag>      --- opening root tag
  <fragment1/> ----
  <fragment2/>    |
  <fragment3/>    |--- each fragment is an XML stanza
  ...             |
  <fragmentN/> ----
</roottag>     --- closing root tag
```

The conversation is two-way, duplexed across a socket connection. The server is listening on port 5222, which is the recommended default Jabber port, for incoming client-initiated

[9][OReilly]

connections. Once a client has successfully connected to the Jabber server, it sends an XML declaration followed by the opening root tag to announce its intention to the server, which in turn responds by sending an XML declaration and opening root tag of its own. From then on, every subsequent piece of data that the client sends to the server is an XML stanza. In XMPP,[10] XML stanza is defined as a discrete semantic unit of structured information that is sent from one entity to another over an XML stream. An XML stanza exists at the direct child level of the root element. The connection can be closed by the client by sending the matching closing root tag. Of course, the connection can be also closed by the server by sending the closing root tag of *its* XML document.

Figure 2.3 shows the pair of XML documents being streamed across the TCP socket connection between client and server, over time.

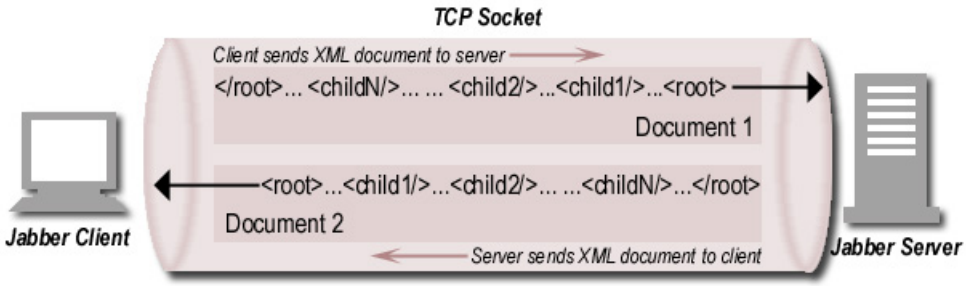

Figure 2.3. A conversation between client and server

2.7. XML Stanzas in Jabber Protocol

Now it is time to focus on *what* actually is exchanged in a Jabber session – the fragments that we saw in the previous section. Surprisingly, there are ONLY three XML stanzas defined in XMPP: `<message/>`, `<iq/>`, and `<presence/>`. Jabber manages to cover all the extensive needs in an IM system with these three basic elements, through clever use of XML namespace.

2.7.1. Message

Messaging is the very core of every IM system, and Jabber is no exception. The Jabber `<message/>` protocol provides a simple yet powerful framework for sending and receiving messages. The following is an example message. Please note that all the optional fields are set in order to make the example clearer:

```
<message
  to='romeo@example.net'
  from='juliet@example.com/balcony'
  type='normal'>
    <subject>Urgent!</subject>
    <body>Wherefore are thou, Romeo?</body>
    <thread>threadid_01</thread>
</message>
```

[10][XmppCore]

This example shows that a user *Juliet* sends a message, using her resource *balcony* to the user *Romeo*, who is on another Jabber domain. The `<message/>` element can be considered as an envelop, containing information in form of attributes and subelements. Addressing is done by the `to` and `from` attributes. There are five different message *types* defined, as the following Table 2.1 summarizes.

Message Style	Type	Model	Typical interface
Normal	normal	Email like message (default)	Message editor
Chat	chat	one-to-one online chat	Line by line chat
Group chat	groupchat	Online chatroom	Line by line chat
Headline	headline	Scrolling marquee message	"Stock Ticker"
Error	error	Message error occurred	Alert dialog box

Table 2.1. Jabber message types and messaging model

The `<body/>` element contains XML character data that specifies the textual contents of the message. Although this is optional, a `<message/>` stanza would normally contain a `<body/>` (which makes sense). The other two message-relevant elements, `<subject/>` and `<thread/>` are also optional and hence can be omitted.

2.7.2. Presence

The basic presence protocol is used in two primary contexts:

- *Presence update* – Informs people of your current presence state.
- *Presence subscription management* – Allows people to subscribe to another user's presence and control who has access to their own presence.

The presence update protocol uses a simple, one-way message. A client sends the presence update packet to the server. The server forwards copies of the packet to every interested party on the user's presence subscription list.[11] This process can be seen as a basic multicast with the "publish-subscribe" mechanism. Here is an example of presence update packet:

```
<presence from='juliet@example.com/balcony'>
    <show>away</show>
    <status>be right back</status>
    <priority>1</status>
</presence>
```

Once the server *example.com* gets this packet, it will look up its internal roster of the users that are subscribed to Juliet's presence and send the presence packet to them.

On the other hand, presence subscription management use a request-response protocol. Here is what Juliet's client would see when successfully subscribing to the presence of her friend Romeo:

[11]This applies to the default case, in which there is no 'to' attribute defined in the presence stanza. If a user specifies the 'to' attribute, the server would forward the presence information only to the intended recipient.

```
SEND: <presence from='juliet@example.com/balcony'
SEND:     to='romeo@example.net'
SEND:     type='subscribe'>

RECV: <presence from='romeo@example.net/orchard'
RECV:     to='juliet@example.com/balcony'
RECV:     type='subscribed'>
```

The presence 'type' attribute is required to determine the packet's type. The presence types are summarized in Table 2.2.

Presence type	**Protocol type**	**Meaning**
unavailable	Update	Signals that the user is no longer available for communication.
subscribe	Subscription management request	The sender wishes to subscribe to the recipient's presence.
subscribed	Subscription management response	Subscription to the sender's presence has been accepted.
unsubscribe	Subscription management request	A notification that an entity is unsubscribing from another entity's presence.
unsubscribed	Subscription management response	The subscription request has been denied or a previously granted subscription has been canceled.
error	Standard Jabber Error	An error occurred with regard to a previously sent presence stanza.
probe	Server request	A request for an entity's current presence; should be generated only by a server.

Table 2.2. Presence packet types

In addition to the attribute, a presence information can be qualified with more detail, using four child elements in the `<presence/>` element.

- `<status/>` – A natural-language description of the user's availability status (e.g. "On vacation", "be right back" etc.)
- `<priority/>` – The numerical delivery priority of this resource. Messages are routed to the resource that is available and has the highest priority as explained in Section 2.6.2.
- `<error/>` – The standard Jabber error packet.
- `<show/>` – This optional element has four pre-defined particular availability status of an entity or specific resource. Clients will typically use the show state to display standard presence icons, sound alerts, and so on. If no `<show/>` state is indicated, the user is assumed to be just in an `online` state and available. The pre-defined states for `<show/>` are:
  - *chat* – The user or resource is actively interested in chatting.

- *away* – The user or resource is temporarily away.
- *xa* – "eXtended Away". The user or resource is away for an extended period.
- *dnd* – "Do Not Disturb". The user or resource is busy and does not wish to receive any message.

From the technical point of view, the definition of `<show/>` element allows us to implement five different availability status: `online` (no `<show/>`), `chat`, `away`, `xa` and `dnd`.

2.7.3. Info/Query

The third and final basic element in Jabber is the `<iq/>` stanza ("iq" stands for "info/query"), which represents a generic framework for sending and receiving administrative or management information. It covers a number of complex use cases such as creating, updating and removing user accounts, user authentication and roster management.

The IQ protocol uses a request-response mechanism, similar to HTTP (Hyper Text Transfer Protocol). It has two primary participants; a requesting entity and a responding entity. In most cases, the requesting entity is a Jabber client. The data content of the `<iq/>` stanza is defined by the namespace declaration of a direct child element (whether `<query/>` or `<error/>`). The interaction is tracked by the requesting entity through use of the mandatory 'id' attribute. Thus IQ interactions follow a common pattern of structured data exchange such as get/result or set/result. Figure 2.4 visualizes this.

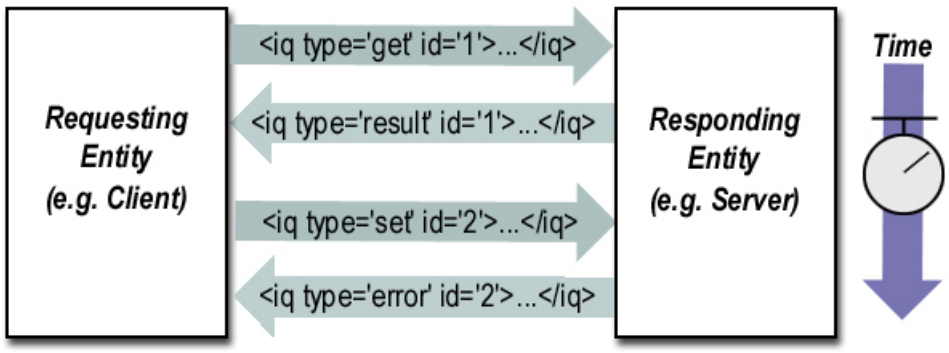

Figure 2.4. Entities in an <iq/> based conversation

The IQ packet must be addressed to a specific recipient, namely to a resource, not to a user. The only exception is IQ packets sent to the server where you can omit the recipient JID. A basic IQ syntax is as follows:

```
<iq  to='handler_jid'
   from='originator_jid'
   type='get | set | result | error'
   id='unique'>
  <query xmlns='iq-extension-namespace'>
    <query-field1/><query-field2/>
  </query>
 </iq>
```

The IQ 'type' attribute indicates the type of packet(s) being sent. These are described in Table 2.3.

IQ type	Direction	Description
get	requester-to-responder	A request for information or requirements.
set	requester-to-responder	The stanza provides required data, sets new values, or replaces existing values.
result	responder-to-requester	The stanza is a response to a successful get or set request.
error	responder-to-requester	An error occurred with regard of previously sent get or set.

Table 2.3. The IQ packet types and their usage

In the next example, a client requests current roster and the server responds to it.

```
SEND: <iq from='juliet@example.com/balcony'
SEND:    type='get'
SEND:    id='roster_1'>
SEND:   <query xmlns='jabber:iq:roster'/>
SEND: </iq>

RECV: <iq to='lala@example.com/work'
RECV:    type='result'
RECV:    id='roster_1'>
RECV:   <query xmlns='jabber:iq:roster'>
RECV:     <item jid='dinkywinky@example.net'
RECV:           name='Dink'
RECV:           subscription='both'>
RECV:         <group>Friends</group>
RECV:     </item>
RECV:     <item jid=dipsy@example.org'
RECV:           name='Dipsy'
RECV:           subscription='from'>
RECV:         <group>Work</group>
RECV:     </item>
RECV:     <item jid='po@example.net'
RECV:           name='Po'
RECV:           subscription='to'>
RECV:         <group>Friends</group>
RECV:     </item>
RECV:   </query>
RECV: </iq>
```

Roster request should be done upon connecting to the server. Should one of the roster item change its content, the server would automatically push the updated roster without being requested.

Chapter 3. J2ME Technology on Mobile Phone Device

This chapter is intended to explain in general terms what Java 2 Micro Edition Technology offers and how to develop an application with it.

3.1. Basic Information about J2ME

J2ME enables Java applications to run on small, resource-constrained devices such as PDA, mobile phone, set-top TV box, interactive pager etc. Being very much different from J2EE/J2SE, the J2ME architecture is designed to be flexible, modular and scalable in order to meet the market demand: It must cover up the large range of existing device types/hardware configurations/applications and features, and should stand the rapid improvement of the device technology. This modularity and scalability are defined by J2ME technology in a model with three layers[12] of software built upon the *Host Operating System* of the device. These layers are illustrated in Figure 3.1.

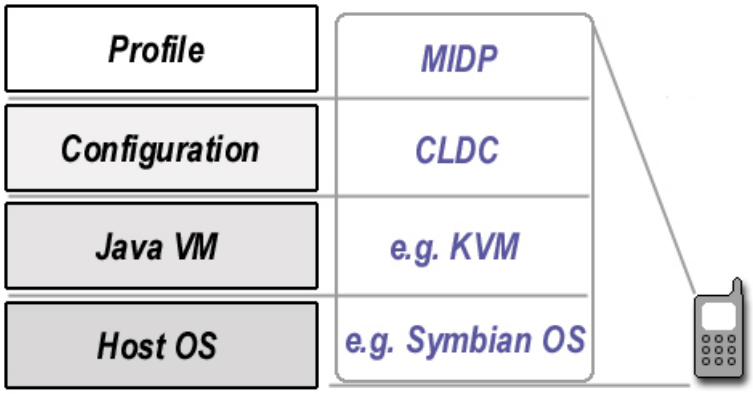

Figure 3.1. J2ME software layer stack and layers on a mobile phone

- *Java Virtual Machine Layer*. This layer is an implementation of a Java virtual machine (JVM) that is customized for a particular device's host operating system and supports a particular J2ME configuration.

- *Configuration Layer*. The configuration layer is less visible to users, but is very important for profile implementers. It defines the minimum platform for a particular "horizontal" category or grouping of devices, each with similar requirements on total memory budget and processing power. In a way, a configuration defines the "lowest common denominator" of the JVM features and libraries that the developers can assume to be available on all devices of the same category.

- *Profile Layer*. A profile is layered on top of (and thus extends) a configuration. This is the most visible layer to users and application provider. It defines the minimum set of Application Programming Interfaces (API) available on a particular "family" of devices representing a particular "vertical" market segment, for instance washing

[12] [J2ME]

machine or mobile phone. Applications are written "for" a particular profile and are thus portable to any device that "supports" that profile. A device can support multiple profiles. The mobile phone specific profile is defined under the name of Mobile Information Device Profile (MIDP).

3.1.1. Configuration

A Configuration defines the basic J2ME runtime environment. This environment includes the K Virtual Machine (KVM) or other conformable VM which is more limited than the VM used in J2SE, and a set of core classes derived primarily from J2SE. Currently, two configurations are defined in J2ME:

- **Connected, Limited Device Configuration (CLDC).** CLDC is aimed at the low end of the consumer electronics range. A typical CLDC platform is a mobile phone or PDA with around 512 KB of minimum available memory. This configuration includes some new classes (packaged in `javax.microedition.io.*`) designed specially to fit the needs of small-footprint devices.

- **Connected Device Configuration (CDC).** CDC addresses the needs of shared, fixed, connected information devices such as TV set-top boxes and web-screen phones, that lie between those addressed by the CLDC and the full desktop systems running J2SE. These devices have more memory (typically more than 2 MB) and more capable processors, hence they can support a much more complete Java software environment. To ensure upward compatibility between configurations, the CDC shall be a superset of the CLDC.

Since the target device of this thesis is mobile phone, I developed a client application on the top of the MIDP which extends the CLDC. Although the CLDC layer is not as visible as the MIDP layer to the application programmers, it is essential to understand the CLDC first because it defines the significant differences between J2ME and the well-accustomed J2SE environment.

3.1.2. CLDC Requirements and Specification

The entire CLDC implementation (static size of the virtual machine + libraries) should fit in 128 kilobytes and the CLDC specification assumes that applications can be run in as little as 32 kilobytes of Java heap space. In order to support a Java runtime environment with such limited resources, the CLDC defines reduced requirements for the virtual machine and the Java language specification. Compared to J2SE, the differences are as follows:

- **No floating point support.** No `float` and `double` exist, `Float` and `Double` either.

- **No object finalization and no weak references.** `Object.finalize()` does not exist.

- **Limitations on error handling.** Most subclasses of `java.lang.Error` are not supported. Runtime errors are handled in an implementation-dependent fashion.

- **No support for the Java Native Interface (JNI) or reflection features.** In particular, there is no support for object serialization.
- **No thread groups or daemon thread.** Threads are supported with reduced number of methods.
- **No user-defined class loaders.** An application cannot influence how classes are loaded. Only the runtime system can define and provide class loaders.
- **Class file verification is done differently.** The standard class verification process is too memory-consuming for small devices, so an alternate process was defined. The most of the verification work is separated into a *pre-verification* step that is typically performed on a server or desktop system before the class file is downloaded to the device. The preverified class files are then processed on the device using a much simpler kind of verification that merely validates the result of the preverification step.

The CLDC includes pretty much limited number of classes and interfaces from `java.lang`, `java.io` and `java.util`. Besides, the CLDC defines the classes that make up the Generic Connection Framework, or GCF for short. With the GCF, all communication is abstracted through a set of well-defined interfaces. For example, to open a Web connection, a developer might simply write:

```
Connector.open(http://www.example.org);
```

The GCF is leveraged and extended by the MIDP to allow the creation of network-aware applications. Now that a basic understanding of J2ME is attained, we step forward to the MIDP and MIDlets technology.

3.2. MIDP – Java on Mobile Phone

As mentioned above, a *profile* extends a configuration, adding domain-specific classes to the core set of classes. In other words, profiles provide classes that are geared toward specific uses of devices, such as user interface classes, persistence mechanism, and so on. While profiles provide important and necessary functionality, we must keep in mind that not every device will support every profile. In Japan, for example, NTT Docomo has released a number of Java-enabled cellular phones based on the CLDC but with their own proprietary profile.[13] Applications written for these devices will not work on cellular phones that support the MIDP.

The Mobile Information Device Profile (MIDP) specification was defined through the Java Community Process (JCP) by an expert group of more than 50 companies, including leading device manufacturers, wireless carriers, and vendors of mobile software. It defines a platform for dynamically and securely deploying optimized, graphical, networked applications.

Developers using the MIDP can write applications once, then deploy them quickly to a wide variety of mobile information devices. The MIDP has been widely adopted as the platform of choice for mobile applications. It is deployed globally on millions of phones and PDAs, and is supported by leading integrated development environments (IDE).

[13] [Ortiz01]

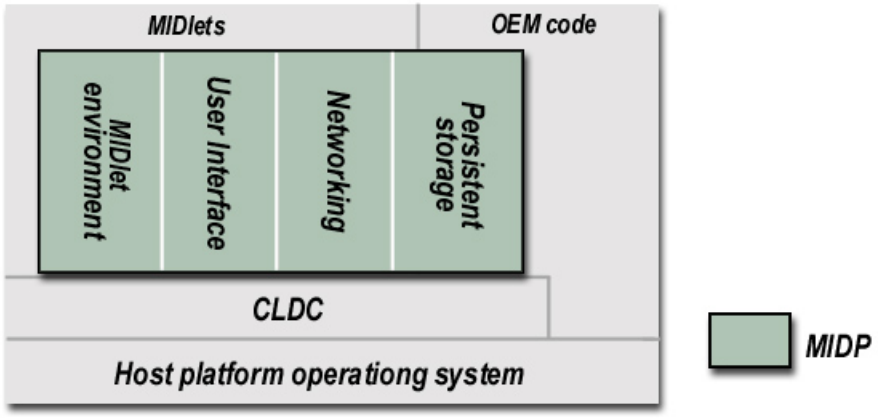

Figure 3.2. The Mobile Information Device Profile

Companies around the world have already taken advantage of the MIDP to write a broad range of consumer and enterprise mobile applications.

Today the MIDP 2.0 is implemented on many devices. It is a revised version of the older MIDP 1.0 specification, and includes new features such as an enhanced user interface, multimedia and game functionality, greater connectivity, over-the-air (OTA) provisioning, and end-to-end security. The MIDP 2.0 is backward compatible with the MIDP 1.0, and continues to target Mobile Information Devices (MIDs) such as mobile phones and PDAs.

3.2.1. MIDlets and MIDlets Suite

Java applications that run on the MIDP are known as *MIDlets*, and in some ways resemble the J2SE concept of Applets. Like Applets, MIDlets must extend the MIDP-defined abstract class `javax.microedition.MIDlet` and provide a public default constructor (i.e. a constructor that requires no arguments), which enables the system software to create an instance of MIDlet. MIDlets run in an execution environment within the Java VM that provides a well-defined lifecycle controlled via `MIDlet` class methods, which each MIDlet must implement. This MIDlet lifecycle is examined in the following section.

A collection of one or more MIDlets is packaged together into one JAR (Java archive) file to form a *MIDlet suite*. All the MIDlets in a suite can be installed, uninstalled and removed only as a group. The MIDlets in a suite share both static and runtime resources of their host environment, e.g. classes loaded into their mutual Java VM and persistent storage; in the MIDP 2.0 though, inter-suite access is allowed as for the persistent storage, if explicit permission is given.

3.2.1.1. MIDlet Lifecycle

MIDlet has one more analogy to Applet: like the Applet's `start`, `stop` and `destroy` methods, the MIDlet class defines three abstract methods that the system software calls to start, pause, and destroy an application – namely `startApp`, `pauseApp`, and `destroyApp`.

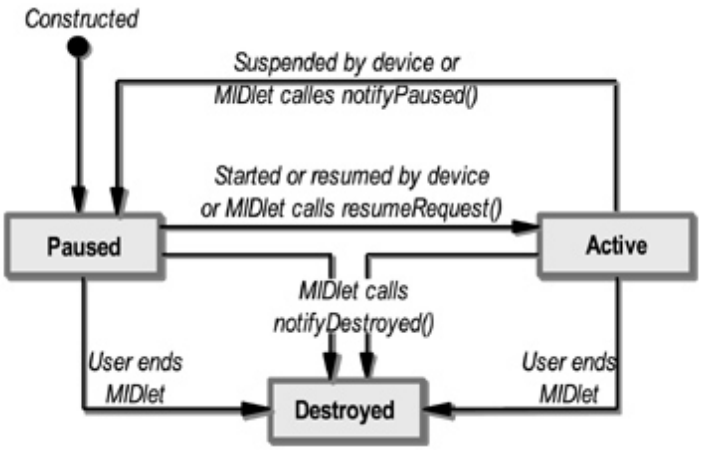

Figure 3.3. The lifecycle of a MIDlet

On a mobile information device (MID), an Application Management Software (AMS) controls the activation and deactivation of MIDlets. The AMS also maintains state information about each MIDlet. As Figure 3.3 shows, there are only three states possible:

- *active* – the application is running
- *paused* – the application has yet to run or is in an idle state
- *destroyed* – the application is terminated

Especially in the case of mobile phone, we must assume that the application would be often interrupted, among other things, by incoming or outgoing phone calls. Therefore it is essential to implement the MIDlet's `pauseApp()` method, to save important data and free up as much memory and other resources as possible before going into the paused state.

The destroyed state indicates that a MIDlet has terminated and resource associated with it can be freed by the system. Once in the destroyed state, the MIDlet cannot transition back to the other state. Again, developers are to implement the `destroyApp` method in a proper way so that the important data are saved, all the allocated resources are released, any background threads are terminated and any active timers are stopped as well as all the connections are closed. Remember, finalizers do not exist in the CLDC-based profiles, so the only way to free the resource used by an object is to do it explicitly.

3.2.1.2. MIDlet Packaging

MIDlets need to be properly packaged before they can be delivered to a device for installation. A single JAR file must contain *all* the required class files and any images (e.g. icons), or other files to which the MIDlet needs access at runtime (e.g. libraries). Apart from the CLDC and MIDP classes and any vendor-specific classes, the JAR file must be complete in itself. The reason for that is the prohibition imposed by the MIDP on the class file sharing between suites or dynamic downloading and installation of new classes as an application runs. Packaging information that tells the device what is in the JAR must be supplied in the JAR's manifest file. The manifest file defines important information about the MIDlet(s), such as the name, main class, icon(s) as well as information about vendor and required profile and configuration versions.

The JAR file is accompanied by an external file called the *Java Application Descriptor* (or JAD file). It is similar to the manifest, in fact, the two files share some data in common. But more importantly, the JAD file gives the information that helps the decision whether to download the complete JAR file and install or not. This information includes the size of the JAR file, the minimum amount of persistent memory required by the application, the version number, and so on. Placing this information in a separate file instead of in the JAR enables the device to quickly download the JAD file (after all, it's much smaller than the JAR) for analysis by the MIDP installation software.

3.2.1.3. MIDlet Security

Downloaded code can be considerably dangerous, especially when you don't know who wrote the code or where it came from. Although MIDlets running on a virtual machine are safer than binary code, there are still risks. In order to tackle this problem, the MIDP implementation utilizes the *sandbox* model: the sandbox metaphor is a way of describing the barriers that the Java runtime environment places in order to secure an application. In the MIDlet context, the sandbox ensures that a MIDlet can only load and use classes and resources from its MIDlet suite, allowing no inter-suite interaction. On the other hand though, it means a MIDlet cannot access address book or calendar functionality on the mobile phone, which might be quite convenient.[14]

In addition to the sandbox model, the MIDP 2.0 defines two categories of MIDlet suites: *trusted* and *untrusted*. Based on Internet X.509 Public Key Infrastructure (PKI), a trusted MIDlet must have a signed certificate. A certificate ensures that a MIDlet is from a trusted source and its contents have not been altered (i.e. data integrity). The `javax.microedition.pki` package provides a `Certificate` interface to read information written in certificates. Untrusted MIDlet suites do not have access to protected APIs like HTTP connection or SMS sending API without explicit confirmation by the user. This mechanism prevents MIDlet to initiate unauthorized outbound connection and/or to send private data to server.

The last but not least: while the MIDP 1.0 required support only for HTTP, the MIDP 2.0 adds support for secure HTTP (HTTPS), realizing end-to-end connection security. HTTPS is used, for example, in Internet banking service.

3.2.2. Over-The-Air Provisioning

There are several ways to install MIDlet suite on a device, such as from desktop or laptop via Bluetooth, or over the Web. The latter is defined in a specification referred to as *over-the-air provisioning* or *OTA provisioning*. The goal of the OTA provisioning is to make it possible and easy for the user to interact directly with web sites on the Internet in order to find and download MIDlet suites. A typical procedure can be described as follows:

[14]For PDAs, there is an API named `javax.microedition.pim.*` to access those utilities on the device.

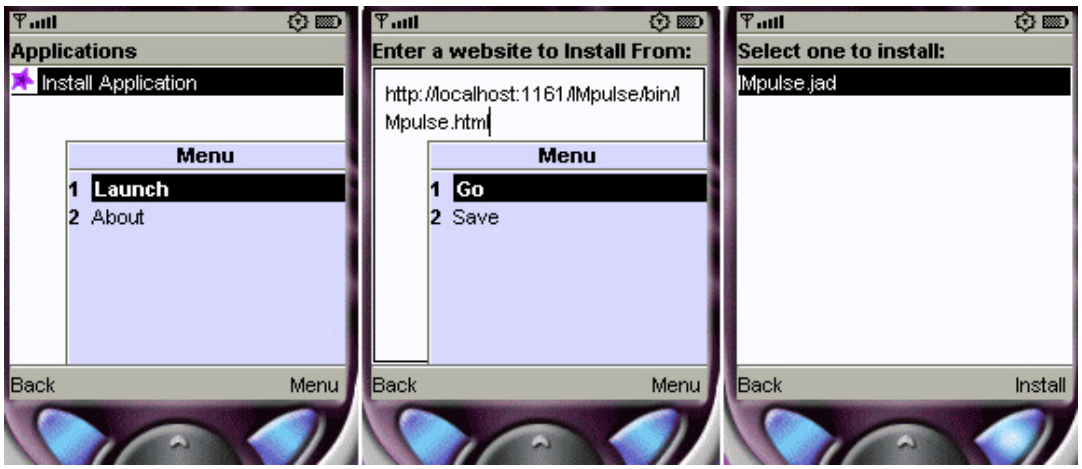

Figure 3.4. OTA provisioning steps to get JAD file

1. User browses the web page with a link to the JAD file via WAP or microbrowser on the wireless device.

2. User clicks the link to fetch the JAD file.

3. Web server (the provisioning server) sends the JAD file. This contains the URL for the JAR, that is, the actual MIDlet suite.

4. All the requirements in terms of memory and version are examined by AMS. If the user confirms the installation, a request to that URL for the JAR is sent.

5. Provisioning server (or proxy, if any) delivers the JAR and the MIDlet suite is installed (Figure 3.5).

After installation is complete, the AMS optionally sends a status report back to the server in form of an HTTP POST request. If any cookie is used, it will be discarded after installation.

Figure 3.5. After downloading of JAR file, MIDlet is ready to be launched

3.2.3. User Interfaces

MIDlets are interactive applications, so even the most minimal MIDlet requires some kind of user interface. Although the latest development allowed mobile phone to gain multicolored, larger LCD screen and relatively sophisticated input keys, you cannot compare these input and display capabilities with those of the desktop system. Besides, space limitations constrains user interface (UI) classes to be minimal and light. Consequently, J2ME needs a completely different approach than J2SE with its Abstract Windows Toolkit (AWT) and Swing. None of those heavy UI classes are used in the MIDP, instead, the `javax.microedition.lcdui` package is defined.

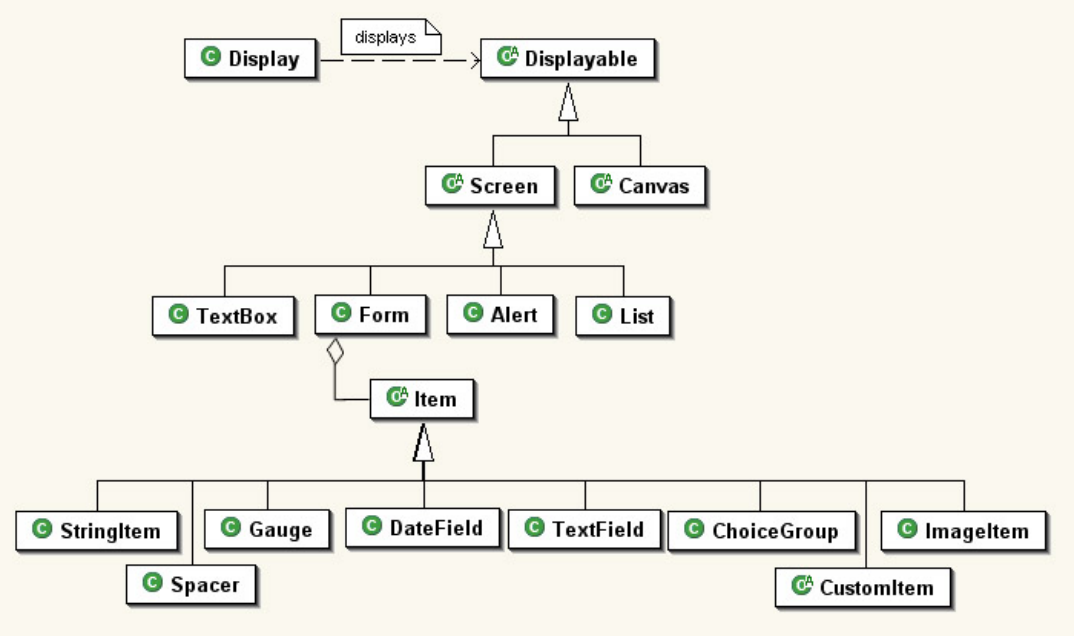

Figure 3.6. User Interface classes in the MIDP

The MIDP UI model considers the display to consist of a single, fixed-size window whose contents are controlled by a MIDlet or the system at any time. The window displays one of a set of virtual *displayables*, which are UI components that have the same height and width as the window. As the application runs, it changes the displayable that the window shows. The application is like a deck of cards – only the topmost card in the deck is visible at any given time. Displaying a different card means shuffling the deck to make another card the topmost card. This method is exactly how the MIDP user interface classes work, but with displayables instead of cards.

Thus, the class `Display` can be seen as the manager of the displayable screens and input devices of the system. There is exactly one instance of `Display` per MIDlet and the application can get a reference to that instance by calling the following static method (n.b. no constructor available in the class `Display`):

```
public static Display  Display.getDisplay(MIDlet myMidlet);
```

The MIDP UI has a high-level and a low-level API. The high-level UI API classes `Alert`, `Form`, `List` and `TextBox` are subclasses of the abstract class `Screen`. These classes are designed mainly for business applications and are highly portable across devices. The particular device's implementation of these classes performs the adaptation to its hardware

and native UI look and feel. The low-level API is based on the use of the abstract class `Canvas` and used mainly for game applications.

Figure 3.7. Same program, different look and feel

3.2.3.1. Navigation through Screens

We know now that a MIDlet can be seen as a set of screens through which the user navigates. The application is responsible for switching between these screens, which is achieved using the statement:

```
Display.getDisplay(this).setCurrent(screen_to_show);
```

The navigational aspects of a MIDP application should be designed with care due to the limitation in display size and input keys. Users need easy access to the key features of the application, so navigation should not be too complicated and entering data should be as simple as possible. So developers should not hesitate to divide long context into multiple screens and let the user switch them frequently.

3.2.3.2. Command Handling

The `Command` object is used for controlling user interaction. Every `Screen` and `Canvas` can have an arbitrary number of commands. The Command constructor has three parameters:

```
public Command (String label, int commandType, int priority);
```

whereas the arguments have the following meanings:

- The `label` argument supplies the text that will be used to represent the `Command` in the user interface. This label is normally a short but descriptive verb, the shorter the better.
- The `commandType` argument describes the action type. There are eight possible action types defined in the `Command` class. They are listed and explained in Table 3.1.
- The third argument, `priority` signals the importance of the action relative to other actions on the same screen. Priority values are integers, where a lower number indicates greater importance. More than one action can use the same priority value.

For instance, an "Exit" command and a "Help" command on a Form might be defined as follows:

```
Command exitCommand = new Command( "Exit", Command.EXIT, 1 );
Command helpCommand = new Command( "Help", Command.HELP, 3 );

Form myForm = new Form( "Demo" );
myForm.addCommand( exitCommand );
myForm.addCommand( helpCommand );
```

Type	Meaning
BACK	Navigate back to the previous screen.
CANCEL	Cancel the current screen and move to the previous screen.
EXIT	Exit the application.
HELP	Display general or context-sensitive help.
ITEM	Apply the action to the currently selected item on the screen.
OK	Confirm the current screen and move to the next screen.
SCREEN	Apply the action to the entire screen. Used for e.g. "Load" or "Save".
STOP	Stop the operation in progress.

Table 3.1. Command types

Command's application-level handling is based on a listener function. When the user triggers a command, nothing happens automatically in response to the command – the triggering merely generates an event for the application to trap. To receive command events, each `Displayable` object must have a single `CommandListener` with its `commandAction()` method properly implemented. Whenever a command is triggered, the listener's commandAction method is invoked and the appropriate action can be performed.

3.2.4. Network Communication

In the MIDP 2.0 specification, implementation of HTTP 1.1 and HTTPS is mandatory for the devices to be compliant. Although the MIDP 2.0 provides the interfaces for UDP-, TCP-, and TLS-based communication, support for those protocols is entirely optional. Figure 3.8 shows the available connection classes.

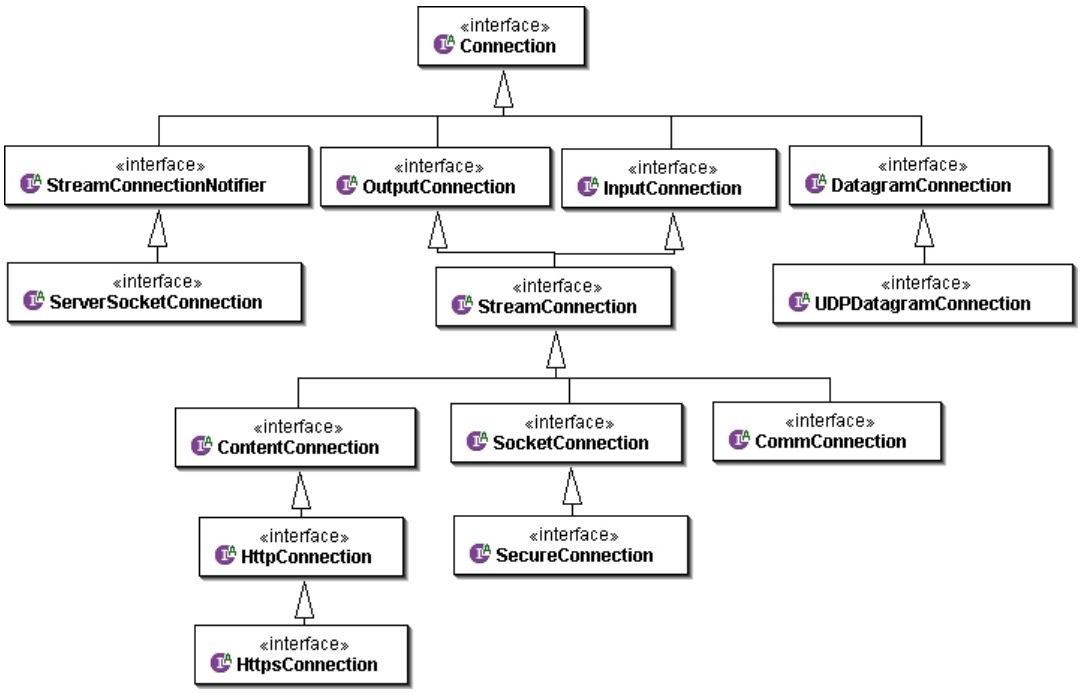

Figure 3.8. MIDP networking classes based on CLDC GCF

An example code to open up an outbound TCP connection might look like:

```
SocketConnection sc = (SocketConnection)
                        Connector.open("socket://myhost:5222");
```

Networking operations are often slow and are therefore typically initiated in a separate thread. Threading is one of the major subjects in MIDlet programming and therefore discussed later in Chapter 6.

3.2.5. Persistent Storage

The MIDP does not allow applications to read and write to a filesystem. However, MIDlets often need to store and retrieve data, such as:

- User configuration or preference information
- A history of the user's past inputs or game score
- Data that the user recently entered or uses frequently

For this reason, a data persistence mechanism named the *Record Management System* (RMS) is defined in the MIDP. In RMS, a persistent datastore is called a *record store* being represented by the `RecordStore` class. It is a collection of records that persists across multiple invocations of a MIDlet and is shared with other MIDlets within the same suite. Also, if explicit permission is given, MIDlets within other suites can access the record store. A MIDlet suite can have multiple record stores, each identified by a string. The `RecordStore` class provides a series of methods for opening, closing and deleting a record store, and modifying the contents of it.

The data in a record store is stored in chunks called records. A record is written as byte arrays of arbitrary size. Attention: Due to the lack of support for object serialization, the MIDlet must serialize objects itself. To read a record, the byte arrays must be converted back to the original "readable" form. Records are accessed by index, starting at 1, and each new record gets a unique index. They can be enumerated, stored and filtered in application-defined way.

Record stores are thread-safe, allowing different threads to access a record store simultaneously without data corruption. The RMS synchronizes all read and write operations and so guarantees that these operations are atomic.

3.2.6. Push Registry

A new *Push Registry* feature came together with the MIDP 2.0. It allows MIDlets to be started based on a request from an inbound connection or scheduled timer. This new feature brings with it a wealth of potential applications. For example, a remote application can push stock quote information on an hourly basis to a mobile device. Using the alarm feature, a MIDlet can now offer up reminders or scheduled events, such as a reminder of a meeting an hour before it begins. The push registry maintains a list of acceptable incoming network requests, which can be any of the following types:

- Message-based, such as SMS
- Packet-based, such as datagram
- Stream-based, such as socket

When the application management system receives an incoming request, it performs a lookup of the port and MIDlet name in the list. If the MIDlet is not currently active, it will be started. Registrations are done in one of the two ways:

- Dynamically at runtime, using `registerConnection` or `registerAlarm` method of `PushRegistry` class
- Statically through entries in the JAD file[15]

A static registration is defined using the `MIDlet-Push-<n>` attribute. The format is:

```
MIDlet-Push-<n>: <ConnectionURL>, <MIDletName>, <AllowedSender>
```

where `MIDletName` is the MIDlet to invoke and `AllowedSender` is the filter to specify which senders are valid for launching the requested MIDlet. When setting the `AllowedSender` parameter as an asterisk (*), the AMS accepts requests from any sender over the specified port.

```
MIDlet-Push-1: sms://:1000, MyMIDlet, *
MIDlet-Push-2: socket://:100, corej2me.NewsLink, *
```

Static registration entries can only be changed by uninstalling a MIDlet, making the changes to the JAD file, and re-installing the MIDlet. Should you need to remove a dy-

[15] Alarm-based activation is not supported through static registration.

namic registration, you can call `PushRegistry.unregisterConnection` method. Dynamic registration can be changed at any time during the MIDlet runs.

Chapter 4. The Environment

4.1. Target Device: Nokia 6600

To accomplish this thesis, Alcatel put the Nokia 6600 to my disposal. The main technical specifications of this device is summarized in Table 4.1. With its camera, video recorder and email feature, the device falls under the category of the so-called smart phones.

Developer Platform	Series 60 Developer Platform 2.0
Operating System	Symbian OS 7.0s
Java Technology	CLDC 1.0 and **MIDP 2.0**
Screen Display	• Color Depth: 65536 Colors (16 bit) • Resolution: 176 x 208
Physical Measurements	• Dimensions: 109 x 58 x 24 mm • Weight: 125 g
Memory	• Heap size: **3 MB** • Shared Memory for Storage: 6 MB + MMC
Max JAR Size	**Memory allocated dynamically**
Network Data Support	CSD, HSCSD, **GPRS**
Band Functionality	GSM
Keypad Descriptions	Grid key mat, 2 labeled soft keys, 5 way scrolling
Browser	WAP 2.0, XHTML over TCP/IP
Messaging	SMS, MMS+SMIL

Table 4.1. Nokia 6600 technical specs

Worth to mention is that there is no limit set to the maximum JAR size. Moreover, General Packet Radio Service (GPRS) support is essential in order to make the transmission between client and server charged per packet volume, not per connection time.

Figure 4.1. Nokia 6600 device © Nokia Corporation.

4.1.1. Developer Platform 2.0 for Series 60

Nokia's developer platforms provide a common set of technologies and APIs that make it easier to adapt applications across a wide range of devices within the same device series. In short, the platforms are the base-software implementations for entire families of devices. Different devices based on the same platform are targeted to different consumer segments, allowing the applications to reach several consumer segments simultaneously.

Nokia 6600 belongs to the series 60, and is the first device compliant with Series 60 Developer Platform 2.0 which supports the installing and running of Java applications on the device series 60, based on the MIDP 2.0 and CLDC 1.0 specifications. All developers intending to code applications on the series 60 devices are well advised to read the documentation[16] about this platform. Although it implements all of the mandatory features of the MIDP 2.0, this developer platform clearly differs from the Sun's J2ME reference implementation in several points:

- `MIDlet.pauseApp()` is never called in Nokia implementations. Appropriate methods can be called, but the implementation does nothing.
- The Series 60 MIDP implementation automatically adds an Exit option to the command menu. A developer-specified Exit command would cause two Exit options to be added to the menu and potentially confuse the user. However, if the MIDlet needs to be portable, an Exit command must be specified.
- In the Series 60 devices there exist certain behavioral rules that will already be familiar to users. For instance, if there are two soft keys in the device (left and right) the right soft key provides Back or Exit functionality. To design easy-to-use applications, developers should respect these usability issues. Such negative, backward actions are always mapped to the right soft key and positive actions to the left soft key.
- Inter-suite sharing of persistent data is not allowed, MIDlets cannot access the persistent data of MIDlets within other suites.

In addition, the following Java APIs are supported:

- Nokia UI API – Nokia proprietary API enables Java applications to use sound, full screen graphics (`FullCanvas` class), etc.
- Wireless Messaging API (JSR-120) – Allows sending and receiving of SMS messages in text and binary format.
- Mobile Media API (JSR-135) – Provides access and control of basic audio and multimedia features.
- Bluetooth API (JSR-82) – Enables Java applications to use Bluetooth connectivity.

[16] [Nokia]

4.2. Utility Tools

4.2.1. Wireless Toolkit 2.1

The J2ME Wireless Toolkit, or WTK[17] for short, consists of sets of tools that provide application developers with the emulation environments, documentation and examples needed to develop applications targeted at MIDP compliant mobile phones. The lack of source editor prevents the WTK from being a fully qualified Integrated Development Environment (IDE), and indeed the WTK does not aim to be such. Instead, several IDEs have the option to integrate the WTK to utilize its emulation environment. Once you have written your MIDlet code on a text editor or on an IDE, the WTK would be the first choice of tool to test the code on its emulator. Besides emulating, you can also build, package and sign the MIDlet suites. WTK can also simulate OTA provisioning and create obfuscated JAR file.

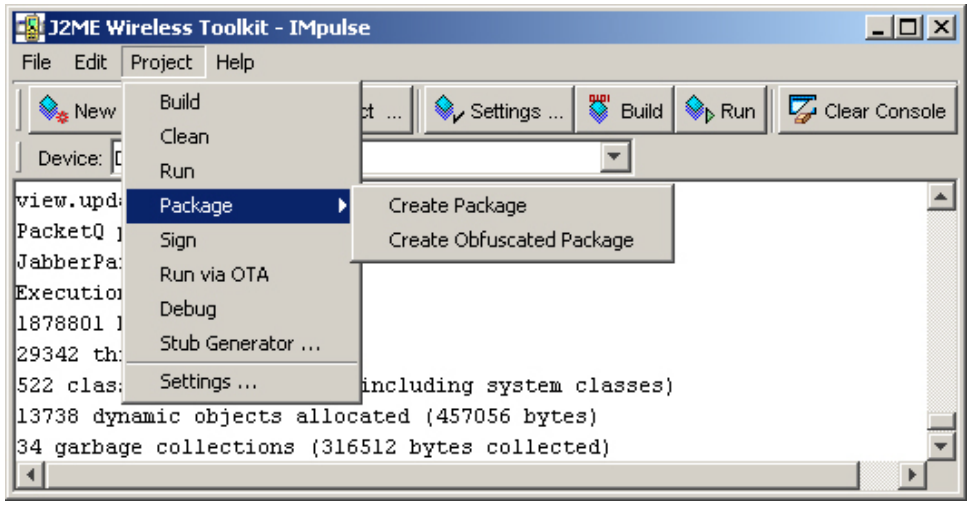

Figure 4.2. Menu and console of Wireless Toolkit

Generally speaking, to obfuscate something means to deliberately make it seem so confusing or opaque as to be difficult to perceive or understand.[18] In the application development context, an obfuscator is a program that garbles a source either by encryption or by substitution of codes. This means that the source to be compiled is hard to understand and to debug. But during runtime it is decrypted as pure source. Building an obfuscated JAR file therefore prevents decompilation and reverse engineering of applications, improves performance and reduces size of applications. In order to create an obfuscated JAR with WTK, you need the ProGuard[19] plug-in, a freely available Java class file shrinker and obfuscator. ProGuard can detect and remove unused classes, fields, methods, and attributes. It can then rename the remaining classes, fields, and methods using short meaningless names like a, b, c. The resulting JARs are smaller and harder to reverse-engineer.

On emulation, one can monitor networking, class loading, garbage collection, exceptions thrown and method calls (the latter results in getting lots of output on the console). The

[17] [WTK21]
[18] *Collins COBUILD English Dictionary for Advanced Learners* (HarperCollins Publishers, © 2001), p.1057.
[19] [ProGuard]

WTK's network monitoring (see Figure 4.3) is very useful to track incoming and outgoing XML packets and therefore helped my project to a large extent.

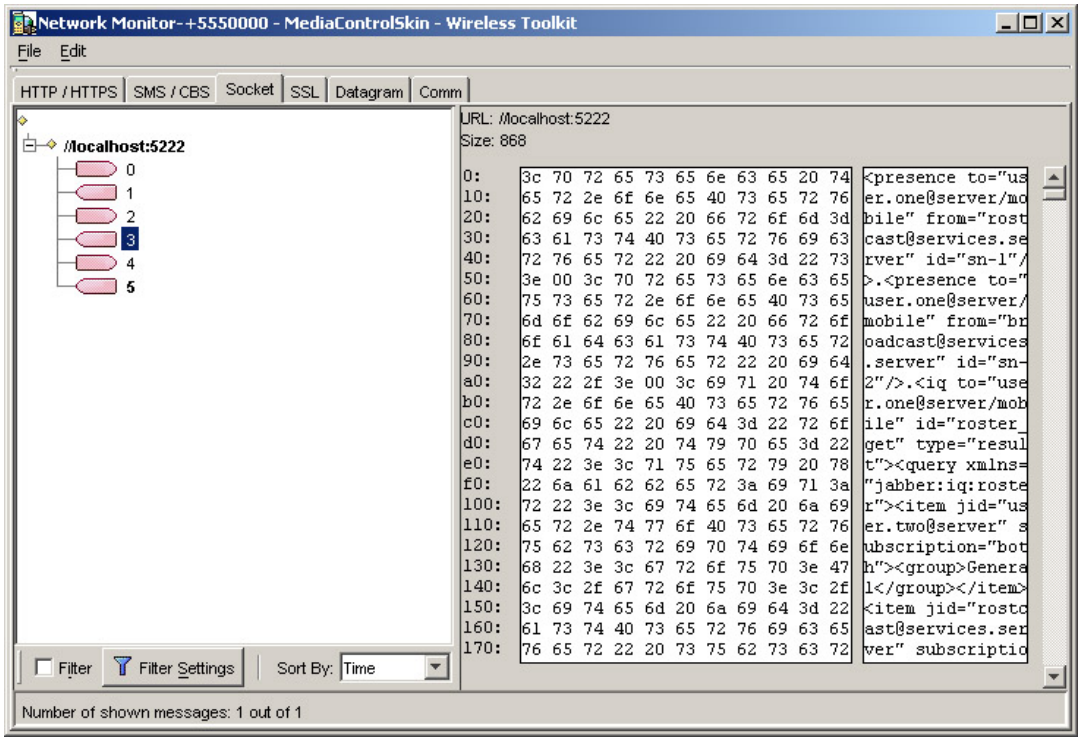

Figure 4.3. Network monitoring tool of Wireless Toolkit

4.2.2. Nokia Developer's Suite for J2ME

The Developer's Suite for J2ME[20] is a utility tool similar to the WTK with Nokia-specific device emulator. Apart from WTK-like utility tools as listed below, this assists developers working on MIDlet suites for a certain device series of Nokia with its example codes and documentation.

- create class and application package (JAR)
- sign application package
- emulation on series 60 MIDP concept SDK beta 0.3.1 Nokia edition
- trace on networking, threading, class loading, exceptions, allocation, gc etc.

4.2.2.1. Series 60 MIDP Concept SDK Beta 0.3.1, Nokia Edition

The Series 60 MIDP Concept Software Development Kit Beta 0.3.1, Nokia Edition contains a Nokia concept emulator for the Series 60 Platform, Java class libraries, APIs (including Nokia UI API), and documentation. The SDK works as a plug-in with the Nokia Developer's Suite for J2ME and can also be used with various Java Integrated Development Environments (IDEs).

[20][Nokia]

The Series 60 MIDP Concept SDK facilitates application development for the user interface used in the Nokia 6600 and other Series 60 phones. The phone emulator in this SDK is still in beta phase and is designed to show how applications may appear on the Nokia Series 60 phones. It emulates the application look and feel as well as the behavior of the Nokia Series 60 phone, but there are some differences between this emulator and the actual phone. Consequently, it is strongly recommended to test the application on an actual device.

4.3. Integrated Development Environment

Java Integrated Development Environments (IDE) specially tailored for MIDlet development is indispensable due to the distinctive quality of J2ME as explained in Chapter 3. I took a close look at the free IDEs simply because they are handy to begin with. They offer J2ME specific features such as:

- Integrated source-level debugging of MIDlets
- Integrated compilation, pre-verification, and execution of MIDlets suites
- Automatic generation of J2ME JAD and JAR files
- Emulator SDK Registry
- Integrated obfuscator and optimizer support

4.3.1. Sun One Studio 4 Mobile Edition

Sun ONE Studio 4 update 1, Mobile Edition pre-integrates Sun's J2ME Wireless Toolkit as a default emulation environment. It allows you to integrate the third-party device emulator of your choice, such as SDKs from Nokia, Sony Ericsson, Motorola, Siemens and JPhone. It provides indeed comprehensive tools. However, after several weeks of employment, this IDE had to be omitted from the choice on account of too slow reaction of its user interface.

4.3.2. Eclipse IDE with J2ME plug-in

Over the last semesters, I have tried different Java IDEs and the Eclipse IDE has convinced me of its high usability. Eclipse supplies an excellent source editor and project management, on top of that, great extensibility through numerous plug-ins. Due to its open source nature Eclipse finds increasing number of plug-ins for almost every purpose, including the J2ME plug-in. To develop MIDlet suites with Eclipse, you need such a plug-in as the SIPtech's J2ME plug-in[21] or Eclipse ME 0.3.0. I used the former one and could develop the J2ME application comfortably without any problem.

4.4. Testing Environment

In order to test my application, I needed a XMPP server and a Jabber client on desktop that simulate a standard Jabber network.

[21] [Plugin]

4.4.1. Jabber/XMPP Server

At least one XMPP server is needed in order to test the client. There is a number of public Jabber servers available on the Internet, on which everyone can create account and enjoy instant messaging. One of the most standard public Jabber servers, jabber.org, is used while a connection to the Internet exists. In case there is no Internet connection, I decided to install a server on the local machine. Table 4.2 shows the Jabber servers which were taken into consideration.

Server	License	Platform	Grade of Feature Implementation
Antepo OPN	Commercial	AIX, HP-UX, Linux, Solaris, Windows	86%
Merak	Commercial	Windows	90%
jabberd 2.x	GPL	AIX, BSD, HP-UX, Linux, MacOS X, Solaris, Windows	78%
Jabber XCP	Commercial	HP-UX, Linux, Solaris, Windows	78%
TIMP	Commercial	Windows	79%
SoapBox Server	Commercial	Windows	67%
ejabberd	GPL	AIX, BSD, HP-UX, Linux, Solaris, Windows	66%

Table 4.2. Jabber servers

Owing to its high score of the feature implementation, Antepo's Open Presence Network (OPN) server[22] was chosen for the purpose of testing. Though OPN server is a commercial product, it provides a free evaluation version which I made use of.

Antepo OPN server implements J2SE 1.4 technologies and XMPP standards. On Windows, this server must be installed as a Windows service which you can start and stop at any time. Its user interface can be reached using Internet Explorer. The installation is uncomplicated and as the product claims itself to be an "out-of the-box" solution, I could start client-server session without any changes to its initial configuration. The evaluation version has some pre-defined users to test server functionality.

4.4.2. Jabber Client for Desktop

Exodus is one of the most feature-rich open source Jabber clients for Windows platform. It is capable to debug XML packet exchange as shown in Figure 4.5. This client is mainly used to test the presence subscription procedure.

[22][Antepo]

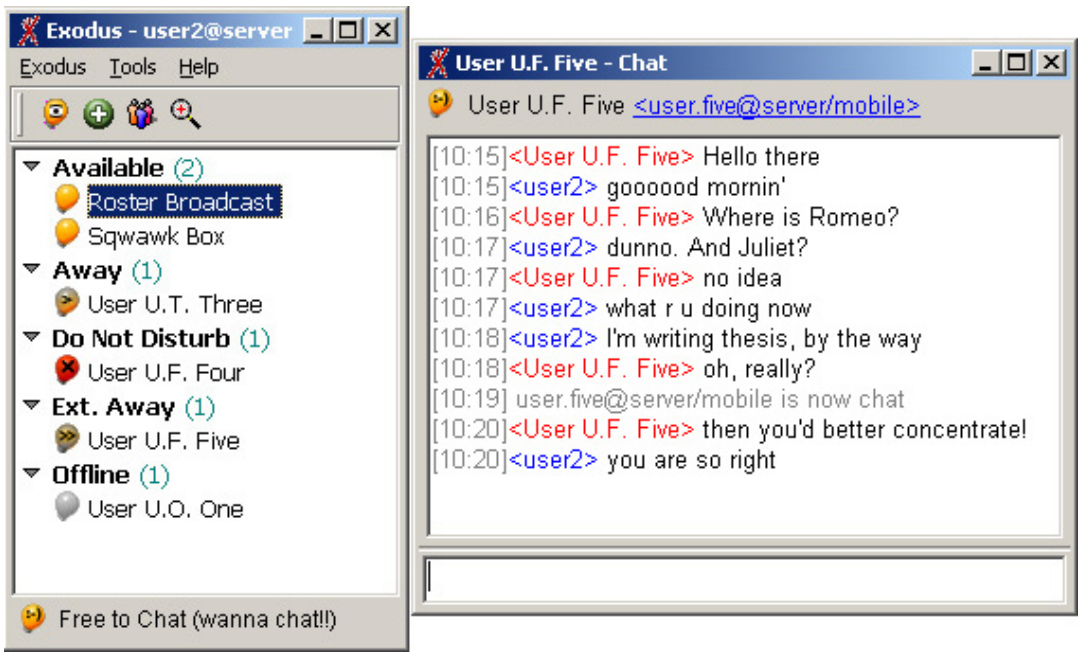

Figure 4.4. Exodus roster and chat window

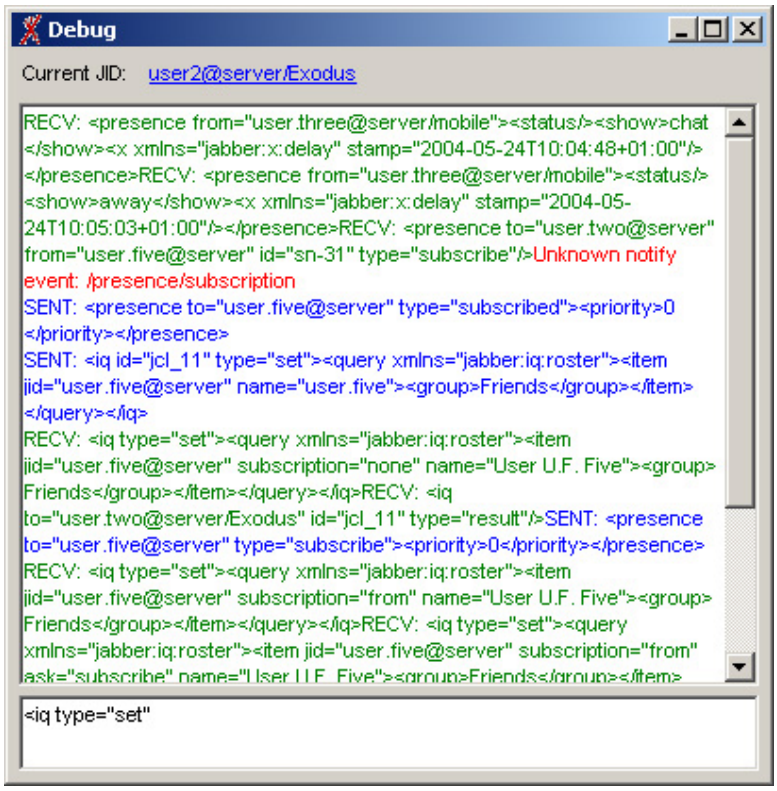

Figure 4.5. Exodus XML debug window

Chapter 5. Application Design

This Chapter describes the necessary steps in the design phase of application development.

5.1. Feasibility Check

First of all, the feasibility of a Jabber IM client must be examined in terms of presence service concept. The presence service aims at providing constantly accurate presence information using data from the HLR. This means that your client application will be "pushed" by the presence server, i.e. get an roster update notification, whenever one or more users on your roster

(a) turned their devices on/off, or

(b) changed their IM status, for instance, from "chat" to "away".

The Push Registry technology in the MIDP 2.0 makes it "*theoretically*" possible for the client on a mobile phone to receive incoming push connection. An emphasis is put on the word theoretically, because the implementation of some network protocols depends on the device as referred to in Section 3.2.4. Even if a protocol is supported on the device as an inbound connection type, it is not required to be enabled as a valid push mechanism. This is understandable particularly for security reasons: not all generic connections will be appropriate for use as push application transport. Although the target device implements the MIDP 2.0 and socket connection, its Developer Platform for Series 60 supports *only* SMS and Bluetooth as valid inbound connection mode for the the Push Registry.[23] Hence SMS is the only possibility for the presence server to contact its clients, i.e. if we definitely want to utilize Push Registry.

Suppose the client implemented the SMS Push Registry. The server hereby ought to send the push to a pre-defined, particular port so that the confusion with usual SMS can be avoided. The client registers that port for incoming SMS push on the device and therefore is invoked by a server push. The client is now responsible for opening the connection to the server and getting the updated roster. But we cannot simply get the roster without authenticating ourselves; the Jabber protocol requires proper session establishment before sending any XML stanza. If you exit the client very often (i.e. closing the connection down), this might become tricky, because you can be almost permanently prompted for

(1) permission to open up a connection by AMS, and

(2) login by the client MIDlet, in order to authenticate yourself with your user name and password.

To suppress (1), AMS offers the choice to select "Yes, always. Don't ask again." in the prompt display. With this setting, any additional Jabber event notifications sent to the device will not require answering this. Similarly, an implementation of automatic login setting might be helpful for (2).

Configurable update interval would help e.g. those who are not necessarily interested in the most accurate presence information. After all, the user is the one to pay for the occurred

[23] [Nokia]

data transmission. Less frequent update costs naturally less. The server should hold each user's configuration in order to meet their demands. This configuration would contain not only the regulation of server push frequency, but more fundamentally, the subscription state of the presence service. Exchanging these information is out of Jabber's specification scope, hence a different protocol and port number must be assigned for it. The conceivable requirements to the presence server can be summarized as follows:

1. First of all, to be a Jabber server that is fully compatible to the XMPP specification.
2. Ability to send SMS push to the client. A particular port number must be defined for this purpose.
3. Management and storage of presence service subscriber information with regard to subscription state and user preferences (e.g. push frequency configuration).
4. Definition of protocol and port number to realize the point listed in 3.

While the implementation of the presence service feature must wait for these specification, the client still needs some kind of roster update mechanism by itself. Although Jabber has the server push mechanism, its implementation is just a recommendation. To be sure, the client application will equip itself with a timer controlled automatism for roster update.

5.2. Requirements

The followings are to be implemented according to the priority as given:

1. Conformity with XMPP, especially full implementation of
   - `<message/>` protocol for messaging
   - `<presence/>` protocol for presence feature
   - `<iq/>` protocol for session establishment and roster management
2. Presence subscription mechanism according to XMPP specification
3. Possibility for auto-login
4. Timer based roster update mechanism with freely adjustable interval
5. Privacy list to enable the blocking of certain communication
6. Ideal: support for secure connection using TLS
7. Ability to receive server push from presence server

Though the Nokia 6600 does not set fixed limitation to the JAR size, the whole amount of available memory resource is limited. As a consequence the size of the MIDlet suite should be not too large. Needless to say, the richer the features, the larger the JAR file becomes. Since this work sets its goal on the basic function defined in Jabber protocol and not on the fancy, colorful user interface, the file size ought not to exceed 50 kilobyte.

5.3. General Specification

Grounded on the requirements as listed in the previous section, I made a table of specification (see Table 5.1). This acts as the guideline for the client implementation as well as a reference for the design of the presence server. The name "*IMpulse*" was given to the application, with a wish that this IM client gives an impulse to its users and does not become yet another boring application.

Application name	IMpulse
Target device	Nokia 6600 with the MIDP 2.0
Provisional Jabber server	• public Jabber server, e.g. jabber.org • Antepo OPN server on localhost • eventually: Alcatel's presence server
Connection type	• TCP socket connection, port number 5222 (default) • Option: secure socket connection, port number 5223
Data transmission	via GPRS
Basic functions	• **Communication:** 1 to 1 chat, group chat, e-mail like message • **Presence feature:** Subscription management, show status icon together with description text • **Roster management:** Add/remove a contact in the roster, show roster, update roster regularly
Configurable parameters	• Presence show as defined in XMPP: `online` (default) and `chat/away/xa/dnd` • Presence status as natural-language description • User name, Jabber server, resource (default: mobile) • Other users' profile: nickname, Jabber ID, group • Resource priority (default: 5) • Auto-login on/off (default: off) • Polling interval in minutes • Entries in the blacklist (in Jabber term: privacy list)

Table 5.1. Specifications of the mobile IM client application

The program flow should be simple and offer the choice to exit at any time. As Figure 5.1 shows, user's roster is going to be the main screen of this application. While logged in, constantly updated roster is shown. From the roster screen, users have the choice to add and delete a contact person (named often "buddy"), to write a message, to change own presence status and to set own preferences. As soon as an incoming instant message from a buddy arrives, it is displayed together with two choices, namely "reply" (write back to the sender) and "OK" (acknowledge and go back to roster).

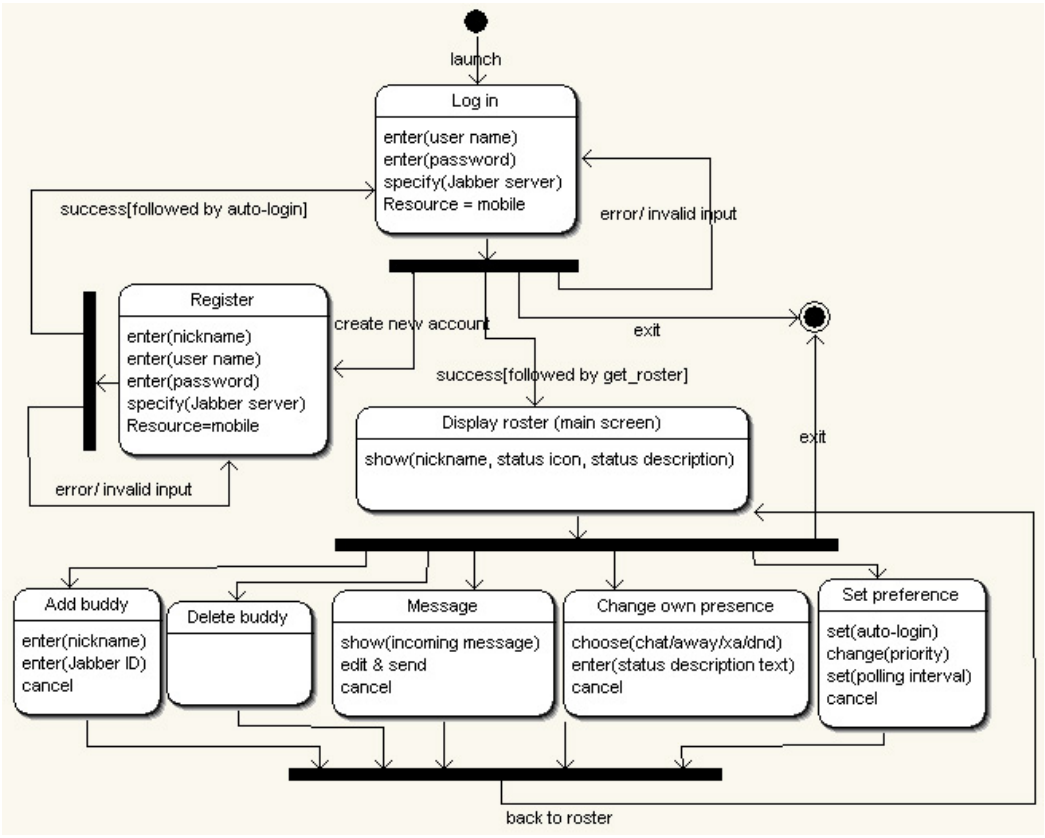

Figure 5.1. IMpulse program flow

5.4. Model View Controller Pattern

As explained in Section 3.2.3, only one `Display` object – the screen manager – can be assigned to a MIDlet. This separation of logic and appearance fits in the Model-View-Controller (MVC) pattern: a `MIDlet` instance as the *controller* and a class which holds a reference to the `Display` as the *view*. The *model* in the MVC pattern usually represents the persistent data storage, therefore a class that manages the `RecordStore` would be suitable for the model.

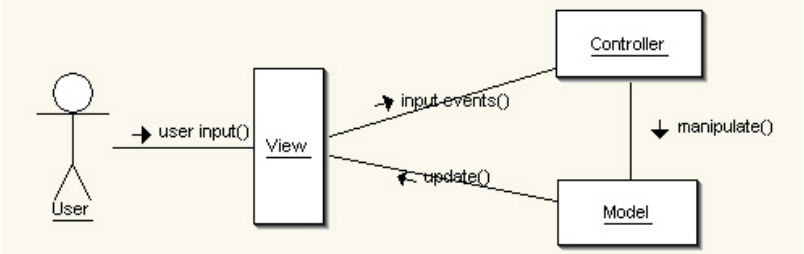

Figure 5.2. Model View Controller pattern

According to the MVC pattern, the whole MIDlet suite is divided into three packages. Each of them has, so to speak the chief class, `Model`, `View` and `Controller`, respectively. The class `Controller` is the MIDlet (i.e. the main class) which has its own Display (`View`) and `Model`. The `View` class acts like a view-manager switching visible screen from one to another, according to the user input and method call from the `Controller`. The `Model` class takes care of the persistent data storage by possessing a

private `RecordStore` object. All the classes which are relevant to the general XML data communication seemed to be not suitable in the controller package, hence I put them in a separate package named xml.

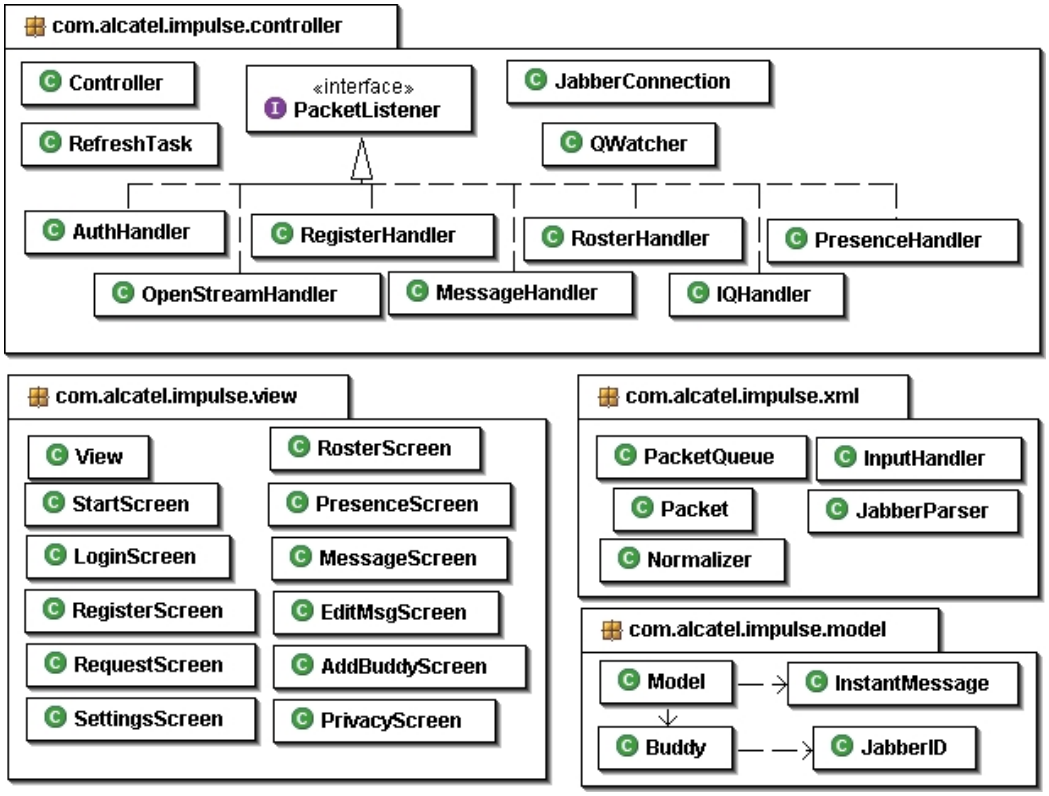

Figure 5.3. IMpulse grouping of classes

5.4.1. Package Details

Within a package, multiple classes interact in order to carry out the tasks assigned to that package. Inter-package communication is enabled through the three chief classes, since they hold references to each other. Some of the significant classes are described here.

The controller package

- *Controller* – a `MIDlet` subclass. Having the references to the `View` and `Model` instances, the Controller takes the control over the whole application.
- *RefreshTask* – a `TimerTask` subclass that handles regular roster update.
- *QWatcher* – keeps a watch on the `PacketQueue` which holds the incoming packets. As soon as a packet arrives, the QWatcher pulls it from the queue and passes it on to the right handler according to the packet's content.
- *JabberConnection* – responsible for the network connection from establishment till closing. The sending of Jabber packets is its task, too. One of its attribute, the `InputHandler` thread class deals with the receiving of packets.

- *PacketListeners* – each sort of packet, such as message, presence and info-query, is processed differently by the packet listener class which devotes itself to that particular sort.

The view package

- *View* – implements the `CommandListener` interface. So this class reacts to the user input and switches the screen accordingly.

The model package

- *Model* – manages the user information such as Jabber ID, password, roster items (i.e. "contact persons" or "buddies") and arrived messages. It holds a reference to the `RecordStore` instance, in which some essential data are saved.

- *Buddy* – represents the user's contact person in the roster.

- *InstantMessage* – incoming messages are represented by this class.

The xml package

- *PacketQueue* – a FIFO (first in, first out) queue of incoming packets.

- *InputHandler* – converts incoming XML stanza to the Jabber `Packet` and pushes it into the queue.

- *JabberParser* – an XML parser used by the `InputHandler`.

- *Packet* – represents a Jabber packet. This resembles somewhat the DOM `Node` class.

- *Nomalizer* – converts character format to and from UTF-8, for example '<' to '`<`'.

The main job of the xml package is to parse incoming XML stream. The upcoming chapter discusses this topic among others.

Chapter 6. Implementation

In this Chapter, I would like to describe some challenging tasks that I encountered during the implementation and how I managed them.

6.1. MIDlet Programming

It took some time until I got used to the way of J2ME programming, due to the numbers of restrictions imposed upon J2ME compared with J2SE. For usability reason and due to the limited display size, the commands should have concise name and well-thought placement order. The most commonly used functionality should appear first in the list. It was confusing though in the testing phase, that some emulators displayed the command priority in the completely different order. The MIDlet should be smoothly recoverable after it is interrupted by a phone call or a SMS message. Likewise, the device could suddenly lose its network connection while a MIDlet is running. Since this client application strongly depends on the networking, it must be able to cope with such circumstances. All in all, programming must be done differently from the standard J2SE applications. Besides, various test cases are crucial for developing high-quality MIDlet suite. It is regrettable that the emulator can never equal the real device and the debugging on the device is considerably difficult without any tracing possibility.

We saw in Chapter 3 that J2ME supports only a confined scope of classes available in J2SE, and that with fewer methods. To name a concrete example, there are neither `java.util.Collection` nor `java.util.Iterator`. Where I would normally use these classes, I used `java.util.Vector`, `java.util.HashTable` and `java.util.Enumeration`. This sort of rethinking was imperative during the whole process of implementation.

6.2. XML Parsing

The IM client's fundamental function lies in the communication with the server. It means, because IMpulse is a Jabber IM client, the capability to handle XML is indispensable. Sending XML output is not a problem, but interpreting XML input stream calls for an internal XML parsing system. Because we need to handle each XML stanza (described in Section 2.7) differently, it is to be desired that the parsing system transforms each stanza into a jabber packet for further processing. The general architecture regarding the XML communication is shown in Figure 6.1.

J2ME API does not include classes that can handle XML. Developers must whether write the XML parsing system by themselves or include a third-party XML parsing API. Due to the size restriction imposed on J2ME environment, only TinyXML,[24] nanoXML[25] and kXML[26] come into question as such API.[27] They are all small enough to be included in the MIDlet suite. To define which one suits our client application, or whether it is better to write the parser code by myself, thorough investigation has to be undertaken.

[24] http://www.grinninglizard.com/tinyxml/
[25] http://nanoxml.sourceforge.net/
[26] [kXML]
[27] [Kroll03]

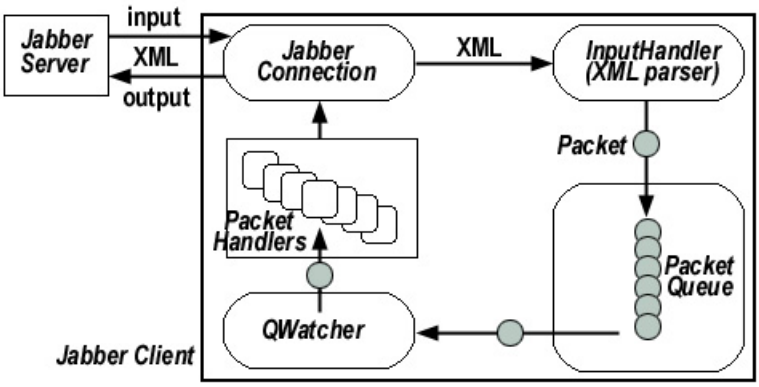

Figure 6.1. The client's architecture

6.2.1. How to Parse Efficiently

In general, there are three types of XML parsing techniques: Document Object Model (DOM), Simple API for XML (SAX) and Streaming API for XML (StAX) or in other words, Pull Parser Model.[28]Although DOM's tree-like data structure is desired in our Jabber packet to some extent, its XML representation is too generic to apply here. Besides, for DOM needs the whole document to be loaded, the choice narrows down between SAX and Pull model. Table 6.1 visualizes the differences between these three models in some aspects.

	DOM	SAX (Push)	StAX (Pull)
Streaming (Forward-only Access)	-	✓	✓
Random Access in memory	✓	-	-
Read-only Access	-	✓	✓
Event Based	-	✓	✓
Memory Consumption	high	low	low
XML Document Modification	✓	-	-

Table 6.1. Comparison of XML parsing technique

But how different are SAX and StAX? SAX is an event-driven push model for processing XML. A SAX parser fires off a series of events as it reads through the document. These events are pushed to event handlers, which provide access to the contents of the document. This way is also referred to as a "callback" mechanism. The disadvantage of SAX is that you have to implement the event handlers to handle all incoming events. You must maintain this event state in your application code. Because the SAX parser does not communicate meta-information such as DOM's parent/child support, you have to keep track of where the parser is in the document hierarchy. Even though there is no need to load the entire document into memory at one time, a SAX parser still needs to parse the whole document, as with DOM.

StAX uses an event-driven model like SAX. However, StAX uses a pull model for event processing rather than SAX's push model. Instead of using a callback mechanism, a StAX

[28][DevXml]

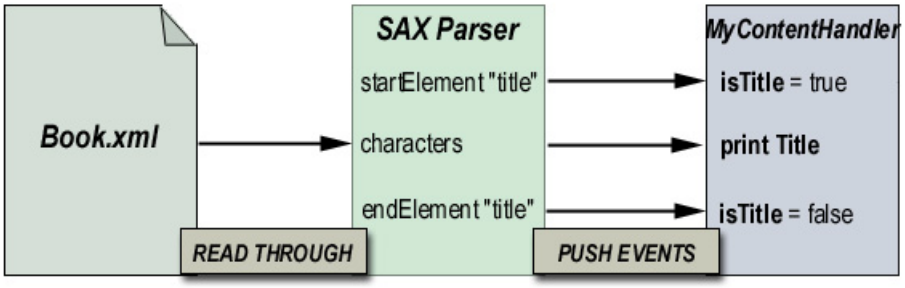

Figure 6.2. SAX's PUSH model

parser returns events as requested by the application. StAX also provides user-friendly APIs for read-in and write-out. Whereas SAX returns different types of events to the ContentHandler, StAX returns its events to the application and can even provide the events as objects. The characteristics of each parsing model follows:

SAX (push model)

- EventHandler class is needed
- Registration of multiple ContentHandlers is allowed
- No support for namespace scoping
- Certain context and state information (e.g. parents and depth) must be tracked

StAX (pull model)

- No Handler required, application has control of the parsing (i.e. easy to code)
- Support for multiple inputs and for namespaces
- No need to track application state
- Powerful filtering capability

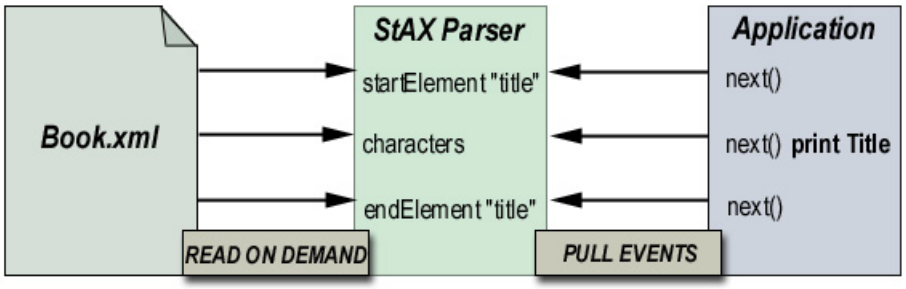

Figure 6.3. StAX's PULL model

Now let us come back to the three XML APIs that can be used in the J2ME environment. The NanoXML and TinyXML are both natively tree-based parsers while kXML is an event-based pull parser. kXML is the only API written specifically for the CLDC. All these parsers are becoming more complete over time with features such as DTD validation and Java interfaces for both tree and event-based parsing; of course at the price of memory footprint (up to 17 kB). In fact, the client's XML parsing system does not require such variety of features offered by these APIs. By writing a no-frills parser on my own, several kilobytes can be cut down easily. Hence I decided first to write an event-driven push parser that calls the `InputHandler`. After this goal was achieved, the reference implementation using kXML parser and its promising pull model was made for the comparison.

StAX parsing satisfies the majority of a SAX application's requirements, so if an application is well suited for one of these two models, it should go well with the other. This was proven in this experiment.

6.2.2. KXMLParser

kXML 2 is a small XML pull parser, specially designed for constrained environments such as Applets, Personal Java or MIDP devices. kXML 2 is based on the common XML pull API.[29] kXML key features are:

- XML namespace support
- Small memory footprint
- A pull-based parser for simplified parsing of nested / modularized XML structures
- XML writing support including namespace handling

In kXML, the `public int KXmlParser.next()` method returns one of the predefined event types: START_TAG, TEXT, END_TAG and END_DOCUMENT. When START_TAG (or END_TAG) occurs, the element name can be retrieved using the `getName()` method, likewise the `getText()` method delivers the text content on the TEXT event. The current event state of the parser can be determined by calling the `getEventType()` method. You can parse through the XML document simply using these few methods, there are many other methods available though. An example code might make this simplicity more obvious:

```
HttpConnection conn = (HttpConnection)
                Connector.open(http://example.org/easy.xml);
KXmlParser parser = new KXmlParser();
parser.setInput(new InputStreamReader(conn.openInputStream()));
while (parser.nextTag() != parser.END_TAG) {
     String name = parser.getName();
     String text = parser.nextText();
     System.out.println("<" + name + ">" + text);
}
```

With the help of the kXMLParser, the reprogramming of the XML parsing system from SAX to StAX was easily and quickly done. kXMLParser's useful methods to get the current depth or attributes' name/value, made the program code quite straightforward. The JAR size has grown 5 kilobyte, this is still acceptable for our target device. But the problem happened in the test run – the parser did not execute the `next()` method, although it was successfully created and a valid inputStreamReader was assigned. Due to the time limit given for this work, I had to give up the idea of debugging this issue; after all my simple push parser was running without a problem under the same conditions.

6.2.3. Final Implementation

As shown in Figure 6.4, the job of the client's XML parsing classes is to read input stream from the server, convert it to `Packet` objects, and store them in the `PacketQueue`.

[29] An open source API for XML Pull Parsing, available at http://xmlpull.org/

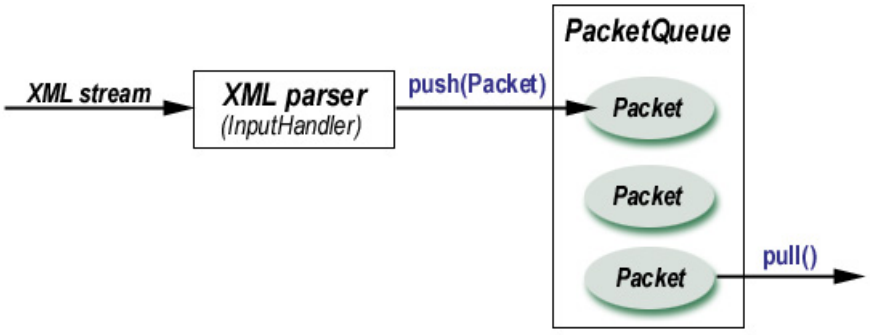

Figure 6.4. The client's XML parsing subsystem

6.2.3.1. Mapping XML fragments to Objects

The `Packet` class represents the Jabber XML fragment. The incoming packets to be recognized by the clients are limited, namely:

- standard Jabber XML stanzas – `<message/>`, `<presence/>` and `<iq/>`.
- `<stream:stream>` – The opening XML stream tag for the server's part, corresponding to the previously sent `<stream:stream>` tag. This signals the start of a Jabber XML session. The closing XML stream tag, `</stream:stream>` would never come because the server closes the connection immediately after the client's closing stream tag arrives there.
- `<stream:error></stream:error>` – A Jabber stream error packet. This packet is used to indicate a problem with a stream (usually indicating an improperly formatted stream).

Apart from the support for these tags, the `Packet` class must perform two primary tasks:

- *Packet data store* – the `Packet` class is primarily a data storage class. It is absolutely necessary for the further process.
- *XML writer* – the `Packet` class must know how to create its own XML string representation. This feature allows other classes to generically convert `Packet` objects to `String`, so that they can be streamed to the server.

The data structure of a `Packet` class mirrors exactly the structure of XML fragments. For example, consider an XML packet:

```
<message to='romeo@example.net'>
   <subject>Greeting</subject>
   <body>Happy birthday to you! All the best.</body>
</message>
```

The `Packet` class maps it into three packets arranged in a tree structure:

```
Packet: message (attribute to='romeo@example.net')
     Packet: subject
          String (value "Greeting")
```

```
    Packet: body
         String (value "Happy birthday to you! All the best.")
```

As we can see, a `Packet` has an element name, zero or more attribute name/value pairs, and zero or more children. A `Packet` object's children are either `String` or other `Packet` objects. In addition, each `Packet` has an associated namespace. The storage of a `Packet`'s children is realized using a `java.util.Vector`, its attributes using a `java.util.HashTable`. The `Packet` class also keeps track of its parent packet. If a packet has no parent, the `Packet`'s parent variable is set to `null`.

6.2.3.2. Queuing Mechanism

The `Packet` objects are stored in a plain vector in the `PacketQueue` object. The `PacketQueue` is an uncomplicated class with its only two methods, namely pull and push. However, for it plays a central role in the XML parsing subsystem, the requirement set to the `PacketQueue` is significant:

- *Store Packets* – accept `Packet` objects pushed onto the back of the queue, maintaining the order of items pushed.
- *Retrieve Packets* – allow `Packet` objects to be pulled from the front of the queue, removing the item.
- *Be thread-safe* – allow multiple threads to push and pull `Packet` from the queue without problems.
- *Provide thread synchronization* – if the queue is empty, attempting to pull `Packet` from the front will cause the caller to wait until a new `Packet` is pushed onto the queue or the thread is interrupted.

The methods `push(Packet)` and `pull()` are *synchronized* to protect them from being executed by overlapping threads. Every push method call invokes `notifyAll()` to wake up any threads waiting on a pull method call. The pull method blocks the caller thread by its internal `wait()` method call while the queue is empty.

6.2.3.3. Packet Handling

So far we have seen the three main entities in the client's XML parsing subsystem: the XML parser, the `Packet` class and the `PacketQueue`. The client application works only when all of these entities cooperate hand in hand, interacting with the other classes smoothly. Figure 6.5 outlines the basic input/output operation of the client application in the form of a sequence diagram.

❶ `Controller` MIDlet creates a `QWatcher` thread as its attribute. The thread starts automatically on its creation.
❷ `QWatcher` creates a `PacketQueue` object as its attribute. From then on `QWatcher` watches whether the queue holds a packet to be pulled out. If the queue is empty, this thread will wait until another thread notifies the arrival of a new packet.
❸ At user's request, `Controller` calls the `JabberConnection.connect()` method. By the way, the `JabberConnection` object is created as a `Controller`'s attribute.

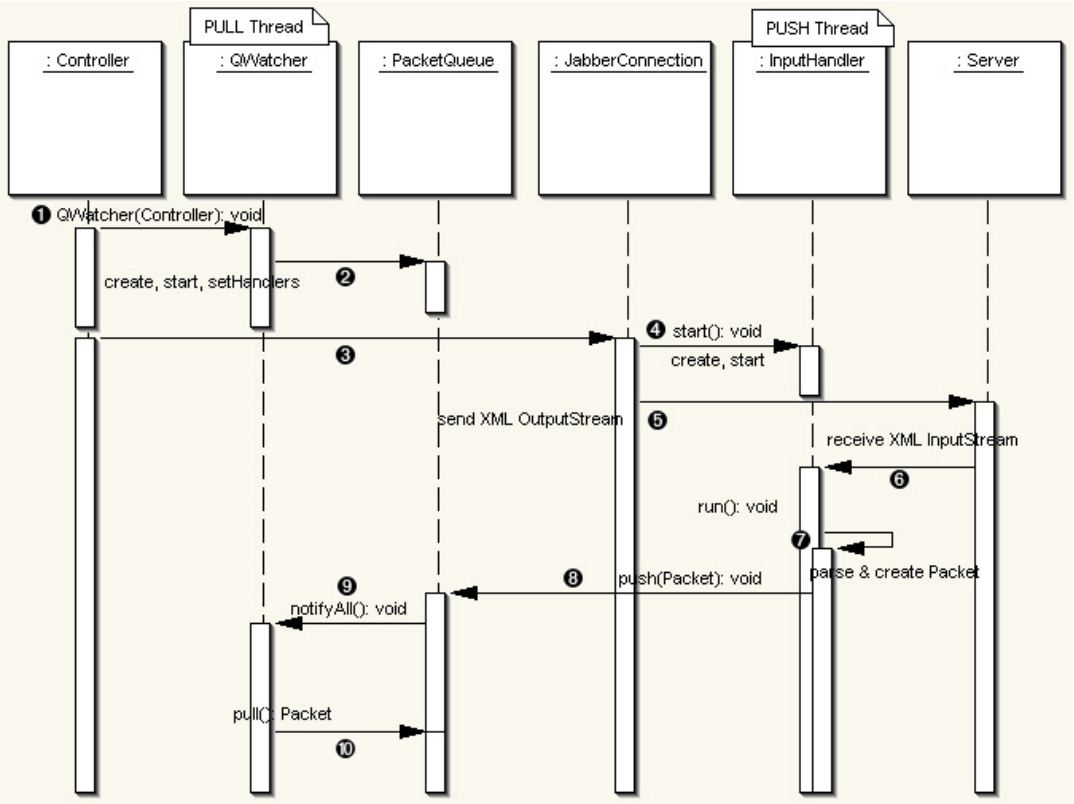

Figure 6.5. Sequence diagram of XML parsing mechanism

❹ `InputHandler` thread is started within the `JabberConnection.connect()` method. This thread creates a `JabberParser` thread on its own, which parses the incoming stream and calls each event back to the `InputHandler`.

❺ JabberConnection starts sending the Jabber output stream to the server.

❻ The response from the server arrives.

❼ The input stream is parsed by the `JabberParser` thread, and the `InputHandler` builds Jabber packets.

❽ Once the Packet is done, the `InputHandler` calls the `push(Packet)` method offered by its attribute `PacketQueue`.

❾ As soon as a `Packet` is saved in the queue, the `PacketQueue` broadcasts this event using the `notifyAll()` method.

❿ Upon the `notifyAll()` notification, `QWatcher` thread wakes up and pulls the packet from the queue. According to the sort of the packet, `QWatcher` forwards the packet to the correct packet handler.

6.3. Multi Threading

As described in Section 6.2.3.2, to make the methods *synchronized* is one measure to address the thread concurrence. Synchronizing threads will slow your application, so it is recommended to minimize the points of contention whenever possible. In our client application, only the `PacketQueue` applies this measure for its push and pull methods. But the client contains more threads than just those two.

In order to ensure that the user interface remains responsive, any methods called within the `CommandListener.commandAction()` method[30] must return quickly. Especially network I/O operations, that tend to be slow, must not be called within this method, because this would freeze up the user interface and the users are left behind helplessly. In consequence, MIDlet suites with networking are constrained to be multi-threaded, always starting network operations on a separate thread from the event thread. Not only does it guarantee a friendly, non-blocking user interface, but it also enables to display and update the operation's progress.

In our application IMpulse, the login procedure takes especially long time for its completion, since it involves the following series of actions:

1. Opening a connection to the server
2. Sending the login information
3. After successful login, retrieving the roster from the server
4. Sending presence update
5. Processing the incoming packets and rendering the roster

The first two actions are executed in a `ConnectThread`, which is an inner class of `Controller`. After the creation of this thread, the `Controller`'s `processLogin()` method returns immediately, allowing the user interface to show an information screen as pictured in Figure 6.6. The rest of actions listed above are done by several `PacketHandler`s upon arrival of corresponding packets from server.

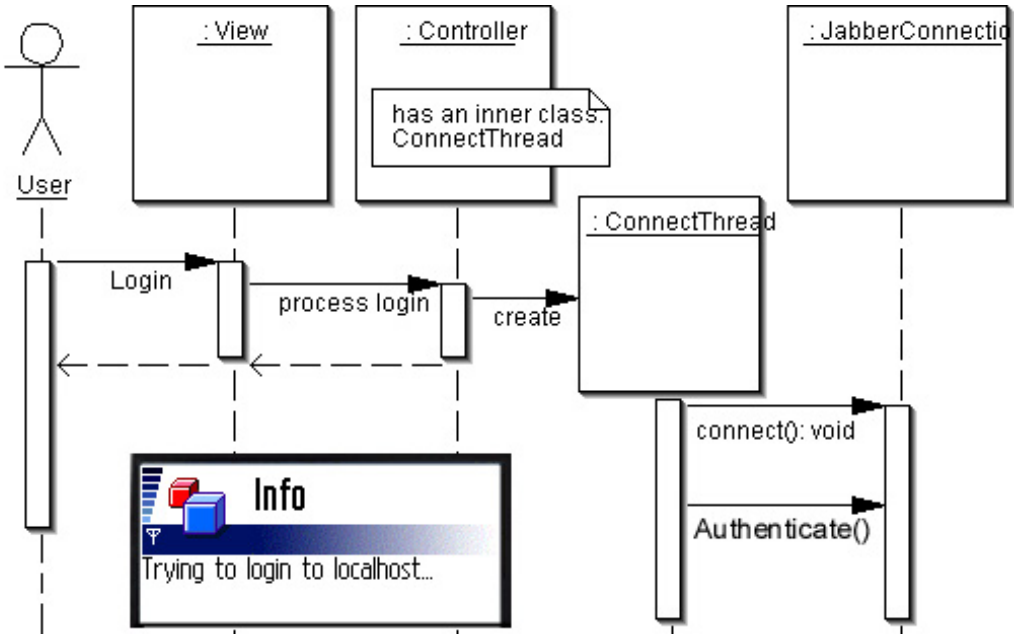

Figure 6.6. sequence diagram of CommandListener

With this implementation, a smooth, relatively (subject to the condition of network/server) quick login was attained. A thread is not halted or notified by the AMS when the application changes to a paused or destroyed state; this is the task of the application developer. MIDlet should always try to terminate threads cleanly before `destroyApp()` completes. This can be done by calling `Thread.join()` method.

[30] `CommandListener.commandAction(Command c, Displayable d)` method is invoked when user triggers a button. See Section 3.2.3.2 for more information.

6.3.1. Using Timers and TimerTasks

Applications running under the MIDP can schedule code for one time or periodic execution using the `java.util.Timer` class. The `Timer` class is thread-safe: multiple threads can share a single `Timer` object without the need for external synchronization. Usually, a MIDlet creates one `Timer` to schedule all its `TimerTask`s, so that these tasks are executed sequentially in a dedicated background thread. Thus, timers can be helpful to reduce the use of the keyword *synchronized* in your code. It is possible to have more than one timer in a MIDlet. However, the care should be taken when using multiple timers. Because each timer runs its own thread, again the synchronization may be required.

In the client application, the periodic roster update is scheduled by a `Timer` object. Initially the update period is set to five minutes, but the users can change this value in the preference screen. Scheduling is caused by the `Timer`'s schedule method, for example:

```
Timer myTimer = new Timer();
RefreshTask task = new RefreshTask(); //a TimerTask subclass
myTimer.schedule(task, 0, 20000); //execute every 20 seconds
```

Similar to threads, the `TimerTask` and the `Timer` need to be canceled upon termination of the MIDlet.

6.4. Subscription State Management

Presence subscription is of great importance in order to register your IM contacts onto your personal roster. Section 2.7.2 explained how to request and accept a presence subscription through the `<presence/>` stanza. However, what the `presence` protocol can handle is merely "*one-way*" subscription. Instant messaging makes more sense if you have *mutual* subscriptions to your contacts, i.e. not only you have subscriptions to their presence but also they have subscription to yours. With a view to achieving mutual subscriptions, the `iq` protocol works closely together with the `presence` protocol.

As introduced in Section 2.7.3, `<iq/>` stanza is mainly used for roster management. Let us cast our minds back to the example `<iq/>` packet from the server, which represents a roster:

```
RECV: <iq to='lala@example.com/work'
RECV:    type='result'
RECV:    id='roster_1'>
RECV:   <query xmlns='jabber:iq:roster'>
RECV:     <item jid='dinkywinky@example.net'
RECV:           name='Dink'
RECV:           subscription='both'>
RECV:         <group>Friends</group>
RECV:     </item>
RECV:     <item jid=dipsy@example.org'
RECV:           name='Dipsy'
RECV:           subscription='from'>
RECV:         <group>Work</group>
RECV:     </item>
RECV:     <item jid='po@example.net'
RECV:           name='Po'
```

```
RECV:           subscription='to'>
RECV:         <group>Friends</group>
RECV:     </item>
RECV:   </query>
RECV: </iq>
```

As you can see, the state of the presence subscription in relation to a particular roster item is captured in the '`subscription`' attribute of the `<item/>` element. Allowable values for this attribute are:

- "*none*" – the user does not have a subscription to the contact's presence information, and the contact does not have a subscription to the user's presence information.
- "*to*" – the user has a subscription to the contact's presence information, but the contact does not have a subscription to the user's presence information.
- "*from*" – the contact has a subscription to the user's presence information, but the user does not have a subscription to the contact's presence information.
- "*both*" – both the user and the contact have subscriptions to each other's presence information. This is usually the most preferable subscription state in a instant messaging system.

Once a contact has the subscription state "to", there is nothing more to do for the user than just to wait for a subscription request from that contact. For the client application to help the transition toward the mutual subscription, the key lies in the "from" state. Whenever a contact's subscription state is "from", IMpulse sends automatically a subscription request to that contact in the hope that the contact accepts it and thus a mutual subscription comes off. This mechanism is implemented in the `RosterHandler` class. [XmppIM] describes the process of subscription state management in detail, which helped the implementation to a large extent.

6.5. Security

However convenient and popular instant messaging may be, or exactly because of its popularity, IM systems are prone to be dangerous in terms of network security. Owing to the wide propagation of (especially commercial) IM systems, the harm can be disastrous. In fact, so-called "*IM worm*" that spread over the IM network has been causing serious damages in recent years. Mostly this kind of problem arises from the file transfer feature. Our client does not offer this feature, because XMPP has not specified such kind of feature yet. Other security threats of IM are:

- *Firewall piercing*: happens often in company network when unwitting user opens IM channel free for attackers.
- *Sniffing* and *data tampering*: if no data encryption is available, the IM traffic travels over the network in clear text and hence can be easily monitored/modified.
- *Spoofing* or *identity theft*: attackers gain unauthorized access to the system by pretending to be someone else and creating fake responses.

The risk of the latter two can be minimized by data encryption or secure connection to the server, provided your contact also takes those precautions. XMPP includes two methods for securing the stream: the use of TLS for channel encryption and the use of SASL for stream authentication. In order to implement these, the client MIDlet suite requires an API for cryptographic algorithms.

Since the MIDP does not include the `java.security.*` package, third-party API must be sought. An open source project, the Bouncy Castle Crypto package[31] provides a light-weight API suitable for use in J2ME. As the security was not the top priority in this work (see Section 5.2), there was just enough time to implement the digest authentication as an alternative to the plain text authentication. The digest authentication takes the session ID from the server's initial `<stream:stream>` tag, concatenate it with the user's password and then hashes the resulting string using the SHA-1 message digest algorithm. In this way the user's password is protected from prying eyes on the network.

[31] [Bouncy]

Chapter 7. Conclusion

7.1. Development Process

At the beginning of this work, the primary idea was to extend the client code which was written in a preceding master thesis.[32] Soon this idea turned out to be impracticable and I had to start from scratch, since the code was copyrighted and was not meant to be an open source project. Yet the author, Tobias Frech kindly disclosed the code to me as a referential work. This was a great aid to get into the thematic. Hereby I would like to express my gratitude for his support. Eventually this reorientation was beneficial, because so I could gain a profound knowledge about J2ME/Jabber technologies and self-confidence to have finished a software project on my own.

I must say that to create a client without the corresponding presence server was an intangible task with full of speculations. The presence server project was entirely under the influence of the Alcatel's circumstances. To my regret, there was no opportunity to communicate with the developers of the presence server. All I could do for the server was to list up the conceivable requirements from the client's point of view. Meanwhile I shifted the goal of this work to create a simple XMPP conform client, and this was tough enough. The investigation phase together with the preparation of the development environment took me a considerable amount of time. One reason was that this work called for the exhaustive inquiry into the protocol specification. Actually, I have gathered some experiences with chat application development during my studies. But at that time I could utilize Message Oriented Middleware (MOM) and Java Messaging Service (JMS) API which conceal the low-level matters like protocol. XMPP Core and IM specifications were approved by IETF as a proposed standard early in February, it was just in time for my work. With this approval, I had the assurance that it would not be changed in some weeks. The Jabber's simplicity was of assistance throughout this work.

In the implementation phase, I learned the MIDlet programming just by doing which opened me a completely new aspect of application programming on the constrained environment. After ten weeks of intensive coding and testing, I accomplished the mobile Jabber client application *IMpulse*. Some test cases showed the necessity for the improvement, though it was to be expected – it was not brought to perfection particularly in terms of optical design. In addition, not all the thinkable test cases were covered due to the time frame. Nevertheless it works fine with any XMPP conform Jabber server. When the coding was finished by and large, I started to write this work.

7.2. Achievement

The presence server project is taking longer than expected; it has not been established as of this writing. The presence service system was consequently not realized during this work and IMpulse could not become the first mobile IM client for that service. At all events I would consider this work as successful, for the primary goal was reached with the functioning client application. IMpulse could fulfill five requirements out of seven as listed in Section 5.2. Group chat is supported, although it is defined in the Jabber IQ extension protocol, not in XMPP. In order to be fully XMPP conform, the support for

[32] [Frech03]

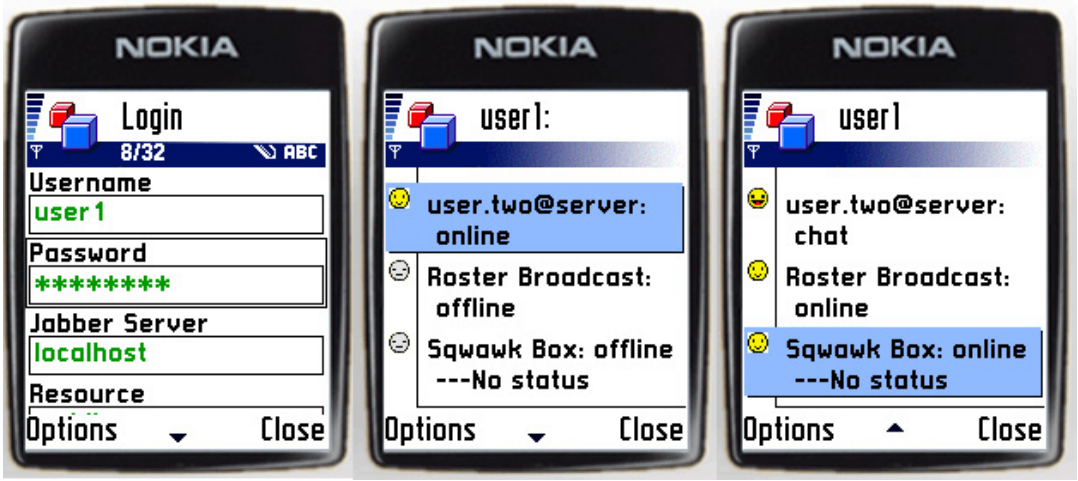

Figure 7.1. IMpulse running on Nokia emulator

TLS and SASL is still to be implemented. IMpulse's `PacketHandlers` are designed in modules allowing flexible extensions, for example in case that XMPP defines new features. The strings used in the user interface are separated from the application logic and saved in the `Config.java` file. Modification to the button labels or information text can be done easily by editing this file. This would be useful for the internationalization/localization.

Without the overall support of Alcatel, this work would never have been achieved. I am very grateful that I could have this great chance. I want to thank especially Rodolfo López Aladros and Nina Koestering from Alcatel for giving me this unique opportunity and helping me this work happen. Rodolfo, my mentor of this work, always stood by me in word and deed. My gratitude goes also to Prof. Dr. Martin Goik who kindly assisted me whenever I needed a help.

7.3. Similar Products on the Market

It is always good to compare own work with the similar applications on the market. Table 7.1 shows available J2ME based Jabber client applications according to [Jabber]:

Client	License	Platform
TipicME	Freeware	J2ME, PalmOS, PocketPC, Symbian
Agile	Freeware	J2ME, PalmOS, Symbian, WinCE
Colibri	Freeware	J2ME
mjabber	Freeware	J2ME
Rhombus IM Mobile Client	Commercial	J2ME, PalmOS, Symbian
Jabber Messenger J2ME	Commercial	J2ME
KomKom	Commercial	J2ME
MiMessenger	Commercial	J2ME

Table 7.1. Jabber clients written for J2ME platform

TipicME was newly released last April and is the latest client application. At the first glance, its colorful user interface is quite appealing. There are plenty of fancy avatars available which make the appearance of your roster amusing. This feature is certainly reflected on the JAR size (103 kB). The navigation is not intuitive because it uses the small icons, not the soft key buttons. Besides the icons have no labels. One more problem is that it does not react to the "Select" key on Nokia 6600, instead, the number "5" key (indeed, it took some time to find this out!). Probably it was developed only on emulators, which behave differently from the real devices. I by myself had also difficulty with that fact during the testing of IMpulse. Except for the user interface, the biggest difference between TipicME and IMpulse is that TipicME has fixed Tipic server to connect, while IMpulse users can change the Jabber server freely as they wish.

7.4. Feature Tendencies

Instant Messaging is today at its turning point: the time is ripe for the standardization. Jabber, with its IETF approved XMPP, is cutting a good figure as the driving force of this movement. Even the IM giant America Online shows the interest in "openness" and announced last April a new version of its ICQ, based on an open programming interface known as Xtraz. However, its platform is limited to Windows and its protocol remains as black box, this is far from being an truly open foundation for interoperability. Another urging demand on IM is the security. Especially business-related IM traffic needs to be protected against danger as stated in Section 6.5. XMPP addresses this issue with the use of TLS and SASL, this will surely contribute to make Jabber networks more secure. Enterprise IM systems such as Lotus Sametime[33] can be deployed to manage, secure and archive all IM traffic. A secure IM system for business may be implemented only with company policies which strictly prohibit the use of all other IM clients.

The presence awareness of mobile user is no longer a dream. Today people are still waiting for (and working on) standardization, but once it matures, this will be one of the hottest feature and become a matter of course. In Jabber, the server is the one to hold user's roster and privacy lists, materializing the *thin client* concept. This concept matches perfectly with the J2ME's restricted environment. Together with the prevalence of the Jabber protocol, there is no doubt that the number of Jabber IM clients for mobile phone will increase rapidly.

Another effort to bring a unified IM/presence protocol is *SIMPLE* (SIP for Instant Messaging and Presence Leveraging Extensions). SIP is an IETF standard signaling protocol for IP-communications, enabling IP-Telephony gateways, client endpoints and other communication systems or devices to communicate with each other. SIP primarily addresses the call setup and is independent of the actual data transmission after the session establishment. Notable is the SIP mobility feature that allows to locate mobile user with varying IP address, using the JID-like SIP address. SIMPLE leverages this feature to offer IM/Presence for mobile users. The IETF approval process for SIMPLE is running in parallel with that for XMPP, SIMPLE is expected to become a proposed standard this year. Supporters of both XMPP and SIMPLE are working on open interoperability between the two. Whether XMPP-based or SIMPLE-based, more and more people and corporations are definitely embracing standard-based solutions.

[33] [IBM]

Finally, I would like to mention the new movement toward IM standardization in Java Community: *JAIN Presence* (JSR 186)[34] and *JAIN Instant messaging* (JSR 187).[35] Both are intended to be the standard API beyond the existing industry specifications such as SIMPLE, Jabber, AIM, MSN, Yahoo! etc. As for developers, these APIs provide standard framework for developing and deploying new Java IM/Presence applications without a prior knowledge of the underlying protocol. This means an enormous improvement in the application development process, for learning of protocol takes its time as my work proves. Furthermore, developers will be able to implement interoperable applications that can run over a wide variety of protocols. JAIN stands for Java API for Integrated Networks. It is an interface for wireless devices that provides a uniform interface to wireless, traditional Internet access, the public switched telephone system, and ATM (Asynchronous Transfer Mode). The JAIN community is eagerly accelerating the standardization of APIs, such as JAIN SIMPLE, JAIN PAM (Presence, Availability and Management, JSR 123) and JAIN SLEE (Service Logic Execution Environment, JSR 022). JAIN SIP (JSR 032) is one of already standardized APIs. All these frameworks aim at enabling IM/Presence service for everyone, including the mobile users.

[34] Java Specification Request 186 [http://jcp.org/jsr/detail/186.jsp]
[35] Java Specification Request 187 [http://jcp.org/jsr/detail/187.jsp]

Glossary

A

avatar
: A graphical icon that represents a real person in a cyberspace system. When you enter the system, you can choose from a number of fanciful avatars.

C

chat
: Realtime communication between two users via computer. Once a chat has been initiated, either user can enter text by typing on the keyboard and the entered text will appear on the other user's monitor. Most networks and online services offer a chat feature.

I

ICQ
: Stands for "I-Seek-You". An online instant messaging program used as a conferencing tool by individuals on the Internet to chat, e-mail, perform file transfers, play computer games, and so on.

IM, Instant Messaging
: A type of communications service that enables you to create a kind of private chat room with another individual in order to communicate in real time over the Internet, analogous to a telephone conversation but using text-based, not voice-based, communication. Typically, the instant messaging system alerts you whenever somebody on your private list is online. You can then initiate a chat session with that particular individual.

IRC
: Short for Internet Relay Chat, a chat system developed by Jarkko Oikarinen in Finland in the late 1980's. IRC has become very popular as more people get connected to the Internet because it enables people connected anywhere on the Internet to join in live discussions. Unlike older chat systems, IRC is not limited to just two participants. To join an IRC discussion, you need an IRC client and Internet access. The IRC client is a program that runs on your computer and sends and receives messages to and from an IRC server. The IRC server, in turn, is responsible for making sure that all messages are broadcast to everyone participating in a discussion. There can be many discussions going on at once; each one is assigned a unique channel.

J

Jabber
: The XML-based open standard protocol for the instant messaging, its core is approved by the IETF under the name of XMPP. One of the most remarkable advantage is the interoperability with other major proprietary IM systems through gateways.

J2ME
: Java 2 Platform Micro Edition (J2ME) is a subset of Java 2 Platform Standard Edition (J2SE) that is geared toward embedded and handheld devices that cannot support a full J2SE implementation.

JAD
: Java Application Descriptor. A simple text file containing information that helps users to decide whether to download and install a MIDlet suite or not.

JAR
: Java Archive. Contains one or more class files and other resources.

JSR
: Java Specification Request, used in the *Java Community Process*. A JSR is submitted with a specific proposal to extend the Java platform. If it is accepted for development, an *Expert Group* (EG) is formed to define a formal specification for the JSR.

K

KVM, K Virtual Machine
: A small-footprint Virtual Machine that satisfies the CLDC requirements. The "K" stands for "kilo" because its memory budget is measured in kilobytes (whereas desktop systems are measured in megabytes).

M

MIDP
: Mobile Information Device Profile. A CLDC-based profile for devices such as mobile phones and PDAs.

O

OTA
: Over the Air. An abbreviation used in "Over-the-Air provisioning" of MIDlet suite.

P

presence
: The ability to detect whether other users are online and whether they are available. Presence services are commonly provided through applications like Finger (a UNIX

program) and instant messaging clients, although quite a few companies are developing products in other areas that leverage presence, such as Voice over IP (VoIP).

S

SIP, Session Initiation Protocol
: Protocol for Internet conferencing, telephony, presence, events notification and instant messaging. The protocol initiates call setup, routing, authentication and other feature messages to endpoints within an IP domain.

SIMPLE
: Stands for Session Initiation Protocol(SIM) for Instant Messaging and Presence Leveraging Extensions, meaning an application of the SIP protocol for server-to server and client-to server interoperability in IM. A step in bringing standardization to IM systems.

SMS
: Short Message Service is the transmission of short (no longer than 160 characters) text messages to/from a mobile phone, fax, IP address.

U

URL
: Uniform Resource Locator. A name and address for an existing object accessible over the Internet. An example of a URL is: `http://www.jabber.org`

X

XML, Xtensible Markup Language
: The Extensible Markup Language, a subset of SGML designed specifically for the user over the Web.

XMPP, Xtensible Messaging and Presence Protocol
: An open, XML-based protocol for near real-time extensible messaging and presence. It is the core protocol of the Jabber IM and presence technology which is currently deployed on thousands of servers across the Internet and is used by millions of people world wide. The most recent approval by the IETF as a proposal standard took place on January 2004 for the core features part, for the part with regard to instant messaging and presence functionality on February 2004.

Resources

Internet Drafts

[XmppCore] Peter Saint-Andre. Copyright © 2004. *Extensible Messaging and Presence Protocol (XMPP): Core*. Available at *XMPP Working Group web site within IETF* [http://www.ietf.org/html/charters/xmpp-charter.html]. January 20, 2004.

[XmppIM] Peter Saint-Andre. Copyright © 2004. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. Available at XMPP Working Group web site or at *Jabber Software Foundation web site* [http://www.jabber.org/ietf/]. January 30, 2004.

Books and Printed Resources

[Frech03] Tobias Frech. Copyright © 2003. *Mobile / Wireless Applications mit J2ME am Beispiel eines Instant-Messaging-Clients*. Master Thesis, Hochschule der Medien.

[Kroll03] Michael Kroll and Stefan Haustein. Copyright © 2003. *J2ME Developer's Guide*. Markt+Technik Verlag. 3-8272-6509-6.

[Ortiz01] C. Enrique Ortiz and Eric Giguère. Copyright © 2001. *Mobile Information Device Profile for Java 2 Micro Edition*. John Wiley & Sons, Inc. 0-471-03465-7.

[Topley02] Kim Topley. Copyright © 2002. *J2ME in a Nutshell*. O'Reilly & Associates, Inc. 0-596-00253-X.

Online Resources

[Jabber] *Jabber Software Foundation Web site* [http://www.jabber.org].

[MIDP2.0] *Mobile Information Device Profile 2.0 Specification (JSR-118)* at *JCP Home* [http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html].

[OReilly] DJ Adams. Copyright © December 2001. *Programming Jabber*. Sample chapter (Chapter 5: Jabber Technology Basics) is available at *O'Reilly Online Catalog* [http://www.oreilly.com/catalog/jabber/]. O'Reilly & Associates, Inc. 0-5596-00202-5.

[J2ME] Diverse Data Sheets and Specifications of J2ME are available at Sun's *J2ME Documentation Web site* [http://java.sun.com/j2me/docs/index.html].

[Shigeoka02] Iain Shigeoka. Copyright © 2002. *Instant Messaging in Java*. The Jabber Protocols. Available at *Manning Publications Web site* [http://www.manning.com]. Manning Publications Co. 1-930110-46-4.

[Cox04] Glen Cox. Copyright © 2004. *Presence Technology*. An article in *TANgents, a quarterly publication of Technology Across Nebraska* [http://extension.unl.edu/tangents/tangents_bitbybit2-04.htm/]. University of Nebraska Cooperative Extension et al.

[Nokia] Diverse Developer Information and Documentation for each Device Series are available at *Forum Nokia* [http://www.forum.nokia.com].

[DevXml] Ping Guo, et al. Copyright © 2004. *Parsing XML Efficiently*. Oracle Corporation [http://otn.oracle.com/oramag/oracle/03-sep/o53devxml.html].

[IMplanet] News, Reports and FAQ all about IM, including Wireless IM: *Instant Messaging Planet* [http://www.instantmessagingplanet.com].

[CNodes] A very helpful Online Lexicon of IT and Networking Terms: *CrossNodes* [http://networking.webopedia.com/].

Tools and IDEs

[WTK21] *J2ME Wireless Toolkit Version 2.1* (Release Date: 11.12.2003) [http://java.sun.com/products/j2mewtoolkit/download-2_1.html].

[ProGuard] WTK obfuscator plug-in: *ProGuard* [http://proguard.sourceforge.net].

[Eclipse] *Eclipse IDE* [http://www.eclipse.org].

[Nokia] *Nokia Developer's Suite for J2ME 2.0* [http://www.forum.nokia.com].

[Plugin] J2ME Plug-in v1.0 for Eclipse IDE available at *SIPtech* [http://www.siptech.com].

[kXML] *kXML 2 Pull Parser* [http://kxml.enhydra.org/].

[Bouncy] The Bouncy Castle Crypto package by the *Legion of the Bouncy Castle* [http://www.bouncycastle.org/].

Messaging Products

[Antepo] *Antepo OPN server* [http://www.antepo.com].

[Exodus] *Exodus* [http://exodus.jabberstudio.org].

[ICQ] *ICQ* [http://www.icq.com].

[AIM] *AOL Instant Messenger* [http://www.aim.com].

[MSN] *Microsoft Messenger* [http://messenger.msn.com].

[Yahoo] *Yahoo! Instant Messenger* [http://messenger.yahoo.com].

[IBM] IBM Redpaper: *IBM Lotus Instant Messaging and Web Conferencing 3.1* [http://www.sun.com/solutions/third-party/global/lotus/RedPpaperLotusInstantMessaging-wc.pdf].

Index