AN ABSTRACT OF THE THESIS OF

Saleh Almanei for the degree of Master of Science in

Electrical & Computer Engineering presented on June 3, 2003. Title:

Secure Instant Messaging The Jabber Protocol.

Redacted for privacy

Abstract approved: ______________________________

Çetin K. Koç

Instant Messaging (IM) has grown rapidly among network users. It has even become a very important tool for the industry around the world. It is used in scheduling meetings, exchanging business information and clients information, and so on. Instant Messaging has been developed by private sectors or providers such as America Online Instant Messenger (AIM), MSN, and Yahoo; however, in 1998 a new protocol has seen the light as an open source Instant Messaging protocol and had the name of Jabber and thanks to Jeremie Miller the founder of the Jabber protocol.

The project gathered wide public attention when it was discussed on the popular developer discussion website Slashdot in January 1999. In May 2000, the core Jabber protocols were released as open source reference server and it have not been changed to this day. Jabber uses client-server architecture, not a direct peer-to-peer architecture as some other messaging systems do. It is actually an Extensible Markup Language (XML) messaging protocol. It relies on XML document format in every aspect of the communication.[1]

Jabber Protocol have gone a long way to be one of the most attractive protocol because of its open source and extensibility. Anyone can build or extend the jabber protocol functionality without actually modifying the core protocol and still maintain interoperability with other IM clients such as yahoo and MSN. Moreover, as the usage of jabber Instant Messaging technology increases, the need for information protection in the

Jabber messaging medium also increases. This thesis will explore the Jabber protocol and the ability to secure a Jabber based communication over the network using third party cryptographic libraries.

Secure Instant Messaging
The Jabber Protocol

by

Saleh Almanei

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 3, 2003
Commencement June 2003

Master of Science thesis of Saleh A. Al-Manei presented on June 3, 2003

APPROVED:

Redacted for privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for privacy

Director of the School of Electrical Engineering and Computer Science

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Saleh Almanei, Author

ACKNOWLEDGMENT

I would like to thank Dr. Koç for giving me the opportunity to work on this interesting subject. Dr. Koç's discussions, directions, reviews and valuable comments helped direct me to the right path to accomplish this work.

After all, I want to give my biggest thanks to my father Abdullah Almanei and my mother Aljoharah AlFantookh for their support and patience during my study overseas.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

SECURE INSTANT MESSAGING
THE JABBER PROTOCOL

1. INTRODUCTION.

1.1. What is Jabber?

Jabber is an open XML protocol for the real-time exchange of messages and presence between any two points or users in a Jabber network. There are various usage for the Jabber technology; however, the first application of Jabber technology is an asynchronous, extensible instant messaging platform, and an IM network that offers functionality similar to legacy IM systems such as AOL Instant Messaging (AIM) and Yahoo Instant Messaging.[1]

1.2. History of Jabber

As Part of the Jabber IM standards efforts, in June 2000 the Jabber community submitted the protocols as a Request For Comments (RFC) to the Internet Engineering Task Force (IETF) as part of its Instant Messaging and Presence Protocol (IMPP) standard. Unfortunately, the IMPP effort didn't succeed. In May 2001 the Jabber Community (www.jabber.org) and Jabber Inc. (www.jabber.com) created the Jabber Software Foundation (foundation.jabber.org) to provide direct organizational assistance and indirect technical assistance to the Jabber community in carrying out its mission. In Fall 2002, the Internet Engineering Steering Group (IESG) approved the formation of an Extensible Messaging and Presence Protocol (XMPP) Working Group within the Internet Engineering Task Force (IETF). The scope of the Working Group is to explore and, where necessary, modify the existing protocol in order to meet the requirements defined in RFC 2779 as well as the requirements defined in the Common Presence and Instant

Messaging (CPIM) specification. The main focus of the Working Group will be on XML streams (including stream-level security and authentication), the core data elements, and the namespaces required to achieve basic instant messaging and presence.[2]

The XMPP working group published their XMPP Core Internet-Draft as the document which describes the core features of the Extensible Messaging and Presence Protocol. This document contains most of the work on Jabber client/server and server/server security and it will be used as a reference in this thesis. As jabber moves forward, much work remains to be done. Further standardization work is still underway and new standards are being proposed to extend the Jabber protocols beyond simple IM.[3]

1.3. Technology Overview

Jabber is based on the well-known client/server architecture. Jabber clients communicate with Jabber servers in their Jabber domain. Jabber domains have the advantage of separating the communication zone, which will be handled by different Jabber servers. Unlike most IM systems which use one centralized server for the whole communication zone.[3] The figure below shows the Jabber Client/Server streams flow:

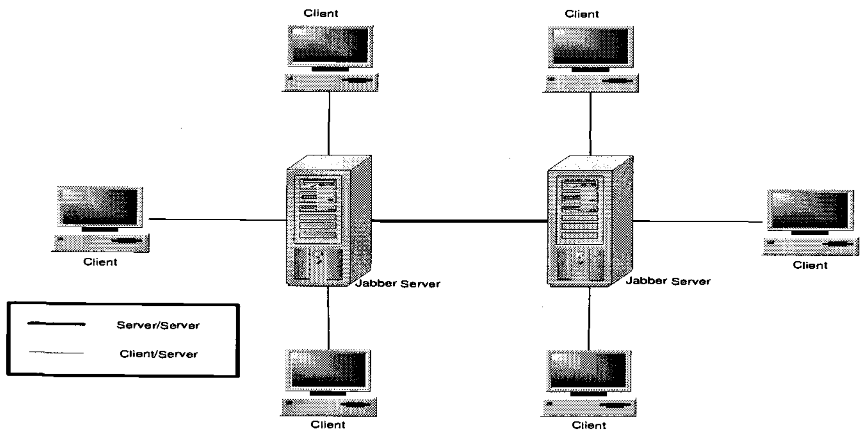


FIGURE 1.1: The Jabber client/server streams.

1.4. Architecture Background

1.4.1. Client/Server Model

Jabber uses client-server architecture, not a direct peer-to-peer architecture as some other messaging systems do. As a result, all Jabber data sent from one client to another must pass through at least one Jabber server. A Jabber client connects to a Jabber server on a TCP socket over port 5222. This connection is "always-on" for the life of the client's session on the server, which means the client does not have to poll for messages as an email client does. Rather, any message intended for delivery to the client is immediately pushed out to the client messenger as long as the client is connected. The server keeps track of whether the client is online or not, and when the client go off-line it stores any messages sent to the client for delivery when he or she connect again.

Modular Server: The Jabber server plays three primary roles:

- Handling client connections and communicating directly with Jabber clients.
- Communicating with other Jabber servers.
- Coordinating the various server components associated with the server.

Jabber servers are designed to be modular, with specific internal code packages that handle functionality such as registration, authentication, presence, contact lists, offline message storage, and so on. In addition, Jabber servers can be extended with external components, which enable server administrators to supplement the core server with additional services such as gateways to other messaging systems.

SIMPLE CLIENTS: One of the design criteria for the Jabber system was that it must be capable of supporting simple clients (e.g., even something as simple as a telnet connection on the correct port). Indeed, the Jabber architecture imposes very few restrictions on clients. The tasks a Jabber client must recognize and accomplish are:

- Communicate with the Jabber server over TCP sockets.
- Parse and interpret well-formed XML "chunks" over an XML stream.
- Understand the core Jabber data types (message, presence, and iq).

The advantage in Jabber is to move complexity from clients to the server. In practice, many of the low-level functions of the client (e.g., parsing XML and understanding the core Jabber data types) are handled by Jabber client libraries, enabling client developers to focus on the user interface.[3]

1.4.2. XML DATA FORMAT

XML is an integral part of the Jabber architecture because it is of utmost importance that the architecture be fundamentally extensible and able to express almost any structured data. The figure below shows the Jabber messaging model which composed of four main elements, XML packets containing markedup-data, XML streams used to transport XML packets, and Jabber clients and servers that exchange XML streams.

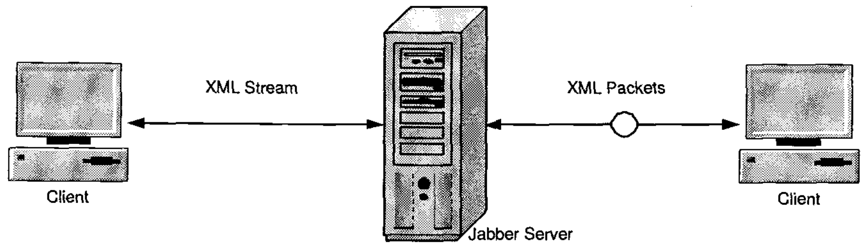


FIGURE 1.2: The Jabber messaging model.

When a client connects to a server, it opens a one-way XML stream from the client to the server, and the server responds with a one-way XML stream from the server to the client. Thus each session involves two XML streams. All communications between the client and the server happens over these streams. Example:

```
<message from='jolie@jabber.com/home' to='aima@rhymbox.com/work'>
   <body>Hello, I need to ask you a question!?</body>
</message>
```

While many Jabber snippets are just that simple, Jabber's XML format can also be extended through an official XML namespaces (managed by the Jabber Software Foundation) and custom namespaces for specialized applications. This makes Jabber a powerful platform to choose for any structured data, including XML Remote Procedure Calls (XML-RPC), Resource description framework Site Summary (RSS), and Scalable Vector Graphics (SVG).[1]

1.4.3. DISTRIBUTED NETWORK

Each user connects to a "home" server, which receives information for them, and the servers transfer data among themselves on behalf of users. Thus any domain can run a Jabber server. Each server functions independently of the others, and maintains its own user list. In addition, any Jabber server can talk to any other Jabber server that is accessible via the Internet (if server-to-server communications are enabled). A particular user is associated with a specific server, and Jabber addresses are of the same form as email addresses. The result is a flexible, controllable network of servers, which can scale much higher than the monolithic, centralized services run by legacy IM vendors.[1]

1.4.4. STANDARDS-BASED ADDRESSING

Within Jabber there are many different entities that need to communicate with each other. These entities can represent transports, groupchat rooms, or a single Jabber user. Jabber IDs are used both externally and internally to express ownership or routing information. Each Jabber ID (or "JID") contains a set of ordered elements. The JIDs are formed of a domain, node, and resource in the following format: [node@]domain[/resource][4]

This is an example: almanei@jabber.com/home.

1.5. The Jabber Protocols

There are three core protocols that Jabber relies on when we deal with messaging mechanism:

1.5.1. Message

The message protocol is in fact the simplest protocol in Jabber. Most of the traffic in any jabber network falls under the message protocol. It consist of four attributes and zero or more child elements. The attributes are:

to: Specifies the intended recipient of the message.

from: Specifies the sender of the message.

id: An optional unique identifier for the purpose of tracking messages.

type: An optional specification of the conversational context of the message.

The child elements are :

body: The textual contents of the message; normally included but not required.

subject: The subject of the message.

thread: A random string that is generated by the sender, used for tracking a conversation thread.

error: Description of the error.

That is all what the message protocol in the current version of the XMPP core protocol standard represent.[5] An example of a message packet is shown below:

```
<message to='romeo@montague.net'
from="juliet@capulet.com/balcony'
type='chat'>
  <subject xml:lang='en'>
    Greeting!
  </subject>
  <body xml:lang='en'>
    Hello!!
  </body>
  <thread>e0f92794b9683a38</thread>
</message>
```

1.5.2. Presence

This protocol is responsible for subscription, approval, and update of presence information in the Jabber community. The attribute associated with this protocol is the same for the message protocol except that the type attribute have 7 states as follows:

1.unavailable: The client is no longer available for communication.

2.subscribe: The sender request to subscribe to the recipient's presence.

3.subscribed: The sender has allowed the recipient to receive their presence.

4.unsubscribe: A notification that a client is unsubscribing from another client's presence.

5.unsubscribed: The subscription request has been denied or a previously-granted subscription has been cancelled.

6.probe: A request for a client current presence.

7.error: An error has occurred regarding processing or delivery of a previously-sent presence packet.

Also the presence packet may contain zero or one of each of the following child elements:

show Describes the availability status of an entity or specific resource. It have four values one of them is used:

1. away : Temporarily away.
2. chat : Free to chat.
3. xa : Extended away.
4. dnd : Do ont Disturb.

status An optional natural-language description of availability status.

priority A non-negative integer representing the priority level of the connected resource, with zero as the lowest priority.

error Description of the error.

An example of a presence packet is shown below:

```
<presence
   from='juliet@capulet.com/balcony'
   to='romeo@montague.net/orchard'>
  <show>away</show>
  <status>be right back</status>
  <priority>0</priority>
</presence>
```

Finally, the presence packet can contain any properly-namespaced child element that doesn't interfere with the available structured namespaces and tags.[5]

1.5.3. Info/Query

The IQ protocol is the last core Jabber protocol and take care of anything other than the message and presence protocols. IQ is a generic request-response protocol that is designed to be easily extensible. Just as HTTP is a request-response medium. The data content of the request and response is defined by the namespace declaration of a direct child element of the IQ element. The attributes associated with this protocol have the same attribute as the message and presence protocols except that it differ in the type attribute. The type attribute have four values that can be used:

1. **get:** Request information.

2. **set:** Provide required data.

3. **result:** Response to a successful get or set request.

4. **error:** Error occurred in processing or delivery of a get or set requests.[5]

The following is an example of an IQ packet that blocks an incoming message from a specific JID:

```
<iq type='set' id='msg1'>
<query xmlns='jabber:iq:privacy'>
<list name='message-jid-example'>
  <item type='jid' value='tybalt@capulet.com'
        action='deny' order='3'>
    <message/>
  </item>
</list>
</query>
</iq>
```

This IQ protocol is very important if the goal is to build server based security policy that the clients should follow. The next step after completing the work for clients security should concentrate on the server based security support.

1.6. Work path and organization

As mentioned earlier, covering the Jabber protocol in full is outside the thesis scope. The goal is to provide enough information in understanding the security available from the client side only. The modification of a jabber server code is not a necessity to accomplish the task of securing a Jabber client messaging mechanism.

The remainder of this thesis work is organized in 5 chapters. Chapter 2 presents the available security options for Jabber as of writing this thesis. Chapter 3 presents an overview of the mechanism used to secure the jabber conversation in this thesis and the tools required. Chapter 4 describes a working secure model for Jabber. The experimental results of the developed design are presented and compared with the available secure Jabber protocols in Chapter 5. Chapter 6 concludes this work.

2. AVAILABLE SECURITY OPTIONS

Current security features in Jabber to protect message data are generally inadequate for many deployments; this is particularly true in security conscious environments like large, commercial enterprises and government agencies. These current features suffer from issues of scalability, usability, and supported features. Furthermore, there is a lack of standardization.[3] The table below shows the Jabber present security features compared to computer security properties.

Security Property	*Description*	*Existing Technology*	*Jabber Support*
Authentication	Ensures an entity is who it claims to be.	JAASI, Kerebos, Microsoft Passport	Jabber Authentication Protocol.
Authorization	Determines what an entity has permissions for accessing(data) or controlling(resources).	JAAS, access control lists	Binary authorization. Unauthenticated users are granted certain rights, and authenticated users others.
Integrity	Ensures data has not been tampered with.	Message digests.	None
None-repudiation	Ensures that the author of data can always be identified.	Digital signatures.	None
Confidentiality	Ensures data can be read only by authorized entities.	Encryption	Limited client/server confidentiality using SSL.

TABLE 2.1: Computer security properties compared with Jabber's present capabilities.

The following will present the security options in details that are available as of writing this document in the Jabber community; however, most of the information presented are proposals or have the title of "work on progress" since there is no final standard that governs security for the Jabber protocol.

2.1. Stream Encryption

XMPP includes a method for securing the stream from tampering and eavesdropping. This channel encryption method makes use of the Transport Layer Security (TLS) protocol, along with a "STARTTLS" extension. The namespace identifier for the STARTTLS XML extension is 'http://www.ietf.org/rfc/rfc2595.txt'. TLS may be used between any initiating entity and any receiving entity (e.g., a stream from a client to a server or from one server to another).[5]

2.1.1. SSL/TLS

To use SSL/TLS, the client start by issuing STARTTLS and the server responds of whether it supports TLS or not. The following example shows the data flow for a client securing a stream using STARTTLS:

- Step 1: Client initiates stream to server:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
  to='capulet.com'
  version='1.0'>
```

- Step 2: Server responds by sending a stream tag to the client:

```
<stream:stream
  xmlns='jabber:client'
```

```
xmlns:stream='http://etherx.jabber.org/streams'
id='12345678'
version='1.0'>
```

- Step 3: Server sends the STARTTLS extension to the client along with authentication mechanisms and any other stream features:

```
<stream:features>
  <starttls xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
   <mechanisms xmlns='http://www.iana.org/assignments/
       sasl-mechanisms'>
   <mechanism>DIGEST-MD5</mechanism>
   <mechanism>PLAIN</mechanism>
   </mechanisms>
</stream:features>
```

- Step 4: Client sends the STARTTLS command to the server:

```
<starttls xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
```

- Step 5: Server informs client to proceed:

```
<proceed xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
```

- Step 5 (alt): Server informs client that TLS negotiation has failed and client continues with stream authentication.

```
<failure xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
```

- Step 6: Client and server complete TLS negotiation via TCP.
- Step 7: Client initiates a new stream to the server:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
  to='capulet.com'
  version='1.0'>
```

- Step 8: Server responds by sending a stream header to the client along with any remaining negotiatiable stream features:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
  id='12345678'
  version='1.0'>
<stream:features>
  <mechanisms xmlns='http://www.iana.org/assignments/
    sasl-mechanisms'>
   <mechanism>DIGEST-MD5</mechanism>
   <mechanism>PLAIN</mechanism>
   <mechanism>EXTERNAL</mechanism>
  </mechanisms>
</stream:features>
```

- Step 9: Client continues with stream authentication.[5]

The above example illustrates a client to server negotiation; however, server administrators may choose to use TLS between two domains for the purpose of securing server-to-server communications. The process and flow of information will not differ from the above example.

2.1.2. OpenPGP

XMPP working group outline the use of OpenPGP solution as it is used today with no actual modification in their Internet-Draft titled End-To-End Object Encryption. They describe object encryption as the key exchange mechanisms were done by using the OpenPGP key servers. The following is a demonstration of OpenPGP usage in a Jabber system.

The encrypted payload contains what would be the main $< body >$ element if the message were not encrypted, along with a message ID to help prevent replay attacks; both pieces of information are wrapped in a $< payload\backslash>$ element scoped by the $'http : //jabber.org/protocol/e2e\#payload'$ namespace, as shown in the following example:

```
<payload xmlns='http://jabber.org/protocol/e2e#payload'>
 <id>someID</id>
  <body>Wherefore art thou?</body>
</payload>
```

The encrypted payload includes an $< id >$ element. The CDATA of the $< id >$ element will be constructed according to the following algorithm:

1. concatenate the sender's full JID (user@host/resource) with the recipient's full JID;
2. concatenate these JID strings with a full ISO-8601 timestamp including year, month, day, hours, minutes, seconds, and UTC offset if appropriate in the following format: yyyy-mm-dd Thh:mm:ss-hh:mm;

3. hash the resulting string according to the SHA1 algorithm;

4. convert the hexidecimal SHA1 output to all lowercase.

The full $< payload/ >$ element (including all XML tag names and angle brackets) will be encrypted according to the OpenPGP algorithm using the sender's KeyID. The armored output in US-ASCII format and have the headers removed. The resulting cipher text will be provided as the CDATA of an $< x/ >$ element scoped by the $'http://jabber.org/protocol/e2e'$ namespace. Finally, the message optionally can contain an unencrypted $< body/ >$ child element whose CDATA informs the recipient that the actual message body is encrypted.

The format of the full message that results from the foregoing procedure is shown in the following example.

```
<message from='juliet@capulet.com/balcony'
  to='romeo@montague.net/orchard'>
  <body>Encrypted is this message.</body>
  <x xmlns='http://jabber.org/protocol/e2e'>
  hQEOA+fczQLixGb6EAP/UvOJNo1x/h9d6ia75foKB1sViwAeXnr
  AwUDuxFhTBdt3HDOeF61b/sqaHBi4B4L5Oxn4W+dZdOsxgf4QNo
  WucI6WfqcV5BT3K62iTGLVJ7LcRoXTylekNsDiNsMVMJBHoYqeo
  RmTuMt3uuljBHHnXVya7XGMmyxbM/QtdxuykssD/jsvER1EyIfY
  SWT+G/djvymd9FfgTwLrgyBjC1SOGfQ6oEjmEz5FK+BpwfRDzxj
  DeRO8Q6m7Y8C84OC4Dq4UCSCcdzhhKHHOpACizjeG/2N+DlEwDk
  wK3b/2ED8fFPE1tCUIl6Z8uvAw5Q6OBeFabgbjdi3QjqY32fV5t
  OtUkkvkOsAEAcRBF9HqEHNDMEb/bGza03mV58dlEOjhZEu2rCff
  R4mqYSDoF8hNb/XuOssDuIvp342ILfAPjyx/AE1/ffdNOtSWt3k
  EZzDzeJfFOBzv2n8OPNUKrRAoinnRr9vdFH5KlIQbTFteOFk/r7
  YA7PghNWtPZJ/mXQPoCylaK86wGc/KHld8Y+RopWeZSoicpIqGB
  rpuwdl/o/OtEmObVnDh3dJpz89aJj2RAAiTaKLotLg/AkmwfQGL
  </x>
```

```
</message>
```

The decoded payload is:

```
<payload xmlns='http://jabber.org/protocol/e2e#payload'>
  <id>e0ffe42b28561960c6b12b944a092794b9683a38</id>
  <body>O Romeo, Romeo! Wherefore art thou Romeo?</body>
</payload>
```

In order to signal support for this method of encrypting message bodies, an entity must broadcast its KeyID in all outgoing presence packets, contained in an $< x/ >$ element scoped by the $'http://jabber.org/protocol/e2e'$ namespace.

```
<presence
  from='juliet@capulet.com/balcony'
  to='romeo@montague.net/orchard'>
   <show>away</show>
   <status>be right back</status>
   <x xmlns='http://jabber.org/protocol/e2e'>88CA1D46</x>
</presence>
```

[6]

The above example assumes that the clients already have created their keys and already can retrieve them from available OpenPGP servers in the network. As of writing this thesis, there are number of clients that support the usage of OpenPGP encryption algorithm based on the illustration above.

2.2. Stream Authentication

XMPP includes two methods for enforcing authentication at the level of XML streams. When one entity is already known to another (i.e., there is an existing trust relationship between the entities such as that established when a user registers with a server or an administrator configures a server to trust another server), the preferred method for authenticating streams between the two entities uses an XMPP adaptation of the Simple Authentication and Security Layer (SASL). When there is no existing trust relationship between two servers, some level of trust may be established based on existing trust in DNS; the authentication method used in this case is the server dialback protocol that is native to XMPP (no such ad-hoc method is defined between a client and a server). If SASL is used for server-to-server authentication, the servers must not use dialback.[5] Both SASL and dialback authentication methods are described in this section.

2.2.1. SASL Authentication

The Simple Authentication and Security Layer (SASL) provides a generalized method for adding authentication support to connection-based protocols. XMPP uses a generic XML namespace profile for SASL and the namespace identifier for this protocol is $'http://www.iana.org/assignments/sasl-mechanisms'$.

The following example shows the data flow for a client authenticating with a server using SASL.

1. Client initiates stream to server:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
  to='domain'
  version='1.0'>
```

2. Server responds with a stream tag sent to the client:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
  id='12345678'
  from='domain'
  version='1.0'>
```

3. Server informs client of available authentication mechanisms:

```
<stream:features>
 <mechanisms xmlns='http://www.iana.org/assignments/
   sasl-mechanisms'>
 <mechanism>DIGEST-MD5</mechanism>
 <mechanism>PLAIN</mechanism>
 </mechanisms>
</stream:features>
```

4. Client selects an authentication mechanism ("initial response"):

```
<auth xmlns='http://www.iana.org/assignments/sasl-mechanisms'
mechanism='DIGEST-MD5'/>
```

5. Server sends a base64-encoded challenge to the client:

```
<challenge
xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
cmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIi
xxb3A9ImF1dGgiLGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGhtPW1kNS1zZXNz
</challenge>
```

The decoded challenge is:

```
realm="cataclysm.cx",nonce="OA6MG9tEQGm2hh",
\qop="auth",charset=utf8,algorithm=md5-sess
```

6. Client responds to the challenge:

```
<response xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
  dXNlcm5hbWU9InJvYiIscmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik9BNk1H
  OXRFUUdtMmhoIixcIGNub25jZT0iT0E2TUhYaDZWcVRyUmsiLG5jPTAwMDAwMDAx
  LHFvcD1hdXRoLFwgZGlnZXN0LXVyaT0ieG1wcC9jYXRhY2x5c20uY3giLFwgcmVz
  cG9uc2U9ZDM4OGRhZDkwZDRiYmQ3NjBhMTUyMzIxZjIxNDNhZjcsY2hhcnNldD11
  dGYtOA==
</response>
```

The decoded response is:

```
username="rob",realm="cataclysm.cx",nonce="OA6MG9tEQGm2hh",
\cnonce="OA6MHXh6VqTrRk",nc=00000001,qop=auth,\
digest-uri="xmpp/cataclysm.cx",
\response=d388dad90d4bbd760a152321f214
3af7, charset=utf-8
```

7. Server sends another challenge to the client:

```
<challenge
 xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
 cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3YjgOZGJhYmNkZmZmZA==
</challenge>
```

The decoded challenge is:

```
rspauth=ea40f60335c427b5527b84dbabcdfffd
```

8. Client responds to the challenge:

```
<response xmlns='http://www.iana.org/assignments/sasl-mechanisms'/>
```

9. Server informs client of successful or failed authentication:

```
<success xmlns='http://www.iana.org/assignments/sasl-mechanisms'/>
Or:
<failure xmlns='http://www.iana.org/assignments/sasl-mechanisms'
 code='454'/>
```

10. Client initiates a new stream to the server:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
  to='domain'
  version='1.0'>
```

11. Server responds by sending a stream header to the client, with the stream already authenticated:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
```

```
id='12345678'
from='domain'
version='1.0'>
```

The SASL authentication should provide added security to the Jabber system.[5] However, as of writing this thesis there is no Jabber client that have SASL support.

2.2.2. Dialback Authentication

XMPP includes a protocol-level method for verifying that a connection between two servers can be trusted (at least as much as the DNS can be trusted). The method is called dialback and is used only within XML streams that are declared under the "jabber:server" namespace. The purpose of the dialback protocol is to make server spoofing more difficult, and thus to make it more difficult to forge XML stanzas. Dialback is not intended as a mechanism for securing or encrypting the streams between servers, only for helping to prevent the spoofing of a server and the sending of false data from it. However, domains requiring more robust security should not consider this option of authentication.[5]

2.3. Other work On the Jabber Community

An incomplete draft of a Jabber security protocol has been published at Jabber software foundation for securing Jabber conversation. The proposed protocol has the following requirements:

- It must be an optional extension of the existing Jabber protocol.
- It must be transparent to existing Jabber servers.
- It must function gracefully in cases where some community members are not running a user agent that supports the protocol.

- It must make good use of XML.
- It must avoid encumbered algorithms.
- It must be straightforward to implement using widely available cryptographic toolkits.
- It must not require a PKI.

The proposed draft have four main security sections discussed below:

2.3.1. Security Sessions

Security sessions are identified by a 3-tuple consisting of the following items:

1. Initiator. This is the JID of the user who initiated the session.
2. Responder. This is the JID of the user who responded to the initiator's request.
3. SessionId. A label generated by the initiator.

Security sessions are used to transport conversation keys between the conversation participants. They are negotiated using an authenticated Diffie-Hellman key agreement exchange. The two goals of the exchange are to perform the mutual authentication and to agree to a secret that is known only to each.

Once the pair agrees on a shared secret, they each derive key material from the secret; this key material is used to securely transport the conversation keys, which are used to actually protect conversation data. The protocol data units (PDUs) that comprise the exchange are transported within existing Jabber protocol elements. There are three PDU sessions to complete this task, shown in figure 2.1. Notice that the attribute values are the version, initiator JID, responder JID, sessionId, and hmac value.

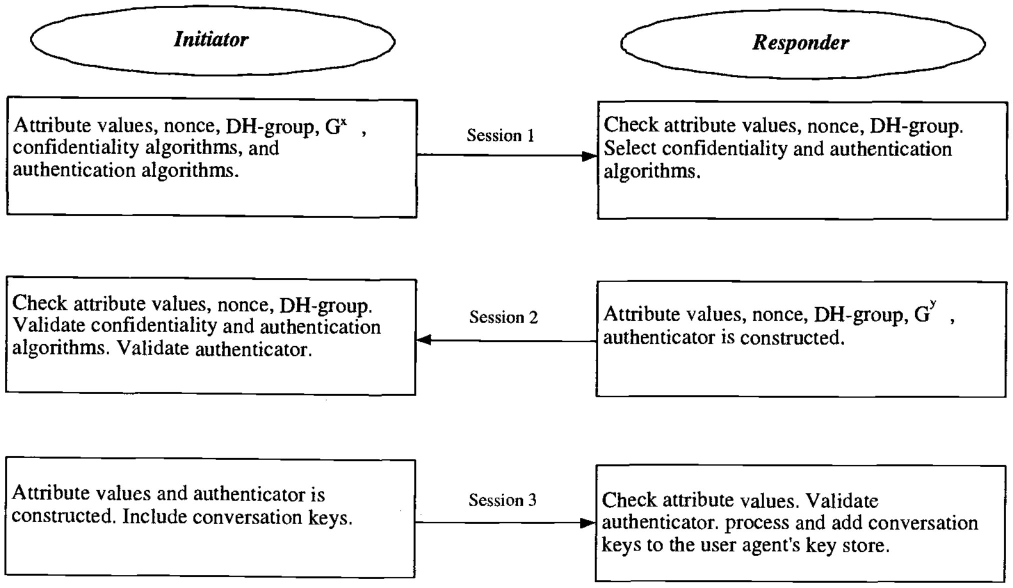

Initiator
Responder
Attribute values, nonce, DH-group, G^x , confidentiality algorithms, and authentication algorithms.
Session 1
Check attribute values, nonce, DH-group. Select confidentiality and authentication algorithms.
Check attribute values, nonce, DH-group. Validate confidentiality and authentication algorithms. Validate authenticator.
Session 2
Attribute values, nonce, DH-group, G^y , authenticator is constructed.
Attribute values and authenticator is constructed. Include conversation keys.
Session 3
Check attribute values. Validate authenticator. process and add conversation keys to the user agent's key store.

FIGURE 2.1: A security session's XML packets generation and receiving between the Initiator and the Responder.

2.3.2. Key Transport Mechanism

Conversation keys are transported using the symmetric key wrap feature of XML Encryption embedded in the keyTransport PDU.

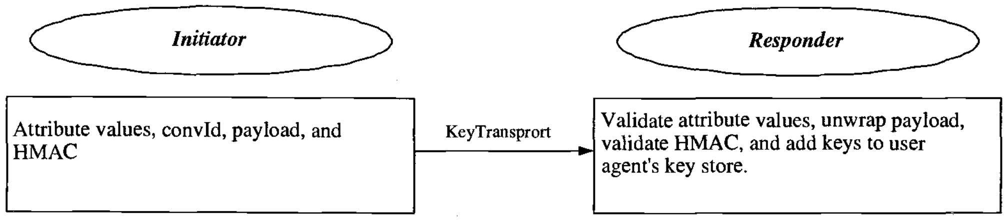
Example:

```
<!ELEMENT keyTransport
                (convId, payload, hmac) >
<!ATTLIST keyTransport
                version CDATA #REQUIRED
                initiator CDATA #REQUIRED
                responder CDATA #REQUIRED
                sessionId CDATA #REQUIRED >

<!ELEMENT convId (#PCDATA)* >
```

```
<!-- These are actually instances of xenc:EncryptedKey -->
<!ELEMENT payload
                (confKey, hmacKey) >


<!ELEMENT hmac (#PCDATA)* > <!ATTLIST hmac
                encoding (base64 | hex) #REQUIRED >
```



Initiator
Responder
Attribute values, convId, payload, and HMAC
KeyTransprort
Validate attribute values, unwrap payload, validate HMAC, and add keys to user agent's key store.

FIGURE 2.2: XML packet for KeyTransport. The attribute values are the version, initiator JID, responder JID, sessionId

2.3.3. Message Protection

A protected message is defined as a traditional Jabber message whose body content is extended to include the transport of a cryptographically protected message body. The two key features are

- The usual body element contains some arbitrary text.
- The message contains Jabber $< x >$ element defining the Jabber:security:message namespace; this element transports the protected message.

This mechanism has the advantages of allowing transparent integration with existing Jabber servers and existing Jabber clients.[7]

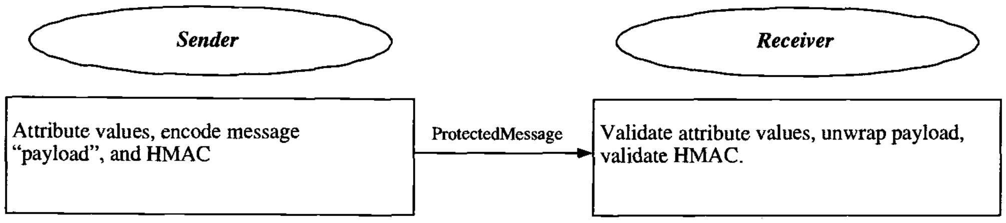

Sender
Receiver
Attribute values, encode message "payload", and HMAC
ProtectedMessage
Validate attribute values, unwrap payload, validate HMAC.

FIGURE 2.3: XML packet for Mesaage Protection. The attribute values are version, from, to, convId, and seqNum.

2.3.4. Requesting Keys

This is the last main section in the proposed draft. However, its titles To Be Added (TBA)! Hence, there are no more information added as of writing this thesis.

2.4. XML Security

XML is one of the most used language when it comes to designing structured documents.[8] Hence,One of the advantages that Jabber standard provide is the ability to add custom security system without altering the Jabber standards, protocols, or the Jabber server. Jabber provides the ¡x¿ extension protocol to allow the extension of the core Jabber protocol. As an example consider the ¡message¿ packet example mentioned previously which clearly shows the usage of the ¡x¿ extension. Hence, the XML encryption standard proposed by the World Wide Web Consortium (W3C) can be used in Jabber without actually modifying or altering the Jabber standards.[?]

As of writing this thesis, various components of XML security are at various stages in the standardization process. XML Digital Signature (XMLDSIG) and XML Encryption (XMLENC) which provides authentication and confidentiality respectively are among the XML cryptographic suit that is underdevelopment.[9]

3. SECURING JABBER MESSAGES TOOLS AND REQUIREMENTS.

This chapter presents the environment setup, tools required, and goals need to be achieved to secure a Jabber message and how to complete the task.

3.1. Proof of Concept

The goal for this thesis is to present a working Jabber client that uses Diffie-Hellman key exchange and DES algorithms for key exchange and encryption/decrption respectively. Since Jabber is an open source project, the search for such a client would be easy on the internet, however as of writing this thesis, I didn't come across a single messenger that deploy both algorithms. Hence, there will be a need to build one or use an already available client and modify the code to add the key exchange and encryption support.

3.2. A secure Messenger!

The requirements to have a working secure Jabber client is limited to the following:

- Basic knowledge of XML.
- Access to a cryptographic library.
- An already working, open source, Jabber Client.

3.2.1. Required Tools

I used Java as the main programming language in this work. The choice has been made since Java is a machine independent language as along as there exist a Java Virtual Machine that can run the code.[10]

The cryptographic library used here is the Java Cryptographic Extension (JCE) from Sun Microsystems. Its already included in their release of J2SE 1.4.2 the beta version.

The usage of the JCE library will be on the JavaJabberC (JJC), a client available for download from the sourceforge.net web site. This client is very simple in its design and looks. A more closer look at this client graphical interface will follow in the next chapter.

3.2.2. XML Modification

Its not necessary to investigate XML document structure any further. The structure of a message XML tag is enough to carry out our task in modifying the Jabber client. However, the following paragraphs will provide the information needed about the XML tags implemented in this work.

All the work in this project deals with the $< message >$ tag since the goal is to secure the message it self. As we show earlier how does a message XML packet looks like in chapter 1, a modification is required to this packet to serve our purpose. Please notice that there is no use of XML namespaces in this demonstration. The reason behind omitting XML namespaces is to make it more easier and direct. In fact adding the namespaces support will not require much work and it should be used to meet the Jabber protocol standards to its fullest. Although there are no reason to reject this design because of the fact that the $< body >$ tag is actually optional in the $< message >$ protocol. Hence, omitting the $< body >$ tag and replacing it with another that serve different purpose but at the same time does not violate the $< message >$ protocol will have no side effect on the protocol design. The rest of this section will shows the added XML tags to support the key exchange and the encryption of the data.

The design of the message packet has three extra tags:

1.$< keyexchange >$**:** This tag is used to initialize the key exchange mechanism with the receiver. It has this format:

```
<message from='node@host' to='receiving-ID'>
        <keyexchange>
        MIIDKTCCApKgAwIBAgIBDDANBgkqhkiG9w0
        BAQQFADCBOTELMAkGA1UEBhMCWkExFTATBg
        NVBAgTDFdlc3Rlcm4gQ2FwZTESMBAGA1UEB
        xMJQ2FwZSBUb3duMRowGAYDVQQKExFUaGF3
        </keyexchange>
</message>
```

2.$<$ *keyexchangestep*2 $>$**:** This is the second step in the key agreement protocol which consist of receiving the other party public key, which is the result of $<$ *keyexchange* $>$ packet above. This tag has the formation as shown below:

```
<message from='node@host' to='receiving-ID'>
        <keyexchangestep2>
        MIIDKkjasdkhjhHHMmsjhYhsj767wJiG9w0
        oksoKKKKjfd8734j2YHSKAWWRffsdvTATBg
        NVBAgTDFdlc3RluMRowGAYDVESMBAGA1UEB
        Kls8HHdfdnppwJuYTRsdfdssQQKExFUaGF3
        </keyexchangestep2>
</message>
```

3.$<$ *cipherdata* $>$**:** This is where all the encrypted message will be hold. This tag is only used to hold the encrypted data and it will signal a decrypt method on the receiver side. it has the formation of:

```
<message from='node@host' to='receiving-ID'>
        <cipherdata>
        JJksjfkds763ajsdhfu4THrY
        </cipherdata>
</message>
```

The following diagram shows the four types of the message that can be used:

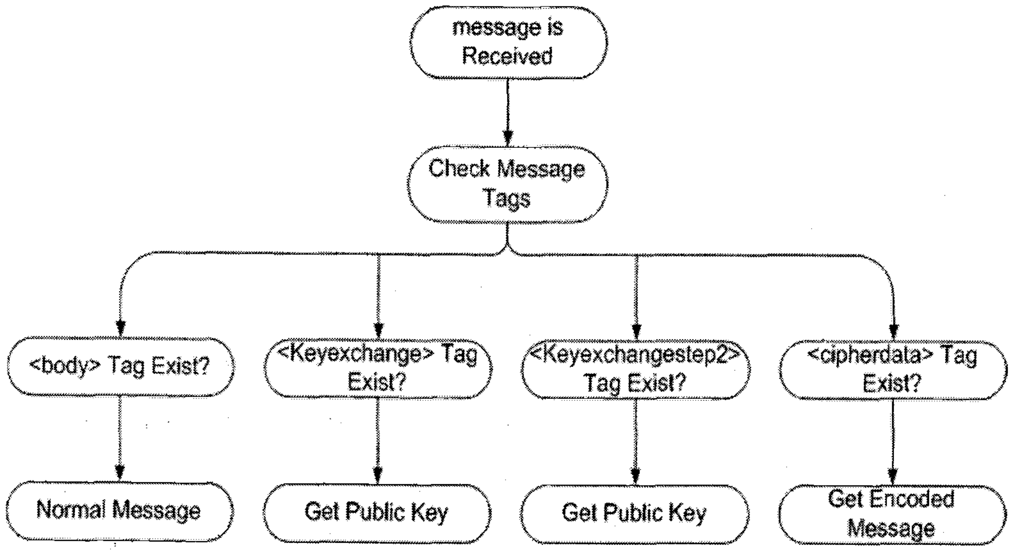


FIGURE 3.1: A flow diagram of the $< message >$ protocol.

3.2.3. Cryptographic Code

The cryptographic library used in this demonstration is the Java Cryptography Extension (JCE) provided by SUN Microsystems. JCE is a set of packages that provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms.[11]

The usage of this crypto library in this project is limited to the Diffie-Hellman key exchange algorithm and the DES algorithm although the opportunity to add more support for encryption algorithm is available.

The JCE library is bundled with the new release of the J2SE or J2EE and there was no need to modify the installation to support the JCE library.

The following figure shows the Diffie-Hellman key exchange algorithm that will take place between the two users namely Alice and Bob.

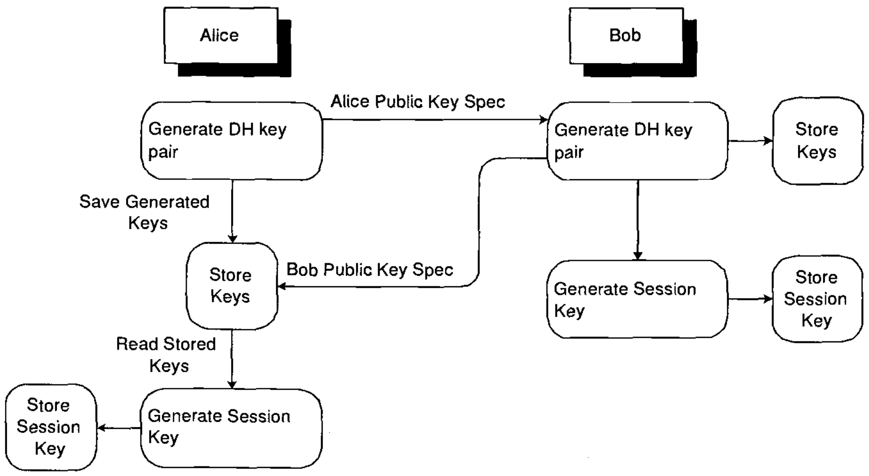


FIGURE 3.2: A flow diagram of the Diffie-Hellman key exchange algorithm.

The following will provide an explaination of the java code used to accomplish the key exchange and encryption tasks respectively.

3.2.3.1. Diffie-Hellman Key Exchange

There are three classes that deal with the key creation and exchange:

1.DHKeysCreation.class: This is the first step in generating and exchanging the Diffie-Hellman public key with other Jabber users.

A fragment of the the code written to generate a Diffie-Hellman Key pair for any of the Jabber users is shown below:

```
System.out.println("Creating Diffie-Hellman parameters...");
```

```
AlgorithmParameterGenerator paramGen
       = AlgorithmParameterGenerator.getInstance("DH");
paramGen.init(512);

AlgorithmParameters params = paramGen.generateParameters();
dhSkipParamSpec = (DHParameterSpec)params.getParameterSpec
                (DHParameterSpec.class);
KeyPair Kpair = KpairGen.generateKeyPair();
System.out.println("Initialization ...");

KeyAgreement KeyAgree = KeyAgreement.getInstance("DH");
KeyAgree.init(Kpair.getPrivate());
// Encodes the public key
byte[] PubKeyEnc = Kpair.getPublic().getEncoded();

byte[] PriKeyEnc = Kpair.getPrivate().getEncoded();
```

2.DHKeysCreationStep2.class: This class will be called when a public key is received from the sender who initiated the key exchange and it will extract the public key material and start generating the public key and session key. The procedure will not differ from the code instant above except in storing and reading the received public key, example:

```
{ ....
 String directoryPath =
        System.getProperty("user.home")+"//jabberkeys//"+to;
 FileInputStream publicKeyFileInputStream = new
     FileInputStream(directoryPath+"//"+to+"PublicKey.DH");
 byte[]enryptedPublicKey = new
      byte[publicKeyFileInputStream.available()];
```

```
  publicKeyFileInputStream.read(enryptedPublicKey);
  publicKeyFileInputStream.close();

  KeyFactory kf = KeyFactory.getInstance("DH");
  X509EncodedKeySpec x509KeySpec = new
                     X509EncodedKeySpec(enryptedPublicKey);

  PublicKey alicePubKey = kf.generatePublic(x509KeySpec);

 ...

  bobKeyAgree.doPhase(alicePubKey, true);
  SecretKey bobDesKey = bobKeyAgree.generateSecret("DES");
  ObjectOutputStream keyOut = new ObjectOutputStream(
            new FileOutputStream(directoryPath+"//MyDesKey.des"));
  keyOut.writeObject(bobDesKey);
  keyOut.close();

  return encodedData;

 }
```

3.DHKeyAgreementPhase1.class: This class will be executed when a reply is received from the receiver end-point which will hold the public key of the receiver and since the user already have his or her public key available, the process will be to generate the session key directly then the session key will be transformed to a DES key then stored and used when necessary.

```
{ ....
\\Generate the session key.
 aliceKeyAgree.doPhase(bobPubKey, true);
 byte[] aliceSharedSecret = aliceKeyAgree.generateSecret();

\\Use the session key for DES.
 aliceKeyAgree.doPhase(bobPubKey, true);
 SecretKey aliceDesKey = aliceKeyAgree.generateSecret("DES");

\\Store the DES key.
 ObjectOutputStream keyOut = new ObjectOutputStream(
 new FileOutputStream(directoryPath+"//MyDesKey.des"));
 keyOut.writeObject(aliceDesKey);
 keyOut.close();
....
}
```

3.2.3.2. DES Encryption/Decryption

There is only one Java class that deal with DES encryption and decryption:

1.DesEncryption.class: This is the class where all the encryption and decryption take place after the two users have agreed in a session key. The DES key stored earlier will be read by the Java class and then used to encrypt or decrypt according to the argument passed to it. The example below show a fragment of the encryption method:

```
{ ....
 String directoryPath =
 System.getProperty("user.home")+"//jabberkeys//"+to;
 ObjectInputStream keyIn = new ObjectInputStream(
    new FileInputStream(directoryPath+"//MyDesKey.des"));
    Key key = (Key) keyIn.readObject();
    keyIn.close();

    Cipher bobCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
    bobCipher.init(Cipher.ENCRYPT_MODE, key);

    byte[] cleartext = msg.getBytes();
    byte[] ciphertext = bobCipher.doFinal(cleartext);
    encodedData = encoder.encode(ciphertext);

    return encodedData;
   .... }
```

3.2.4. Open Source Jabber Client Code

The Jabber client source code is very long to be printed in papers; however, most of the modifications were made in the JabberMessageCenter.java and for illustration, the code below presents a check for the tag $< cipherdata >$ and if it does exist then start decrypting the message:

```
if (msg.cipherdata != null){
        try{
        String s = new String(JabberUtil.deXML(msg.cipherdata));
        final String nick = jr.getNickname(msg.from);
        DesEncryption cipher = new DesEncryption();
        String cipherMessage = new String(cipher.decrypt(s,nick));
        ........... }
}
```

Looking at the code above we can see that msg.cipherdata is the XML node having the name cipherdata and it will be sent to the deXML class which will get the value that this node hold then it will be stored in the (s) string and then transferred to the class DesEncryption to use the method called decrypt to decrypt the cipher message.

The integration of the crypto-code into the already available Java Jabber Client code was very straight forward as long as there is full understanding of how does the code implements the Jabber protocol.

Also modification to the graphical user interface were mandatory to present to the user the availability of such support for key exchange and encryption. A more clearer look at the graphical user interface will follow in the next chapter and a comparison will also follow in chapter 5.

4. SECURE JAVA JABBER CLIENT ILLUSTRATION

This chapter presents the working secure Jabber client in graphical details which explain how does the generation and exchanging keys take place and how does a secure communication is established between two users.

The Jabber client interface that I am using is not difficult to understand; however a step by step illustration is provided in this chapter. The first step is to start the Jabber client which have the interface shown in figure 4.1:

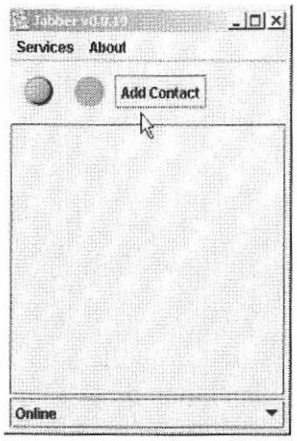
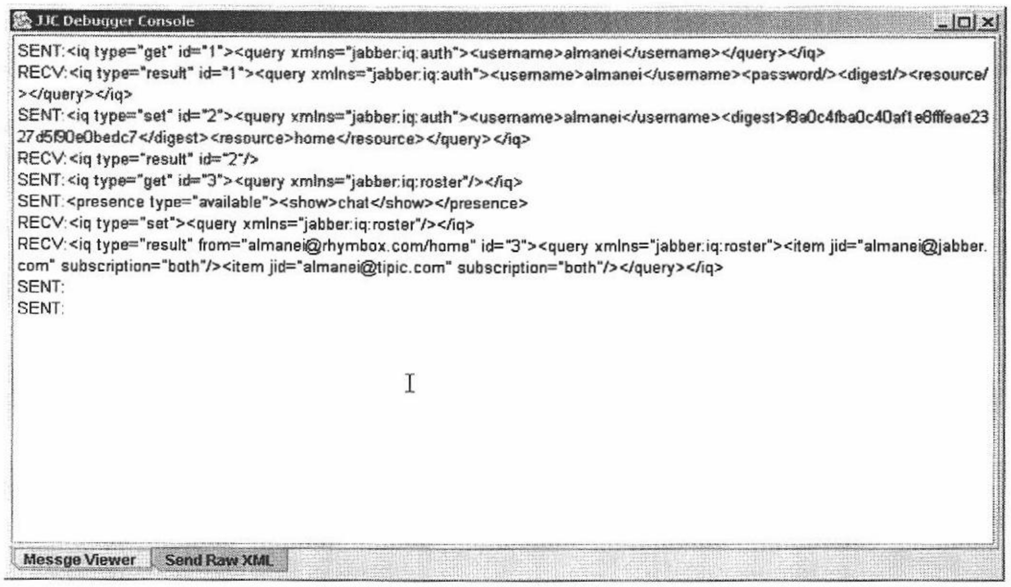


FIGURE 4.1: The Jabber Client.

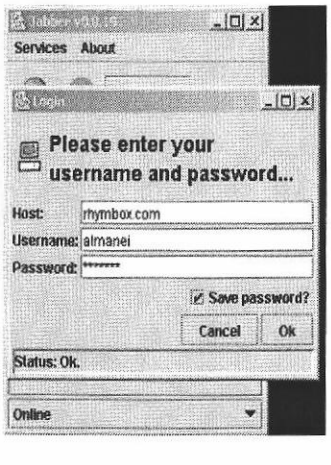
This client comes with a very neat plug-in which shows the actual messages sent and received in raw XML format. This feature is very helpful in showing what is actually going to the server! This is called the Debug Console as shown in figure 4.2:



```
SENT:<iq type="get" id="1"><query xmlns="jabber:iq:auth"><username>almanei</username></query></iq>
RECV:<iq type="result" id="1"><query xmlns="jabber:iq:auth"><username>almanei</username><password/><digest/><resource/
></query></iq>
SENT:<iq type="set" id="2"><query xmlns="jabber:iq:auth"><username>almanei</username><digest>f8a0c4fba0c40af1e8fffeae23
27d5f90e0bedc7</digest><resource>home</resource></query></iq>
RECV:<iq type="result" id="2"/>
SENT:<iq type="get" id="3"><query xmlns="jabber:iq:roster"/></iq>
SENT:<presence type="available"><show>chat</show></presence>
RECV:<iq type="set"><query xmlns="jabber:iq:roster"/></iq>
RECV:<iq type="result" from="almanei@rhymbox.com/home" id="3"><query xmlns="jabber:iq:roster"><item jid="almanei@jabber.
com" subscription="both"/><item jid="almanei@tipic.com" subscription="both"/></query></iq>
SENT:
SENT:
```

FIGURE 4.2: The Debug Console plug-in.

JJC does not support the creation and registration of an account with public servers nor there exist a server for it. Hence, the user needs to have a working Jabber account from any publicly available Jabber servers around the globe. The sign on window is shown in figure 4.3:



Host: rhymbox.com
Username: almanei
Password: *******
Status: Ok.

FIGURE 4.3: Logon Window.

After the user logged in, the friend list will be displayed directly and the presence will be noticeable by a light icon, figure 4.4. This is very important since this secure model will not really be useful if the receiver is not online! All key exchange requests will reside on the server and will be outdated!

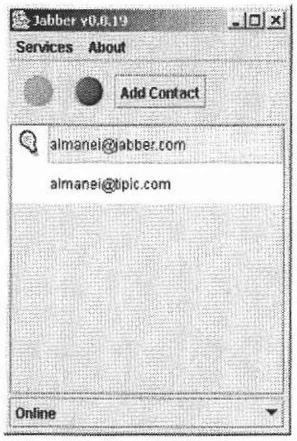


FIGURE 4.4: Client Friend List.

By double clicking on any of the names on the friend list, a message window will appear, which has 4 buttons as shown in figure 4.5:

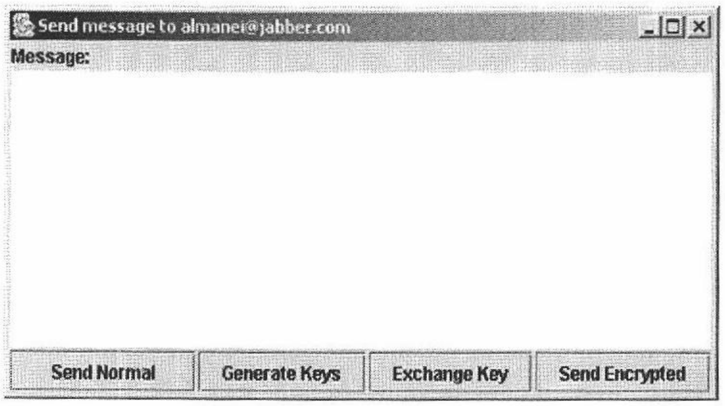


FIGURE 4.5: An empty message window.

The first button sends the message as normal (ascii) code with no encryption at all and this is the actual messenger capabilities before the modification presented in this thesis it only has a send button. However, the modified version shown above has three buttons which have the following roles:

Generate Keys This is the button where it will generate and store the Diffie-Hellman keys in a folder that have the name of the intended receiver.

Exchange Key This button will read the stored public key and encode it in (base64) format then send it to the receiver.

Send Encrypted This button only has an action if a DES key have already been created by the user which required the key exchange to be completed first.

Figure 4.6 shows the result of the Generate Keys button when it is clicked. It will create the folder named **jabberkeys** which will hold the keys of the users which the client will exchange the secure messages with.

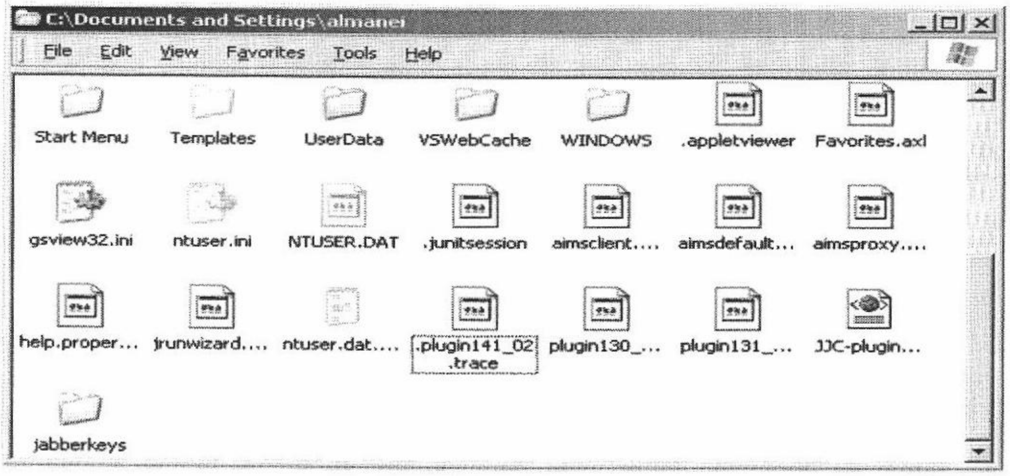
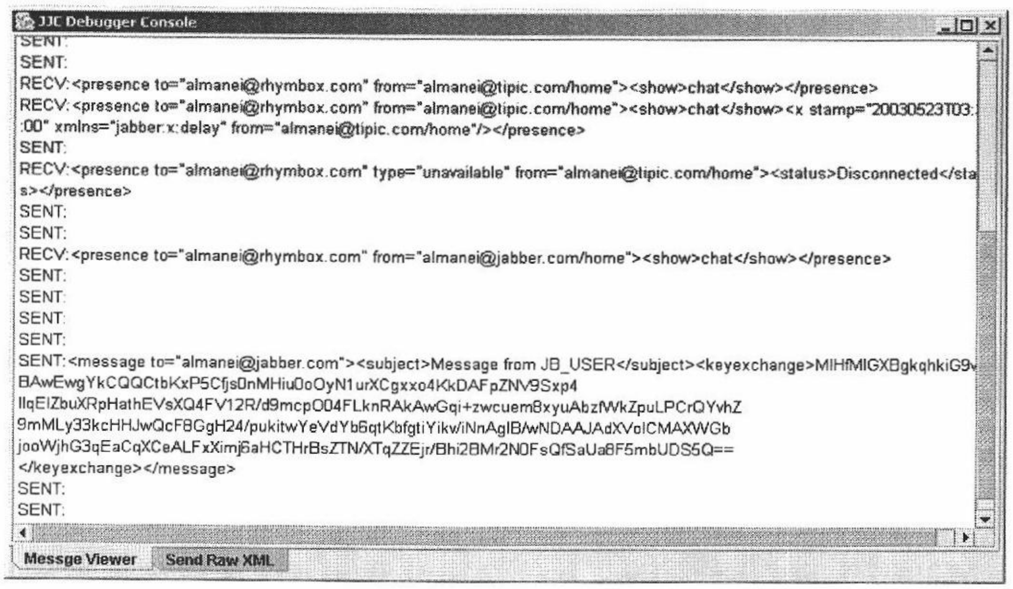


FIGURE 4.6: The generated keys folder.

After the key generation, the sender will click on the key exchange button and the messenger will read the public key of the sender in (base64) format and then wrap it in the tag < *keyexchange* > as shown in figure 4.7:



```
SENT:
SENT:
RECV:<presence to="almanei@rhymbox.com" from="almanei@tipic.com/home"><show>chat</show></presence>
RECV:<presence to="almanei@rhymbox.com" from="almanei@tipic.com/home"><show>chat</show><x stamp="20030523T03:
:00" xmlns="jabber:x:delay" from="almanei@tipic.com/home"/></presence>
SENT:
RECV:<presence to="almanei@rhymbox.com" type="unavailable" from="almanei@tipic.com/home"><status>Disconnected</sta
s></presence>
SENT:
SENT:
RECV:<presence to="almanei@rhymbox.com" from="almanei@jabber.com/home"><show>chat</show></presence>
SENT:
SENT:
SENT:
SENT:
SENT:<message to="almanei@jabber.com"><subject>Message from JB_USER</subject><keyexchange>MIHfMIGXBgkqhkiG9
BAwEwgYkCQQCtbKxP5CfjsDnMHiu0oOyN1urXCgxxo4KkDAFpZNV9Sxp4
IlqEIZbuXRpHathEVsXQ4FV12R/d9mcpO04FLknRAkAwGqi+zwcuem8xyuAbzfWkZpuLPCrQYvhZ
9mMLy33kcHHJwQcF8GgH24/pukitwYeVdYb6qtKbfgtiYikv/iNnAgIB/wNDAAJAdXVolCMAXWGb
jooWjhG3qEaCqXCeALFxXimj6aHCTHrBsZTN/XTqZZEjr/Bhi2BMr2N0FsQfSaUa8F5mbUDS5Q==
</keyexchange></message>
SENT:
SENT:
```

FIGURE 4.7: Key exchange XML packet.

The other party will receive the key exchange message,figure 4.8, and for simplicity it will not show any information about the key nor the message.

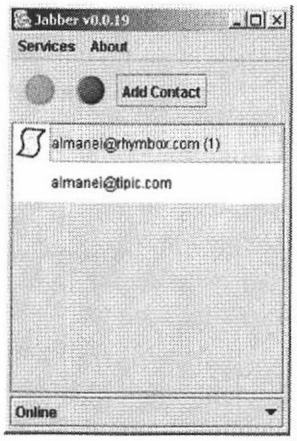
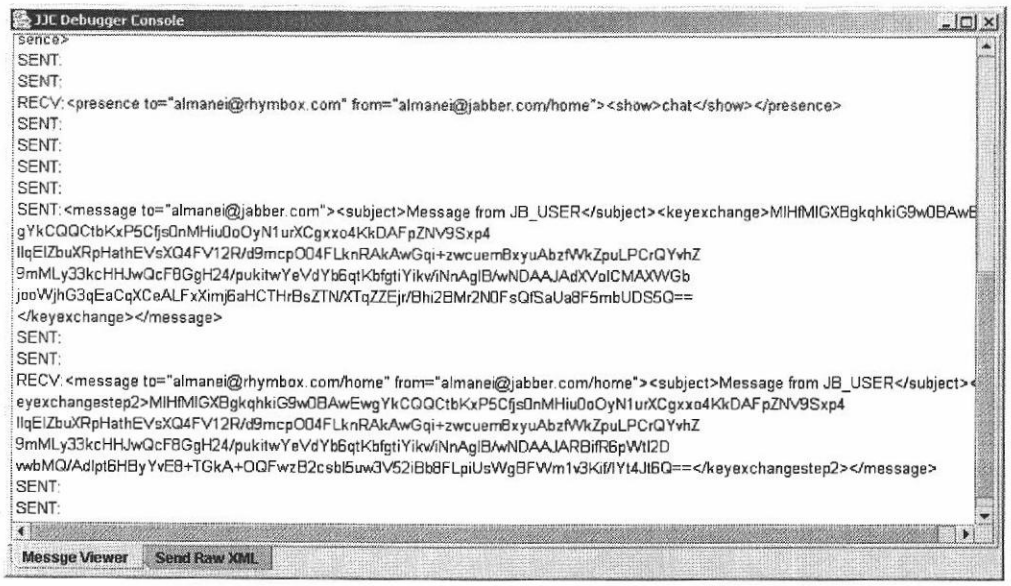


FIGURE 4.8: Key exchange message (receiver).

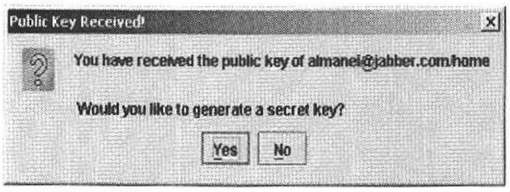
The user will double click on the name which hold the message and the execution of generating the key material from the sender public key will start. Also since the public key of the sender already available the secret key will be generated and the public key of the receiver will go directly to the sender without requiring the user action. The sender will receive the key exchange response as $< keyexchangestep2 >$ XML tag as shown in figure 4.9.



```
sence>
SENT:
SENT:
RECV: <presence to="almanei@rhymbox.com" from="almanei@jabber.com/home"><show>chat</show></presence>
SENT:
SENT:
SENT:
SENT:
SENT: <message to="almanei@jabber.com"><subject>Message from JB_USER</subject><keyexchange>MIHfMIGXBgkqhkiG9w0BAwE
gYkCQQCtbKxP5CfjsOnMHiu0oOyN1urXCgxxo4KkDAFpZNV9Sxp4
IlqEIZbuXRpHathEVsXQ4FV12R/d9mcpO04FLknRAkAwGqi+zwcuemBxyuAbzfWkZpuLPCrQYvhZ
9mMLy33kcHHJwQcF8GgH24/pukitwYeVdYb6qtKbfgtiYikv/iNnAgIB/wNDAAJAdXVolCMAXWGb
jooWjhG3qEaCqXCeALFxXimj6aHCTHrBsZTN/XTqZZEjr/Bhi2BMr2N0FsQfSaUa8F5mbUDS5Q==
</keyexchange></message>
SENT:
SENT:
RECV: <message to="almanei@rhymbox.com/home" from="almanei@jabber.com/home"><subject>Message from JB_USER</subject><
eyexchangestep2>MIHfMIGXBgkqhkiG9w0BAwEwgYkCQQCtbKxP5CfjsOnMHiu0oOyN1urXCgxxo4KkDAFpZNV9Sxp4
IlqEIZbuXRpHathEVsXQ4FV12R/d9mcpO04FLknRAkAwGqi+zwcuemBxyuAbzfWkZpuLPCrQYvhZ
9mMLy33kcHHJwQcF8GgH24/pukitwYeVdYb6qtKbfgtiYikv/iNnAgIB/wNDAAJARBifR6pWtI2D
vwbMQ/Adlpt6HByYvE8+TGkA+OQFwzB2csbl5uw3V52iBb8FLpiUsWgBFWm1v3Kif/IYt4Jt6Q==</keyexchangestep2></message>
SENT:
SENT:
```

FIGURE 4.9: Receiving other user public key XML packet.

However, a confirmation message is displayed when a key is received asking the sender wether to proceed with the secret key generation or ignore the task, figure 4.10.



You have received the public key of almanei@jabber.com/home

Would you like to generate a secret key?

FIGURE 4.10: Receiving a public key window.

When both users generate the keys and exchange their public keys, both of them will have the same share secret which in fact transformed to a DES format and used as a DES key to encrypt the message data. both of them will have the following folders associated with the user name as an identifier, see figures 4.11 and 4.12.

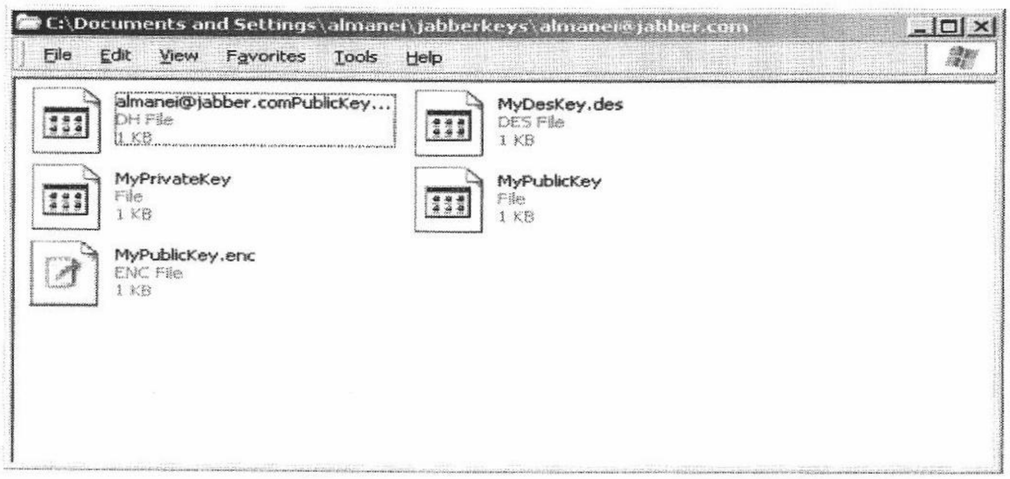

C:\Documents and Settings\almanei\jabberkeys\almanei@jabber.com
File Edit View Favorites Tools Help
almanei@jabber.comPublicKey...
DH File
1 KB
MyDesKey.des
DES File
1 KB
MyPrivateKey
File
1 KB
MyPublicKey
File
1 KB
MyPublicKey.enc
ENC File
1 KB

FIGURE 4.11: The keys files folder.

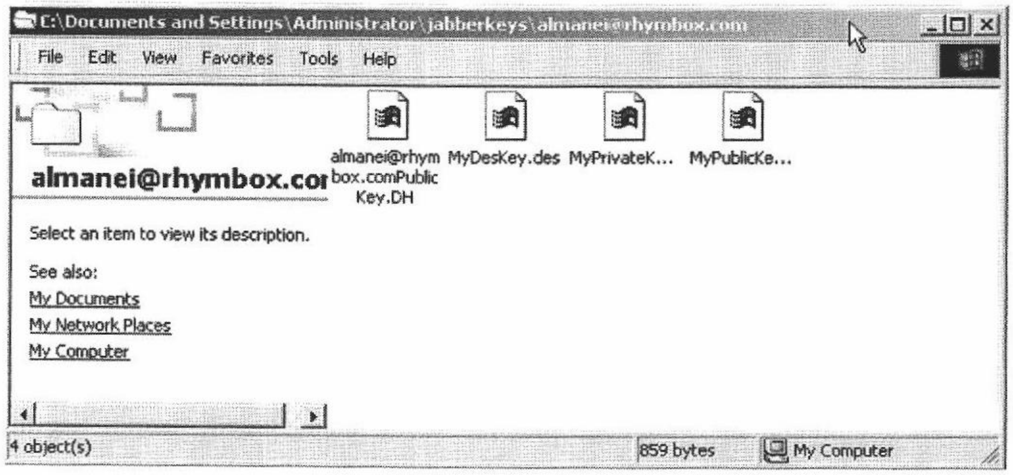
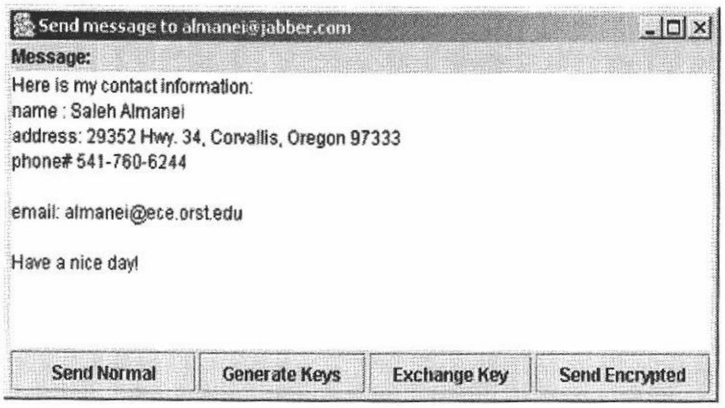

C:\Documents and Settings\Administrator\jabberkeys\almanei@rhymbox.com
File Edit View Favorites Tools Help
almanei@rhymbox.com
almanei@rhymbox.comPublicKey.DH
MyDesKey.des
MyPrivateK...
MyPublicKe...
Select an item to view its description.
See also:
My Documents
My Network Places
My Computer
4 object(s)
859 bytes
My Computer

FIGURE 4.12: The keys files folder for the other user.

Now, either user can start encrypting messages. Figure 4.13 is an example of a message that the user will send to the other party encrypted.



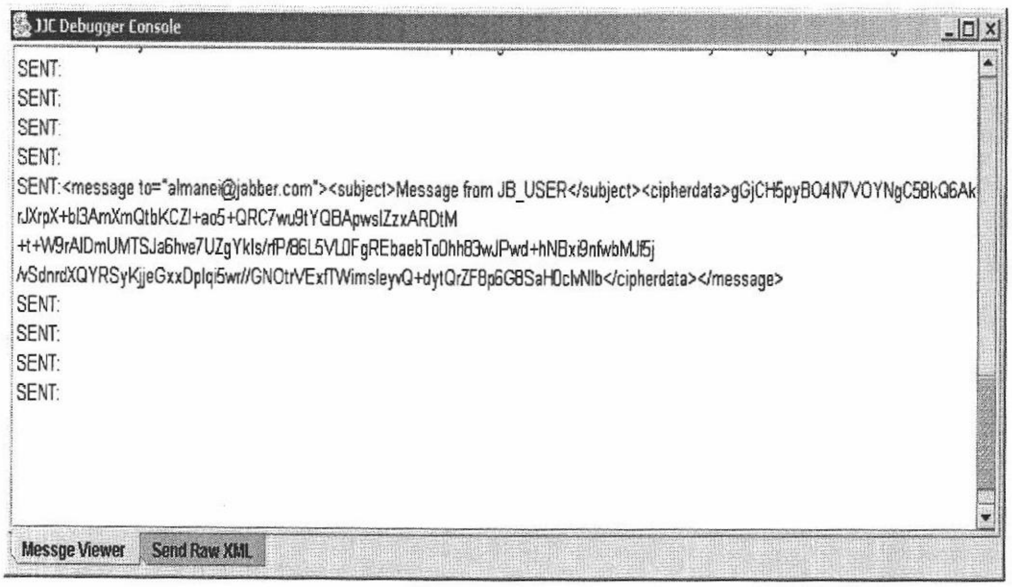
Here is my contact information:
name : Saleh Almanei
address: 29352 Hwy. 34, Corvallis, Oregon 97333
phone# 541-760-6244

email: almanei@ece.orst.edu

Have a nice day!

FIGURE 4.13: Example message which will be encrypted.

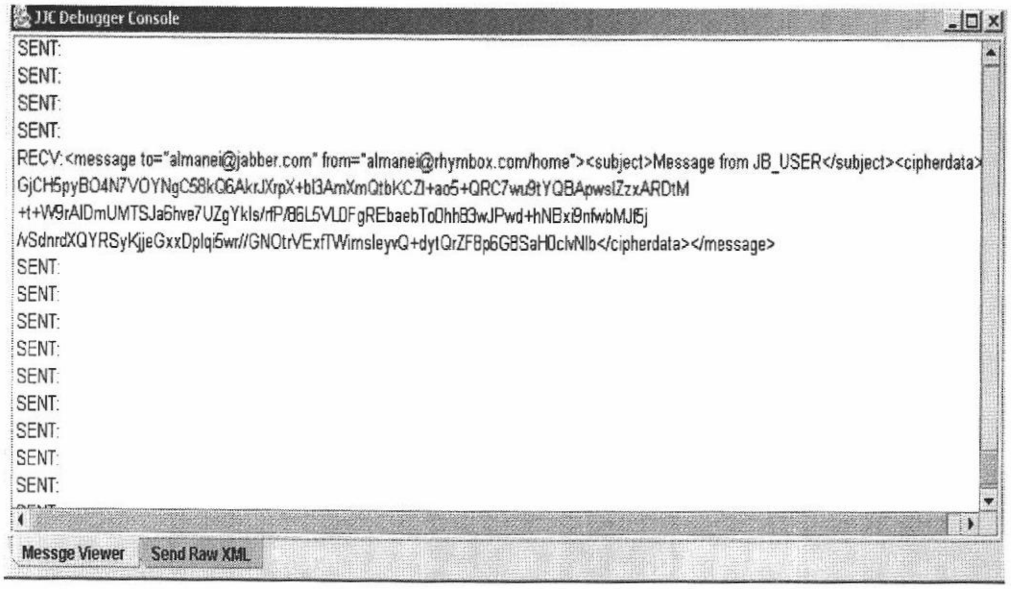
We can see the encoded < *cipherdata* > in (base64) format after pressing the send encrypted button in figure 4.14.



```
SENT:
SENT:
SENT:
SENT:
SENT:<message to="almanei@jabber.com"><subject>Message from JB_USER</subject><cipherdata>gGjCH5pyBO4N7VOYNgC58kQ6Ak
rJXrpX+bl3AmXmQtbKCZI+ao5+QRC7wu9tYQBApwsIZzxARDtM
+t+W9rAIDmUMTSJa6hve7UZgYkls/rfP/86L5VL0FgREbaebToDhh83wJPwd+hNBxi9nfwbMJf5j
/vSdnrdXQYRSyKjjeGxxDplqi5wr//GNOtrVExfTWimsleyvQ+dytQrZF8p6G8SaH0cMNlb</cipherdata></message>
SENT:
SENT:
SENT:
SENT:
```

FIGURE 4.14: The encrypted message XML packet.

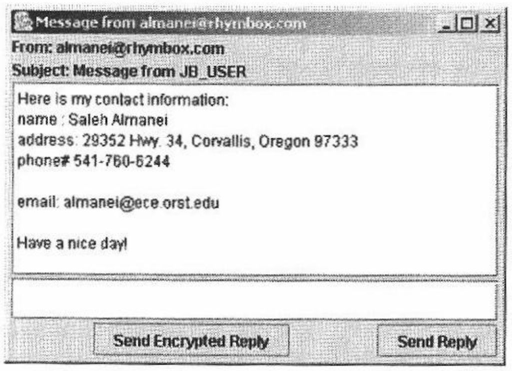
The receiver on the other end should receive the same encoded message that the sender sent as shown in figure 4.15.



```
SENT:
SENT:
SENT:
SENT:
RECV:<message to="almanei@jabber.com" from="almanei@rhymbox.com/home"><subject>Message from JB_USER</subject><cipherdata>
GjCH5pyBO4N7VOYNgC58kQ6AkrJXrpX+bl3AmXmQtbKCZI+ao5+QRC7wu9tYQBApwslZzxARDtM
+t+W9rAIDmUMTSJa6hve7UZgYkls/rfP/86L5VL0FgREbaebTo0hhB3wJPwd+hNBxi9nfwbMJf5j
/vSdnrdXQYRSyKjjeGxxDplqi5wr//GNOtrVExfTWimsleyvQ+dytQrZF8p6G8SaH0clvNlb</cipherdata></message>
SENT:
SENT:
SENT:
SENT:
SENT:
SENT:
SENT:
SENT:
SENT:
```

FIGURE 4.15: The encrypted message XML packet(receiver.)

Figure 4.16 shows the reply window which displays the message received.



From: almanei@rhymbox.com
Subject: Message from JB_USER

Here is my contact information:
name : Saleh Almanei
address: 29352 Hwy. 34, Corvallis, Oregon 97333
phone# 541-760-6244

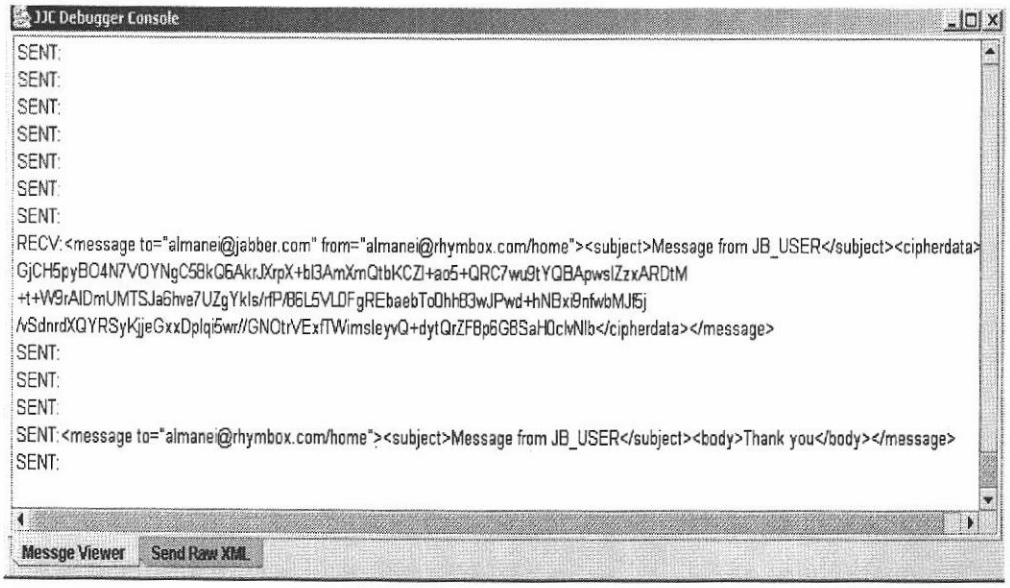
email: almanei@ece.orst.edu

Have a nice day!

FIGURE 4.16: Reply message window.

When the receiver double click on the name that has a message waiting, the program will check for the message tag and if its $< cipherdata >$ then it will fetch the DES key corresponding to the sender and will decrypt the cipher data and display it in a reply window with the option to encrypt the reply as shown above.

however, if the receiver didn't want to encrypt the message then the button "Send Normal" will do just that as shown in figure 4.17.



```
SENT:
SENT:
SENT:
SENT:
SENT:
SENT:
SENT:
RECV:<message to="almanei@jabber.com" from="almanei@rhymbox.com/home"><subject>Message from JB_USER</subject><cipherdata>
GjCH5pyBO4N7VOYNgC58kQ6AkrJXrpX+bl3AmXmQtbKCZl+ao5+QRC7wu9tYQBApwslZzxARDtM
+t+W9rAIDmUMTSJa6hve7UZgYkls/rfP/86L5VLDFgREbaebToDhh83wJPwd+hNBxi9nfwbMJf5j
/vSdnrdXQYRSyKjjeGxxDplqi5wr//GNOtrVExfTWimsleyvQ+dytQrZF8p6G8SaH0clvNlb</cipherdata></message>
SENT:
SENT:
SENT:
SENT:<message to="almanei@rhymbox.com/home"><subject>Message from JB_USER</subject><body>Thank you</body></message>
SENT:
```

FIGURE 4.17: Normal reply XML packet.

This illustration should give a clear view of the graphical interface and the user action. Also, with the help of the Debug Console the user or developer can check exactly how does the message is parsed using the Java XML parser.

5. EXPERIMENTAL RESULTS AND ANALYSIS.

5.1. Simulation Environment.

This Secure Jabber Client application presented in this thesis was developed using the SUN One Studio environment installed in a Windows XP Professional machine. It have all the tools required to link and compile the project code. However, since the code is written in Java then there will be no problem in compiling it in any other Java compiler or environment as long as the JRE 1.4.2 that have the JCE cryptographic library exist in the CLASSPATH.

There were no significant problems in coding the cryptographic codes nor in modifying the JJC code; however, understanding how the code implements the Jabber protocols and most importantly parsing the XML format and getting the values from an XML formatted message were the most time consuming in regards to implementation.

5.2. Jabber Security Analysis.

The working model in this thesis is far from complete as a secure Jabber client. However, this demonstration should open the door for developers to start designing their custom security functions for the Jabber protocol and contribute to the Jabber community to finalize the standards for securing a Jabber environment.

On the other hand, this model can be adopted by the regular users who care about their privacy from the server administrators since all the message data is encrypted using a not so easy, but possible to crack, algorithm known as DES.

There are some vulnerabilities that have to be mentioned to clarify the actual security strength of the Secure JJC. The Intruder-in-the-Middle attack is possible for the Diffie-Hellman key exchange where an intruder can fool the two users.[12]

DES is also an old algorithm that is going to be replaced with Advance Encryption Standard (AES). It have been possible to break DES in a matter of days, and with the advancement in technology there will be no place for DES as a secure algorithm in the future.[13] However, the design implemented here can easily be extended to support the new AES algorithm and make DES an optional algorithm or replace it with the modified algorithm named TripleDES.

JJC original design didn't support any cryptographic algorithms and so do most of the Jabber clients available to the public except those clients which support OpenPGP as mentioned earlier. The following section will compare the two designs in a graphical illustration and also discuss the features added.

5.3. Comparison to the Original Design

The comparison to the original design will have two sections. The first is the graphical comparison while the second present the features added and representation of the table introduced at chapter 2.

5.3.1. Graphical Interface

The new implementation of the JJC differ from the original design in two graphical areas as seen in the figures below: Figure 5.1 shows the original message window while

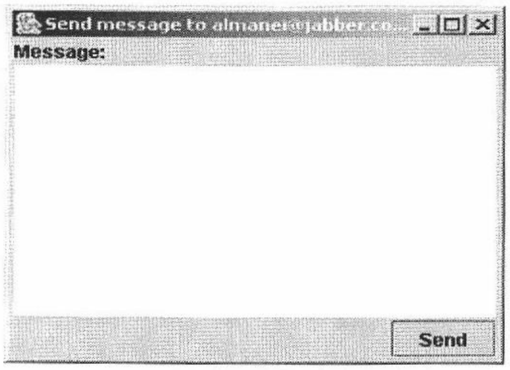


FIGURE 5.1: Original Message Window.

Figure 5.2 shows the modified message window. Also the Reply Message Window is

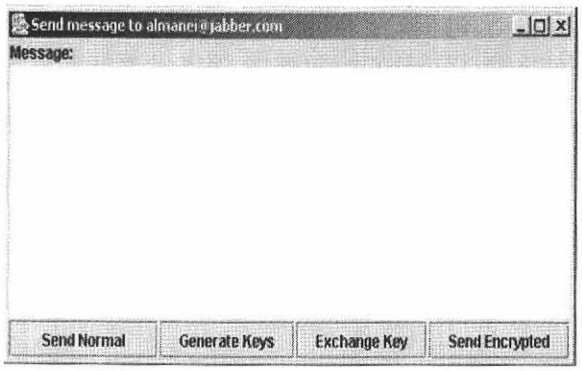
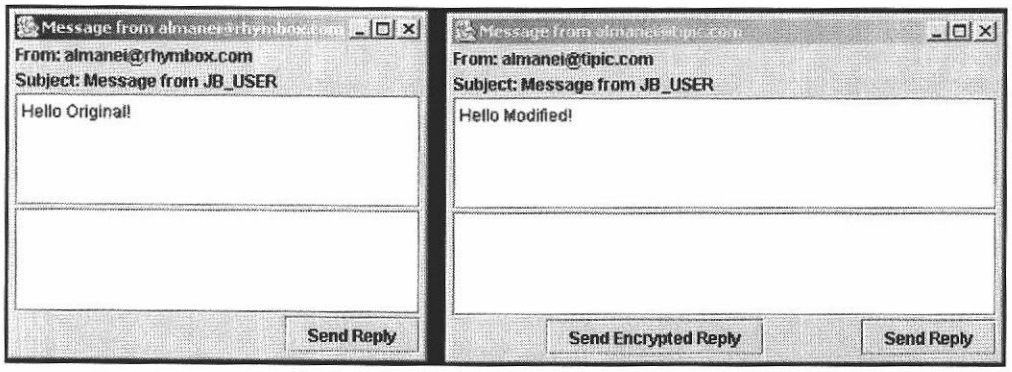


FIGURE 5.2: Modified Message Window.

modified from the original to include the "Send Encrypted Reply" button as shown below:


Message from almanei@rhymbox.com
Send Reply
Message from almanei@tipic.com
Send Encrypted Reply
Send Reply

From: almanei@rhymbox.com
Subject: Message from JB_USER
Hello Original!

From: almanei@tipic.com
Subject: Message from JB_USER
Hello Modified!

FIGURE 5.3: The Original and Modified Reply Message Window.

5.3.2. Features

As mentioned in chapter 2 that current security implementation for Jabber are inadequate. The application presented in this work can be used against that fact since it proofs the ability to support almost any symmetric or asymmetric cryptographic algorithms. which will modify the table 2.1 as shown:

Security Property	*Description*	*Existing Technology*	*Jabber Support*
Authentication	Ensures an entity is who it claims to be.	JAASI, Kerebos, Microsoft Passport	Jabber Authentication Protocol and SASL
Authorization	Determines what an entity has permissions for accessing(data) or controlling(resources).	JAAS, access control lists	Binary authorization. Unauthenticated users are granted certain rights, and authenticated users others.
Integrity	Ensures data has not been tampered with.	Message digests.	Can Support Message digests
None-repudiation	Ensures that the author of data can always be identified.	Digital signatures.	Can support Digital signature.
Confidentiality	Ensures data can be read only by authorized entities.	Encryption	Full Encryption support.

TABLE 5.1: Computer security properties compared with Jabber's modified capabilities.

The table above shows an increased chance in adopting Jabber based IM as the future IM protocol.

The next chapter will discuss the future of Jabber and the future work that can be done to increase the Jabber protocol adoption in the network communities.

6. CONCLUSION AND FUTURE WORK.

6.1. Can Jabber be Secure?

This thesis proved that existing Jabber protocol can be secured easily when the required modification is made. However, when looking at the market point of view of security, in the present time, we can see that Jabber is not a good candidate for an Instant Messaging System which the industry is shopping for since it is lacking the security support the industry and corporate are looking for.[14] However, this work can be extended to support more cryptographic algorithms and protocols, such as digital certificates, digital signatures, public key cryptography, i.e. RSA.

Judging from the results of this project, Jabber protocol will be very successful if Jabber developers start designing support for cryptographic libraries or even have a standard open source library as a backbone for security tasks carried by Jabber, such an example is the new release of the XML encryption library (still in beta version) by the W3C, which can be used in the Jabber protocol since its based on XML format.

As seen in table 5.1, Jabber future can provide most if not all the security requirement that the industry is looking for, confidentiality, authorization, authentication, Integrity, and finally non-repudiation.

6.2. Jabber is not only IM

The Jabber protocol is not limited to Instant Messaging alone, its far more flexible to support applications that are used daily like scheduling meetings and White boarding!

The Jabber Protocol is extensible since its relies on XML in every piece of the core protocol which means that anyone who have good imagination or even have a specific task that can be deployed in a network, Jabber will probably can support it with little modifications to extend the core protocol.

6.3. Future work

The future of a secure Jabber environment is very promising and the work done by the XMPP group to standardize the end-to-end encryption and the added feature of SSL/TLS and SASL in the core protocol is a sign of such future. Developers that implement their own specific design without actually modifying the core Jabber protocol can be a valuable source of speeding the task to reach the goal of having a well balanced secure Jabber system. This thesis is an example of such work that can be added to the community of Jabber users and developers.

BIBLIOGRAPHY

1. "What is jabber," WWW page, 2003, http://www.jabber.org.

2. Robin Cover, "IETF Charters Extensible Messaging and Presence Protocol (XMPP) Working Group.," WWW page, 2002, http://xml.coverpages.org/.

3. Iain Shigeoka, *Instant Messaging in Java The Jabber Protocols*, Manning Publications Co., 2002.

4. Stephen Lee and Terence Smelser, *Jabber Programming*, Hungry Minds, Inc., 2002.

5. J. Miller P. Saint-Andre, "Xmpp core draft-ietf-xmpp-core-12," WWW page, May 2003, Expire on November 2, 2003.

6. P. Saint-Andre, "End-to-end object encryption in xmpp," WWW page, May 2003, Expire on November 18, 2003.

7. Paul Lloyd, "A Framework For Securing Jabber Conversations.," WWW page, 2002, http://www.jabber.org/.

8. Naokhiko Uramoto Makoto Murata Andy Clark Yuichi Nakamura Ryo Neyama Kazuya Kosaka Hiroshi Maruyama, Kent Tamura and Satoshi Hada, *XML and Java Second Edition, Developing Web Applications*, Prentic, 2002.

9. Donald E. Eastlake III and Kitty Niles, *Secure XML The new Syntax for Signature and Encryption*, Addison-Wesley, Person Education, 2002.

10. Cay S. Horstmann and Gary Cornell, *Core Java2, Volume II-Advanced Features*, Addison-Wesley, Person Education, 2002.

11. "Java Cryptography Extension," WWW page, 2003, http://java.sun.com/.

12. H.X. Mel and Doris Baker, *Cryptography Decrypted*, Addison-Wesley, Person Education, 2001.

13. Wade Trappe and Lawrence C. Washington, *Introduction to Cryptography with Coding Theory*, Prentic-Hall, Inc., 2002.

14. Symantec Security Response, "Secure instant messaging," White Paper, May 2003, http://www.symantec.com/avcenter/whitepapers.html.