View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Theseus

Mohammad Milon

Communication over Internet with Instant Messaging

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Thesis
25 April 2012

Metropolia

Abstract

Author Title Number of Pages Date	Mohammad Milon Communication over Internet with instant messaging 45 pages + 3 appendices 25 April 2012
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software
Instructor	Dr. Mahbubur Rahman, Lecturer

The goals of this project were to explain briefly all methods of the Java platform that allow networking communication and to design and implement a multi-client chat application for desktop computers and mobile devices which support instant messages. The chat application that was created is based on a client-server model which means all clients communicate with each other with the help of a server. The server is responsible for sending and receiving messages.

The application was developed on Eclipse IDE with the Remote Method Invocation (RMI) plugin since the RMI method was used for a desktop chat application. The EclipseME plugin was also installed on Eclipse IDE to implement the mobile client side application.

The results of this project demonstrated that it is possible for multiple clients connected on the internet to communicate in real-time manner and share information with one another. Since the chat application was object-oriented programming, it provides the application with a high level of reusability and also makes it possible to employ only certain parts of the software.

Keywords	VOIP, IRC, RMI chat, JSP, TCP

Contents

1 Introduction 1

2 Internet-based Communication 3

2.1 Instant Messaging (IM) 3
2.2 Email 4
2.3 Internet Telephony and VOIP 5
2.4 Internet Relay Chat (IRC) 5
2.5 Videoconferencing 6

3 Common Technologies in Chat Systems 7

3.1 Client-Server Model 7
3.2 Multi-threading 8

4 Existing Methods 10

4.1 Sockets and Server Sockets 10
4.2 Remote Method Invocation (RMI) 15
4.3 Java Server Pages (JSPs) 21

5 Development Environments 26

5.1 Eclipse Installation 26
5.2 Installing Plugin 26

6 Implementation of a Chat Application 27

6.1 Overview 27
6.2 Graphical User Interface (GUI) 28
6.3 Class Diagram 32
6.4 RMI Registry 35
6.5 GUI for Mobile Client 36

7 Analysis of Results 40

7.1 Desktop Chat 40
7.2 Mobile Chat 41

8 Conclusion 43

References 44

Appendices

Appendix 1: Server Implementation
Appendix 2: GUI for Client
Appendix 3: Client Implementation

1 Introduction

Communication is a process by which information is exchanged between individuals or devices through a common system. Information can be transmitted using various communication methods. For successful completion of transmitting information, all communication methods have four common elements: information source or sender, message or information, media that provides the pathway by which the message can be transmitted from source to destination, and receiver. Nowadays, the internet has become the most popular communication system for sharing information.

The internet consists of globally interconnected computer networks that serve millions of users worldwide. This network connects many independent networks such as millions of private, public, academic, business, and government networks that are tied by an extensive array of electronic, wireless and optical networking technology. The internet makes it possible to communicate in various ways. Email is a momentous communication service available on the internet. Internet chat, such as Internet Relay Chat (IRC) that uses an instant messaging system or a social networking site, is another important communication service that allows colleagues to stay in touch in a very convenient way during their working time. Internet telephony, voice over internet protocol (VOIP), and videoconferencing are other important examples of communication services.

The objectives of this thesis are to clarify and discuss all possible ways the Java platform supports networking communication, and to design and implement a multi-client chat application for desktop computers and mobile devices which allows instant messaging. With this chat application users can communicate from a PC or from a mobile device in a uniform way. This chat application is based on a client-server model. The client is a program that runs on the computer and sends and receives messages to and from a chat server. The chat server is responsible for making sure that all messages are broadcast to everyone or a particular user. Once users join a chat room, they can type messages in the public chat room where all the participants can see

them, or users can send private messages to single participants. J2SE/J2EE was used for desktop computers and J2ME technology was used for mobile devices.

2 Internet-based Communication

2.1 Instant Messaging (IM)

Instant messaging (IM) is a tool used by the vast majority internet users. IM functions as a text-based computer conference between two or more people. An IM communication service creates a kind of private chat room with another individual in order to communicate in real-time over the internet. This form of messaging, in comparison to e-mail, allows for quick and easy responses to satisfy one´s need for instant and constant communication.

IM is used by millions of internet users to contact family and friends. It is also growing in popularity in the business world. Employees of a company can have instant access to managers and co-workers in a different office and can eliminate the need to place phone calls when information is required immediately. Overall, IM can save time for employees and help decrease the amount of money a business spends on communication.

Different IM clients offer slightly different features and benefits, but the main concept of an IM client is basically the same. Public IM clients and services available include ICQ, AOL Instant Messenger (AIM), Microsoft MSN Messenger, and Yahoo! Messenger. For businesses and enterprises looking for a more secure method of IM, there are enterprise IM packages available such as Microsoft Live Communication Server 2005 and IBM Lotus Instant Messaging.

There are some drawbacks and issues associated with IM spim and virus protection. Spim is the equivalent of spam and is perpetuated by bots that harvest an IM screen name off of the internet and simulate a human user by sending spim to the screen name via an instant message. Additionally, viruses and trojans can be spread through IM channels. These malicious programs are usually spread when an IM user receives a message that is linked to a Web site where the malicious code is downloaded. [14]

2.2 Email

Electronic mail, shortened as e-mail, is the transmission of a message over a communication network. It is a method of transferring messages from one person to one or more recipients. Nowadays email systems are based on a store-and-forward model. A server called Email server is responsible for accepting, forwarding, delivering and storing messages. A modern email system does not required either the users or their computers to be online simultaneously.

However, some email systems are limited to a single computer or network. Others have gateways to other computer systems enabling to send email anywhere in the world. An email consists of three elements: the message envelop, the message header, and the message body. The message header contains control information including the author’s email address and one or more recipient addresses. Usually descriptive information is also added such as a subject header field and message submission date/time stamp.

Since email is the first broad electronic communication medium, it is widely accepted by the business community. Nowadays face to face meetings are no longer the essential way to communicate as one can use computer mediated communication such as email. Email like postal mail solves logistics and synchronization problems in business communication. [1]

One of the biggest drawbacks over email is spam. Spam is unsolicited commercial email. Spammers can send millions of email messages each day due to very low cost of sending email. Spamming causes information overload for many users who receive unsolicited email each day. Attachment size limitation is another drawback over email.

2.3 Internet Telephony and VoIP

Internet telephony integrates telephone services into computer networks. This technology is the association of hardware and software that enables to use the internet as the transmission medium for telephone calls. When making an internet telephone call, an internet telephony system converts an analogue voice signal into a digital signal, transmits it and converts it back again. Internet telephony software essentially provides free telephone calls anywhere in the world.

Voice over IP (VoIP) is a common internet telephony service which is growing the popularity. VoIP hardware and software work together to transmit telephone calls by sending voice data in packets using IP rather than traditional circuit switches, called PSTN (Public Switched Telephone Network). The voice traffic is converted into a data packet, then routed over the internet or any IP network. When the data packets reach their destination, they are converted back to voice data again for the recipient. [2]

2.4 Internet Relay Chat (IRC)

IRC is a multi-user, multi-channel internet text messaging system that enables people to gather on a channel to talk in groups or privately. An IRC application is a client/server application. Joining an IRC discussion, a user needs an IRC client which is a program that runs on a computer and exchanges messages between IRC servers. The IRC server broadcasts all messages to every participating user in a discussion. An IRC server also connects to other IRC servers to expand the IRC network. There can be many discussions going on simultaneously and each one is an unique channel. [1]

2.5 Videoconferencing

Videoconferencing is a process of communication and interaction between two or more participants at different sites by using the internet. This process can transmit audio and video data simultaneously. In order to do videoconferencing, users need a video camera, microphone and speaker connected to his or her computer. For the time being, videoconferencing, voice data packets and images are transmitted over the network and delivered to the destination. In recent years, this process has become a popular way of distance communication in classroom to provide distance learning, multi-school collaboration projects, and guest speakers because of its cost efficiency. Videoconferencing also provides a visual connection and interaction that cannot be achieved with standard IM or email. [1]

3 Common Technologies in Chat System

3.1 Client-Server Model

In the modern IT world, client-server architecture is a popular model for computer networking because of its versatility and flexibility. A server is a computer or software program that has some shared resource to provide a specific kind of service to client software which runs on another computer. There are many kinds of servers such as file servers, which store all kinds of files; print servers, which manage a collection of printers; and database servers, which process database queries. A client is any entity that has access to the server, requests the server for some kind of tasks and displays the result. The client-server model is composed of three components: user interface (client), processing management (server), and database management (data server). The user interface provides a user friendly layer to make requests to the server and proposes multiple forms of input and output. Processing management is responsible for process development, process implementation, and process resource service. Database management supports database and file service. Figure 1 illustrates a simple client-server model.

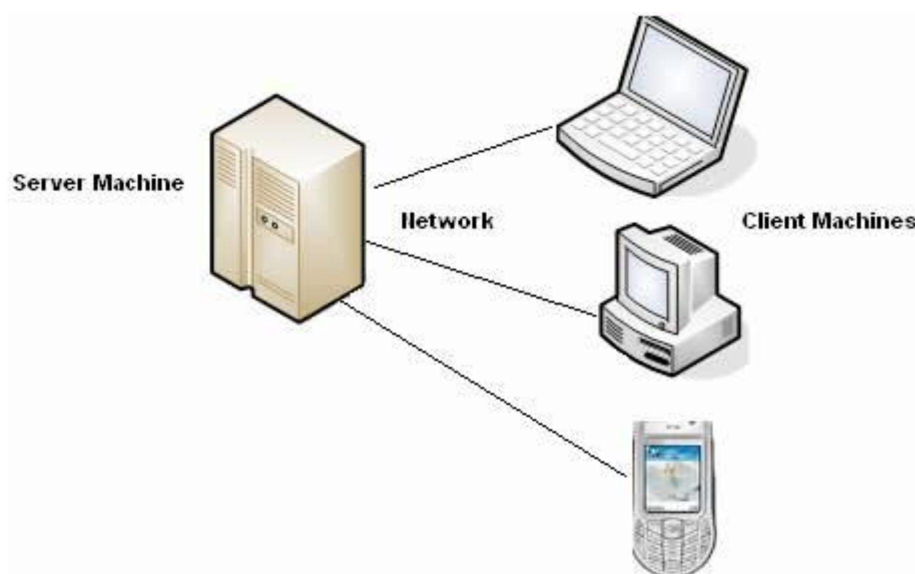

Figure 1. Client-Server Architecture. [15]

In the client-server model, resources and data security are controlled through the server. All data are stored in the server for backing up files and searching files and data easily. New technology can easily be integrated in this model according to different users and different system requirements. Hardware or software can be replaced individually to meet user requests. The client network and server function together with other systems without requiring human intervention. [3]

The client-server network model has some limitations. Since this model relies on a centralized server, the entire system cannot work if the server goes down due to any kind of problem. The server is responsible for the majority of the network traffic, as all queries for resources are directed towards the server. This can cause network overload and slow down the whole system. A single network administrator with a high level of IT skills is required to manage and maintain the equipment and the whole network while other network systems, such as peer-to-peer network systems, do not require an administrator. This causes high expenses for the client-server architecture. [4]

3.2 Multi-threading

The necessity of threading has grown because a graphical interface has become the standard for desktop computers. Multi-threading systems enable the user to understand program performance better. A multi-threaded program consists of two or more parts that are capable to run concurrently. Each part of a program is called thread, and each thread explains accurately a separate path of execution. As a result, multithreading is a specialized form of multitasking. [5]

Multithreading is supported by virtually all modern operating systems. There are two separate types of multitasking: process-based and thread-based. A process is an execution of a program. Thus, process-based multitasking enables computers to run two or more programs concurrently. For example, it allows downloading a file at the same time that a user can compile a program or sort a database. In a thread-based multitasking environment, a single program is able to perform two or more tasks

simultaneously. For instance, a word processing program can check the spelling of a word in a document while the user can write the document. [5]

Multithreading is so deeply rooted in Java that it is impossible to write even the simplest programs in Java without using threads. Java programmers have to use multithreading techniques because Java has no concept of asynchronous behaviour. Another reason to use multithreading is its lightweight behaviour. Threads share the same address space and share the same process. As a result multithreading communication is inexpensive and the advantage of context switching from one thread to another is low cost. Furthermore, multithreading allows writing very efficient programs that make maximum use of the CPU. [6]

4 Existing Methods

4.1 Sockets and Server Sockets

The internet and WWW (World Wide Web) have come into view as global omnipresence media for communication, and changing the way of conducting science, engineering, and commerce. Java APIs provide Socket and ServerSocket classes as an abstraction of standard Transmission Control Protocol (TCP) socket programming techniques for networking application. A socket is an end-point of bidirectional communication link between two programs running on the network. The Socket class provides a client-side socket interface similar to UNIX sockets. The java.net package contains a collection of two classes and interfaces, and Socket and ServerSocket that implement client side and server side connection, respectively. [7]

Computers communicate with each other in a very simple way. For sharing data, computers need to stream a few millions of bits and bytes back and forth agreeing on speed, sequence, and timing. In low-level networking, a set of packaged protocols do this job every time. These sets of protocols are called stacks. Nowadays the most common stack is TCP/IP. According to OSIRM (Open Systems Interconnect Reference Model), there are seven logical layers in a reliable framework for computer networking. TCP/IP maps the Transport Layer and Network Layer in the OSI (International Standards Organization) model, as shown in the following figure.

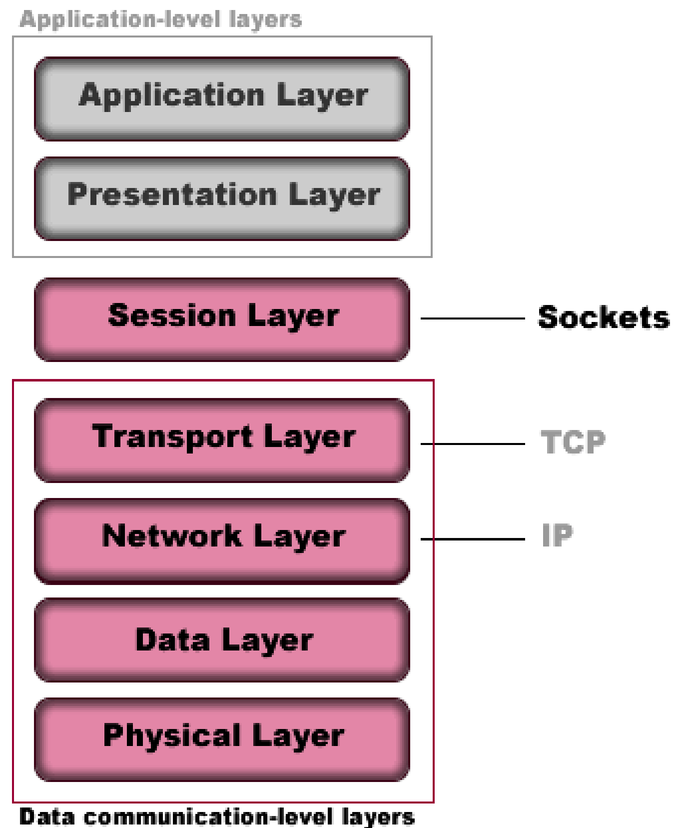

Figure 2. OSI model

Roughly, sockets stay at the Session Layer of the OSI model. The Session Layer is the interface between the application-oriented upper layers and the real-time data communication lower layers. This layer is responsible for managing and controlling data flow between two computers. In Session Layer, the socket allows an abstraction that hides the complexities of getting the bits and bytes on the wire for transmission. The code for sockets works at the Presentation Layer which provides a common representation of information that the Application Layer can use. [8]

The java.net package provides two communication protocols for socket programming.

- Datagram communication
- Stream communication

Datagram communication: The datagram communication protocol, named as UDP (User Datagram Protocol), is an OSI transport layer protocol for client/server network applications. It is a connectionless protocol because each time the socket descriptor and the receiving socket´s address are needed to send datagram. In UDP, there is a size limit of 65,535 bytes for sending a datagram to a distinct location. A datagram consists of one message unit where the first eight bytes of the datagram contain header information and the remaining bytes contain message data. There is no guarantee in UDP that sending the datagram will be received in the same order by the receiving socket.

Stream communication: The stream communication protocol is known as TCP (Transfer Control Protocol) which corresponds to the transport layer in OSI. A TCP is a connection-oriented protocol because before communication between the pair of sockets takes place, a connection must be established first. When the pair of sockets have been connected, they can transmit data in one or both directions. In a TCP, there is no size limit and two sockets behave like streams. A TCP is a reliable protocol because TCP ensures that sending packets will be received in the same order in which they are sent.

The java.net package provides a ServerSocket class to create servers. When a ServerSocket is created, it will register itself with the system for client connection. A server socket´s job is to run on the server and listen to either local or remote client programs. Each ServerSocket listens to a port on the server. When a client socket attempts to connect to the port, the server wakes up, negotiates the connection between the client and the server, and opens a regular socket. Data always travel over the regular socket. The ServerSocket class has a constructor that creates a ServerSocket object, methods that listen to the connection and methods that return a Socket object to send and receive data.

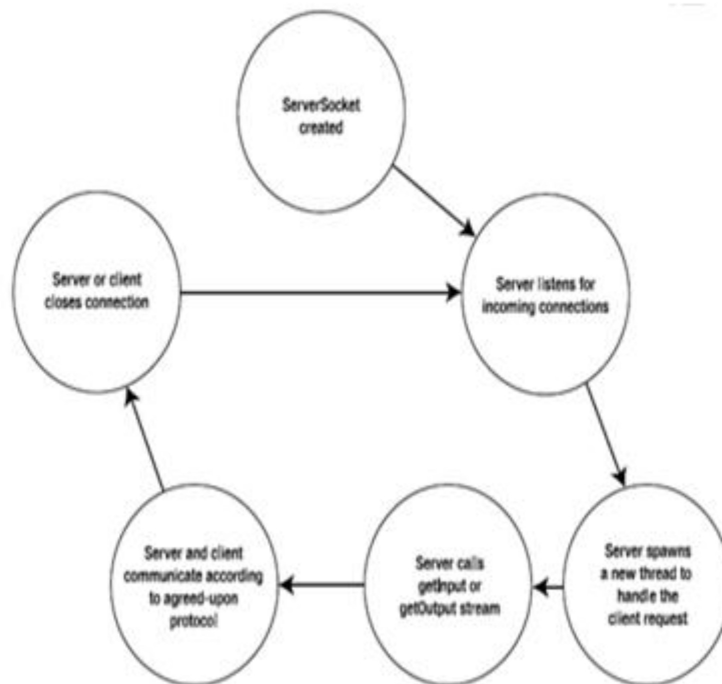

Figure 3. Life cycle of ServerSockets

Figure 3 demonstrates the basic life cycle of a server:

1. A new ServerSocket is created on a particular port using a ServerSocket( ) constructor.
2. The ServerSocket listens for an incoming connection on that port using its accept() method. Accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket´s getInputStream() method, get OutputStream method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.

6. The server returns to step 2 and waits for the next connection.

The advantages and disadvantages of the Sockets and ServerSockets are the following:

Advantages

- Sockets are powerful, flexible and sufficient.
- Sockets are easy to implement for networking communications.
- If efficiently used, sockets cause low network traffic.
- It is different from the HTML form and CGI script that generate and transfer the whole web page for each new request.
- Only updated information can be sent.

Disadvantages

- Security restrictions are sometimes overbearing.
- Socket based communication enables only to send packets of raw data between the client and server.
- Both the client and server have to provide mechanism to make data useful in any way.

4.2 Remote Method Invocation (RMI)

Remote Method Invocation (RMI) is a Java provided mechanism that a Java programmer can use to write code for object-oriented programming in which objects on different computers can interact in a distributed network. It allows a Java object that is executed on one Java Virtual Machine (JVM) to invoke a method of a Java object located in another JVM. The RMI mechanism, introduced by Sun, facilitates the programmers to invoke distributed components across a network environment which is an easy alternative to the complex coding involved in server-socket programming. This mechanism interoperates between a remote object and local object. [5]

RMI uses Java Remote Method Protocol (JRMP) and Internet Inter-ORB Protocol (IIOP) to create a connection from a client to a server. This method sends a command to a server and receives back the results of the execution of that program. Java Object Serialization Protocol and Hypertext Transfer Protocol (HTTP) are responsible for the message format of RMI. The HTTP protocol uses the POST command to get its instructions executed on the server.

RMI Procedure

RMI applications consist of two separate programs, a server and a client. At the server side, the server program creates the RMI service which is for remote objects, binds it into the RMI registry with references to these objects accessible, and waits for clients to invoke methods on these objects. The client program obtains a remote reference to remote objects on the server and calls methods on them. The server and the client exchange information back and forth by using the RMI mechanism. The RMI application needs to handle the following:

- **Locate remote object:** The RMI mechanism needs to obtain references to remote objects by using RMI´s naming facility, the RMI registry, or by passing and returning remote objects.
- **Communicate with remote objects:** The RMI mechanism handles the details of communication between remote objects.

- **Load class definitions for objects that are passed around:** The RMI system provides all mechanisms for loading an object´s class definition and transmitting data. [9]

Figure 4 illustrates the RMI application that uses the RMI registry for getting reference to a remote object.

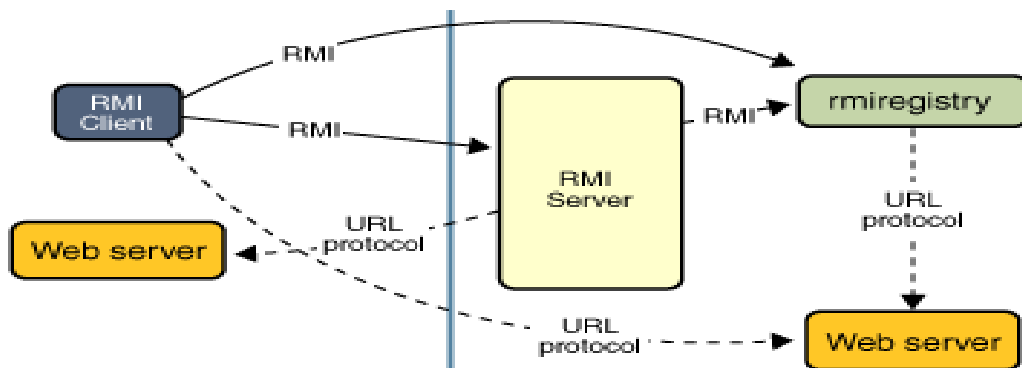

Figure 4. RMI mechanism

RMI Architecture

The RMI architecture follows one important principle: the definition of behaviour and the implementation of that behaviour are separate concepts and they run on separate JVMs. In RMI, the definition of a remote service is coded using a Java interface which does not have executable code and the implementation of that remote service is coded in a class. The RMI architecture supports two classes that implement the same interface. The implementation of the behaviour is the first class which runs on the server and the second class acts as a proxy which runs on the client. [10]

The RMI architecture consists of three abstraction layers.

- Stub and Skeleton Layer
- Remote Reference Layer
- Transport Layer

Stub and Skeleton Layer: When the client requests for a remote object, the stub and skeleton layer attend to the remote method calls and redirect these to the remote service on the server. The stub is a client-side object. The client´s request starts with the stub for invoking methods of a remote object. The stub plays the role of the proxy to the skeleton. When a client calls a server method, the JVM looks at the stub first for type checking, and then the request is routed to the skeleton on the server. The stub does the following things to invoke a remote method:

- Establishes a connection with the remote JVM
- Marshals (prepares and transmits) the parameters to the server
- Waits for the result of the methods
- Un-marshals (reads) the return value or exception
- Returns the value to the client

The skeleton is a server side proxy and it resides on the server machine. The skeleton knows how to communicate with the stub across the RMI link. The Skeleton performs the following operations for each received call:

- Un-marshals the parameters for the remote method
- Calls the method in the actual object implementation
- Marshals the result to the caller
- Dispatch the client call to the actual object implementation. [11]

Figure 5 shows the RMI runtime architecture with a different layer.

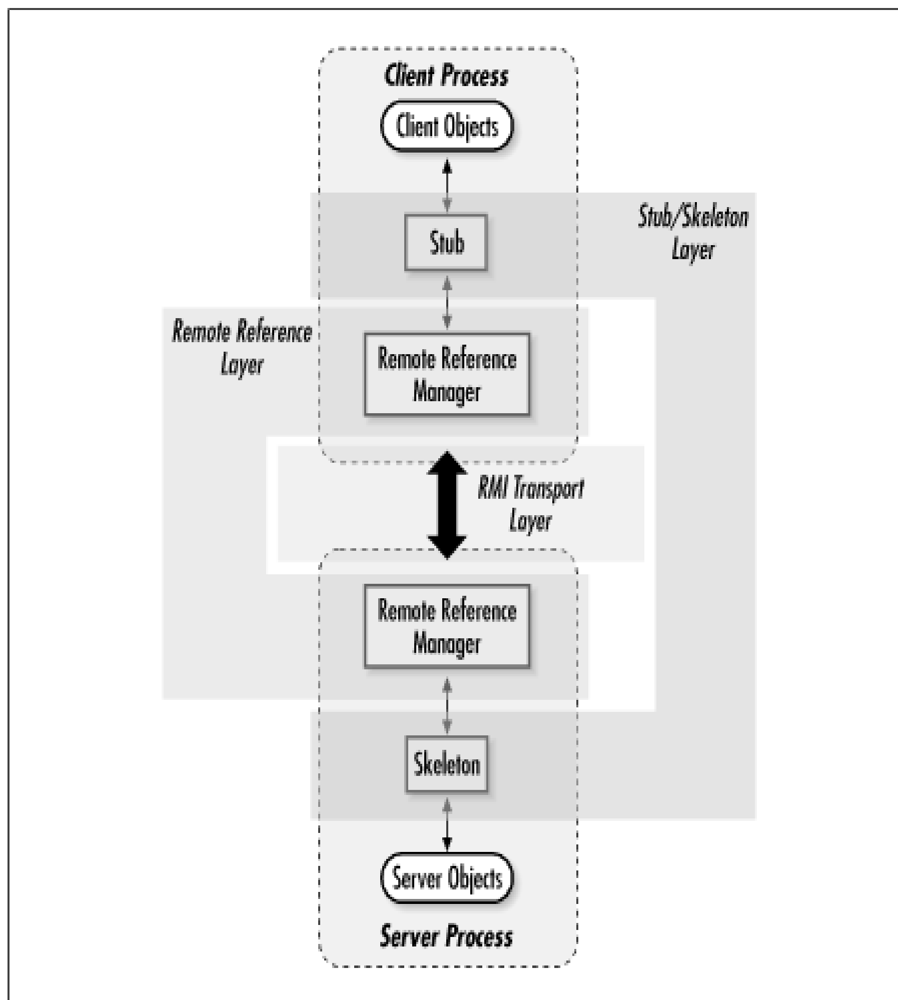

Figure 5. The RMI runtime architecture

Remote Reference Layer (RRL): The RRL is the middleware between the stub and skeleton layer and the underlying transport layer. This layer interprets and manages the references of the client´s request to the remote object on the server. The RRL resides on the client and the server. The client-side RRL receives the request for the methods from the stub and transfers them as a marshalled stream of data to the RRL of the server. [11]

Transport Layer: The transport layer is a binary data protocol. This layer links the RRL on the server side and the RRL on the client side. The transport layer is responsible for making a connection between JVMs and handling remote objects that are presented in its address space. [11]

The advantages and disadvantages of the RMI mechanism are the following:

Advantages:

- RMI is a simple and clean implementation that leads to more maintainable, robust and flexible application. It removes a great deal of terrestrial tasks like parsing and switch logic. A more complex system can be built easily using RMI.
- RMI allows establishing a distributed system while decoupling the client and server objects simultaneously.
- RMI is a pure Java solution that allows creating a zero-install client for users.
- RMI passes objects by their actual classes, so the behaviour of the objects is not changed when they are sent to another JVM. Thus, dynamic loading of classes is available in RMI.
- If the database is changed, recompilation is needed only in server, while the server interface and client remain the same.
- RMI extends the internal garbage-collection mechanisms of the Java virtual machine to provide distributed garbage collection of remotely exported objects.
- It is simpler than Common Object Request Broker Architecture (COBRA). RMI can migrate entire objects to a remote host. This is not possible in COBRA. [12]

Disadvantages:

- RMI requires Java installed for the client and server. It does not work with other languages the way COBRA does.

- RMI applications are tightly coupled because of the connection-oriented nature. It is a big challenge to achieve high scalability in such an application model.
- RMI does not support any specific session management support. In a typical client-server implementation, the server has to maintain the session and give information of the multiple clients who access it.
- In an RMI application, both the client and server need access to the latest identical class definition of the objects that a traditional transaction processing or CGI environment does not require.
- RMI mechanism is higher over headed than other techniques. [12]

Comparing RMI to Socket

RMI is a higher level of abstraction while socket is a lower level programming. Developing distributed object-based applications in RMI is much simpler than using sockets. RMI allows a hidden multithreading system, but in socket programming, multithreading has to be implemented in a socket layer. The RMI client can invoke a server method directly while socket level programming only allows data to be passed that must be decoded. An RMI server can be modified or moved to another host without changing the client application. In the server socket mechanism, a client sends a message to the server and waits for the result. A server cannot invoke the method for client. However, the RMI mechanism supports the idea of a call back in which the server invokes methods on the client.

4.3 Java Server Pages (JSPs)

A JSP page is a programming technology that provides a simplified way to create web pages and web applications. It is consists of two types of text: static data and a JSP element. Static data can be expressed in any text-based format such as Hypertext Mark-up Language (HTML), or Extensible Mark-up Language (XML) or other document types. On the other hand, JSP elements construct dynamic content. JSP pages are components in a web or Java EE application which deliver dynamic content to a client in a portable, secure and well-defined way. The code in a JSP page runs on the server that can access data across the whole web application and can use server-side resources such as databases, directories and other application components. When a client such as a user makes a request to the Java application container such as a server, the static page is converted behind the scenes and displayed as dynamic content to the user. JSP technology, created by Sun Microsystems, enables Java code and selects predefined actions to be inserted into static web page content. This code is compiled at runtime for each request made to the page. [13]

Development of JSP

The process of developing a JSP page that can respond to a client request requires three main steps:

- **Creation:** The Java developer builds a JSP source file that consists of HTML and embedded Java code.
- **Deployment:** The JSP is installed into a server using web archives that contain the JSP code, a supporting class, and other necessary files such as HTML and XML files. This can be a full Java server or a stand-alone JSP server.
- **Translation and compilation:** The server requires a JSP container for processing JSP pages. To process all JSP elements in the page, the JSP container first turns the JSP pages into servlet class that runs within a web server and translates the HTML and Java code into a Java code source file that implements the

corresponding dynamic behaviour. The JSP container then compiles the source file into a Java class that is executed by the server. The JSP container initiates the translation and compilation step for a page automatically when it receives a request for the page. Figure 6 illustrates the JSP page translation and processing phases. [13]

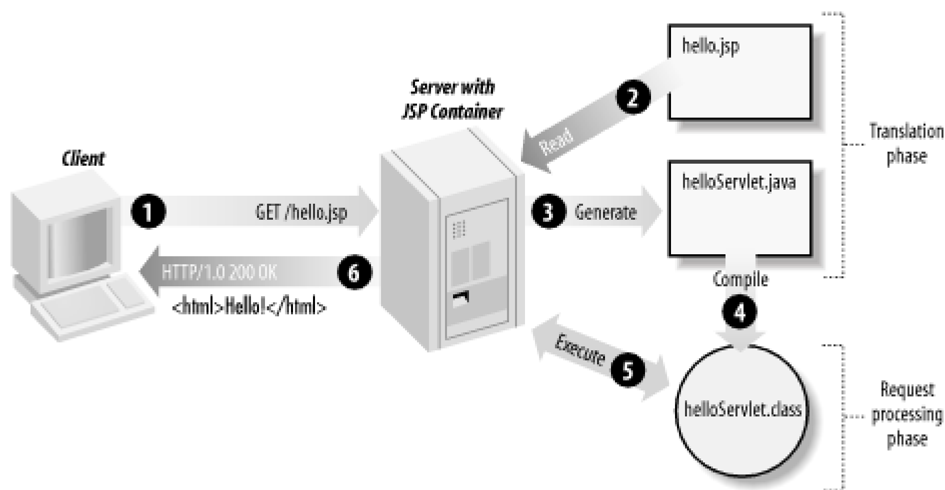

Figure 6. JSP page translation and processing phases

Basic JSP Lifecycle

After completing translation and the compilation step of a JSP page, the JSP lifecycle has the following phases:

Loading and instantiation: The JSP implementation class is created by the server and is loaded into the JVM (Java Virtual Machine) before being used. The default class loader of the JSP container loads this class. Once the class is loaded, the JVM creates an instance of the class. The instantiation of the class can occur immediately after loading or when the first request is made.

Initialization: The JSP container uses the *JspPage* interface which contains the *jspInit* () method to initialize the newly created instance before serving any requests. Typically initialization is performed only once during the entire life cycle of JSP.

Request processing: This phase represents all interactions with a request. After initialization of the new instance, the JSP container invokes the *_jspService ()* method to handle the request and returns a response to the client. Each request normally is executed in a separate thread of execution. The*_jspService ()* method is called once per request, and it takes HttpServletRequest and HttpServletResponse as its parameters. Figure 7 describes the life cycle of a JSP.

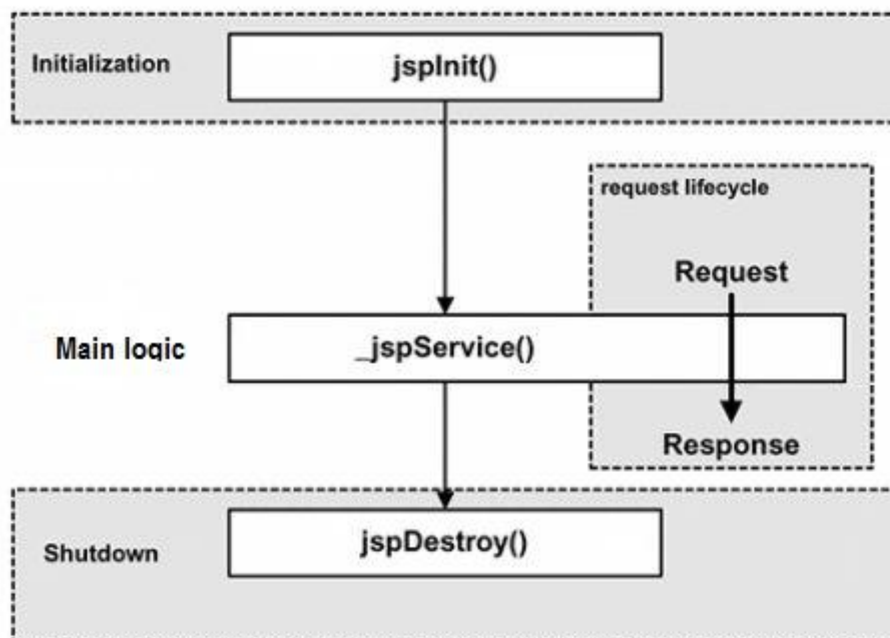

Figure 7: JSP life cycle

End of life: In this phase, the server restrains sending requests to the JSP. After finishing all current requests, any instances of the class are released. The *jspDestroy ()* method is called before the JSP container releases the JSP instance. The server uses this method for several reasons such as shutting down the server, conserving resources, updating the JSP source file, or terminating the instance for other reasons. After executing the jspDestroy *()* method, the JSP instance is marked for garbage collection. [12]

The advantages and disadvantages of the JSP page are the following:

Advantages

- JSP pages can be swiftly developed and easily maintained because they are based on HTML and XML. HTML is easy to understand, and there are many automated tools for dealing with HTML and XML document.
- The dynamic part of JSP is written in Java, so it is more powerful and better suitable for complex applications that require reusable components.
- JSP is portable to other operating systems and servers, and it has an extensive API (Application Programing Interface) for networking, database access, and distributed objects.
- JSP pages are automatically translated and compiled in the Java servlet but are easier to develop than the Java servlet.
- JSP is better because it has a richer set of tools for building external pieces and more options for HTTP responses at which the pieces actually get inserted.
- JSP pages use simplified scripting language based syntax for embedding HTML into JSP. [13]

Disadvantage

- JSP pages are translated to a .java file, and then compiled into Java servlets. Due to the transformation, some errors are very difficult to diagnose.
- JSP pages are translated into .java class files, and the server has to keep those files with JSP pages. This essentially doubles the disk space requirements needed to store JSP pages.

- JSP requires a compiler to compile JSP pages on the server when first accessed. This initial compilation produces a delay when accessing the JSP page for the first time.
- JSP technology locks into a fixed language. Java technology is the only language for JSP. There are no language-independent solutions in JSP technology.
- Database connectivity is not such an easy task in JSP technology. A Java developer has to write a lot of custom code to do the job.

Comparing JSP to RMI

JSPs extend the server side functionality of a web application. JSP pages are converted into servlets. Servlets communicate with other applications on the server and perform tasks above and beyond the normal static data. They can receive a request to get some information from one or more databases, and then convert this data into a static HTML page for the client. RMI is just a way to invoke methods. Using RMI technology, an application can talk to another remote machine and execute different methods. The action performs like a local machine. JSPs are mainly used for any web related activity. Using JSPs, the client only needs to know the web address to display the page and perform some action. In RMI, the client must bind the RMI server to an IP and port number and the client must know this IP and port to communicate with the remote server.

5 Development environments

5.1 Eclipse installation

Eclipse is a popular integrated development environment (IDEs) for developing open platforms and products because of its flexible environment. The Eclipse software development kit (SDK) consists of the Eclipse platform, Java development tools and plug-in development environment. Eclipse IDE was downloaded from the download page of www.eclipse.org. It was a freeware and open source product. The downloaded zip file was unzipped in a known directory and then installed according to integrated instructions. The Java development kit (JDK) 6 and Java runtime environment (JRE) were also installed for good performance of the project.

5.2 Installing Plug-in

Java developers can extend Eclipse abilities by installing plug-ins written for the Eclipse platform and can write and contribute their own plug-in modules. The RMI plug-in is the most comprehensive solution for developing an RMI system. The RMI plug-in was used to create the RMI chat system in this project. The RMI plug-in helped getting started with the Java RMI technology and provided advanced configuration, analysis and debugging tools. RMI plug-in was downloaded from http://www.genady.net /rmi/v20/downloads.html link and was installed with the help of the update manager of Eclipse. The RMI Registry Inspector, a component of the RMI plug-in, displayed and examined the class structure, even invoking remote methods directly from within Eclipse. Figure 8 shows a screenshot of the basic registry inspector interface.

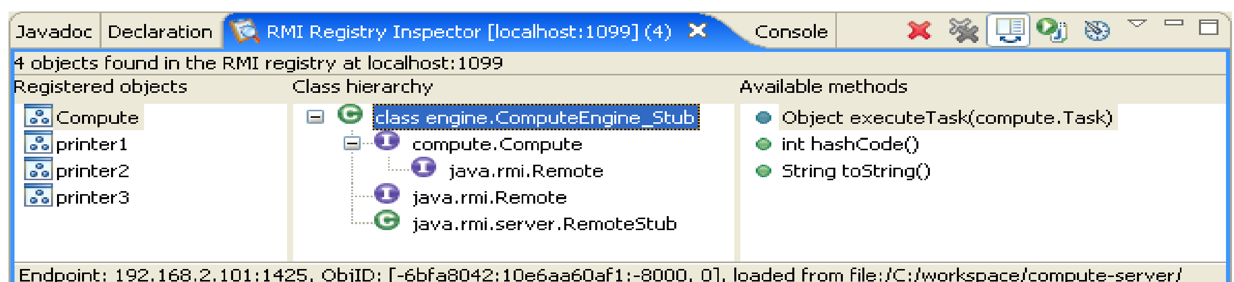

Figure 8. RMI Registry inspector interface

6 Implementation of a Chat Application

6.1 Overview

A chat system is a client-server application on the internet in which users can communicate with each other in real time. The system offers an instantaneous transmission of text-based messages from sender to receiver. It may address point-to-point communications as well as multicast communication from one sender to many receivers. Basically, a client-server application needs the server and client. The server always listens to the client and waits for a task from the client. In my project I used the RMI mechanism to build a simple multicast chat system. A number of message queues were created for the number of users in the server side program. The server always checked queues for tasks that came from users and performed those tasks immediately. Figure 9 illustrates the basic idea of the chat system.

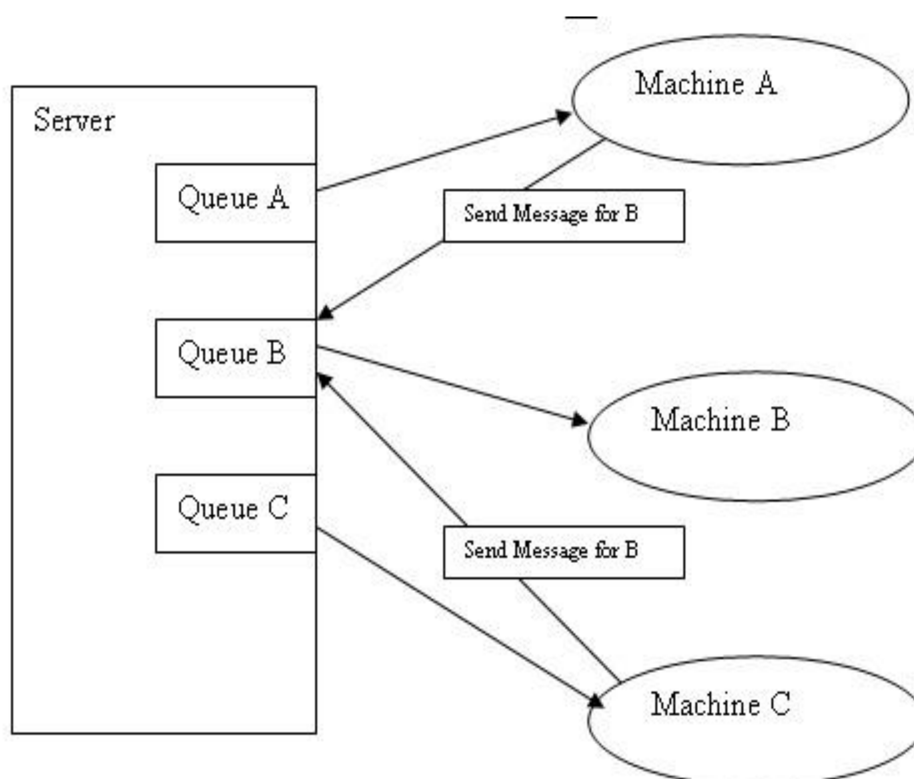

Figure 9: Basic idea of chat system.

6.2 Graphical User Interface (GUI)

A software developer should always keep in mind the user interface when creating an application. In general, the Java platform enables application developers to create GUI applications using Applets and Swing. In this project, the swing application was used to create a user interface for the chat system because it describes both the appearance of the interface and how its components function when they are used. This chat system was designed to have a screen where a user is able to connect to the server via the internet and can send messages to other users who are connected to the server.

When the application starts, the main screen pops up for the user´s viewing. This window consists of four buttons, one list box, three text fields, and one text pane. The four buttons are named *Log in, Broadcast, Send,* and *Clear*. The list box is *User List*, and the three text fields are *Server, User name,* and *Message*. Before connecting to the server, *Log in, Server* and *User name* components are active and other components are disabled. Figure 10 illustrates the window that appears when the application is launched.

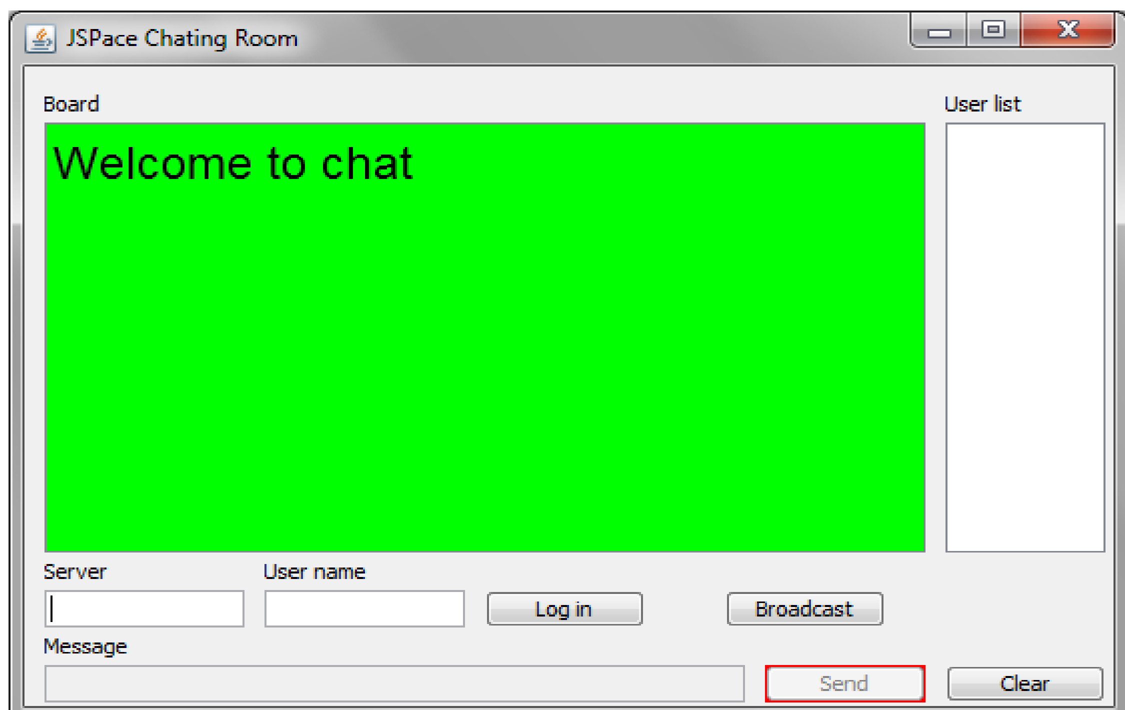

Figure 10. Window before connecting to server

After writing in the text fields *Server* and *User name,* a user clicks on the "*Log in*" button to connect to the server. If the server name is correct and the user name does not match with existing users, the client can log in successfully and the user name will be shown in the list box. In this window, the text field, list box, text pane and all other buttons are active and the colour of text pane is changed automatically. Figure 11 shows the window after it has successfully been connected to the server.

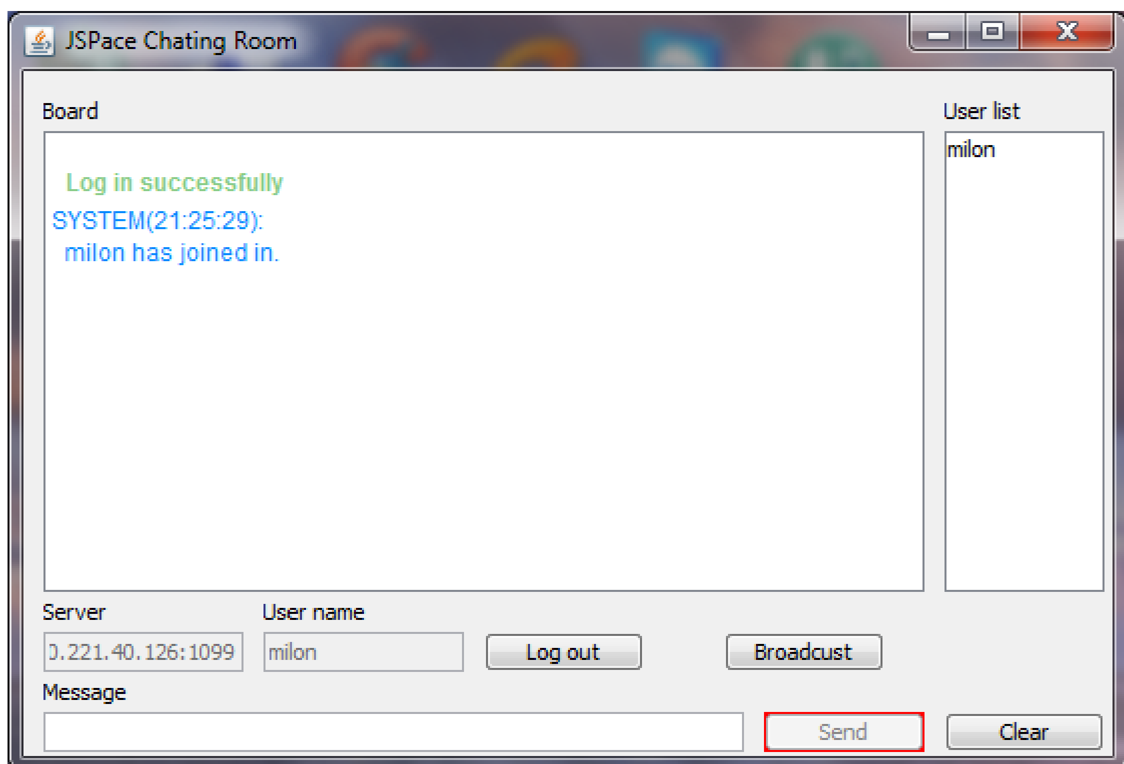

Figure 11. Window after connecting to server

In this stage, the user can write a message in the text field *Message* and click on the *"Send"* button to send a message to other users. After clicking on the *"Send"* button, a message is sent to the server message queue box, and the server checks to whom to send the message, and it performs the task. Technically, the *Log in* and *Send* buttons were implemented in the following way:

```
private class ButtonCallBack implements ActionListener{

	public void actionPerformed(ActionEvent arg0) {
		JButton button = (JButton)arg0.getSource();
```

```
        if (button == btnLogin){
            if (!logged){
                String host = txtServer.getText();
                String userName = txtUserName.getText();
                if (host.equals("")){

    JOptionPane.showMessageDialog(button.getParent(),
            "Please input server address");
                    return;
                    }
                if (userName.equals("")){

    JOptionPane.showMessageDialog(button.getParent(),
                "Please input user name");
                    return;
                    }
                login(host, userName);
                }else{
                    logout();
                }
            }
            if (button == btnSend){
                post();
            }
            if (button == btnClear){
                txtConsole.setText("");
            }
            if (button == btnBroadcust){
                name = null;
            }
        }
    }
```

The message can be sent to a particular user by double clicking on the name of the user listed in the list box. The user can continue sending messages to that user until the sender clicks on the *Broadcast* button. The *Clear* button clears the conversation text on the screen which is shown in the text pane. In this chat system, a maximum of one hundred users are able to chat at the same time. Figure 12 illustrates the multi user chatting system.

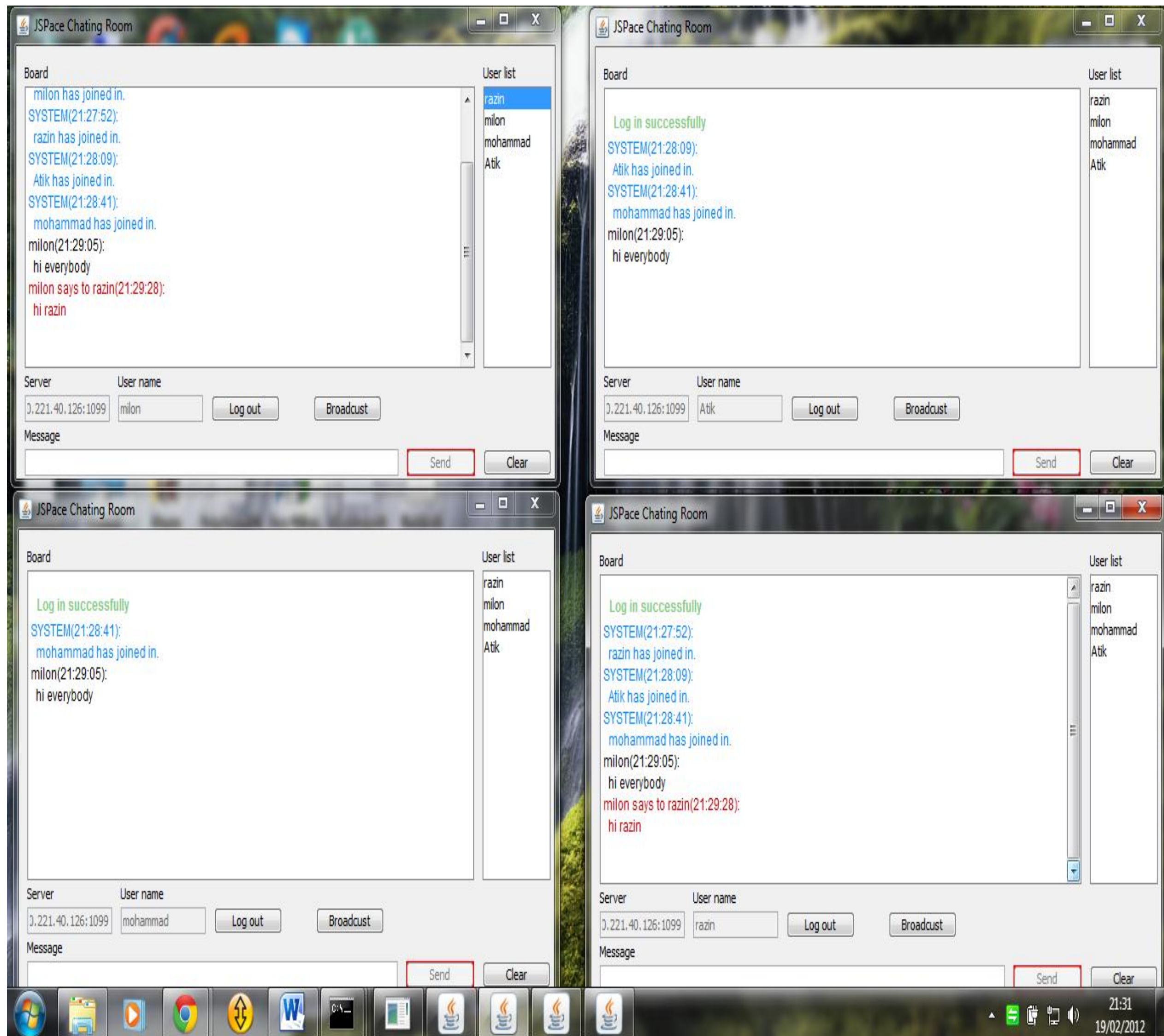

Figure 12. Multi user chat

When a user wants to log off the system, he or she clicks the "*Log out*" button in the window. All other users are notified and they get a message about that the another user logging off in the text pane area.

6.3 Class Diagram

The prototype and the implementation of the system consist of several classes and functions which perform different tasks independently or sometimes in collaboration. The classes of the system, their interrelationship and the methods and attributes of the classes are described in the following figures.

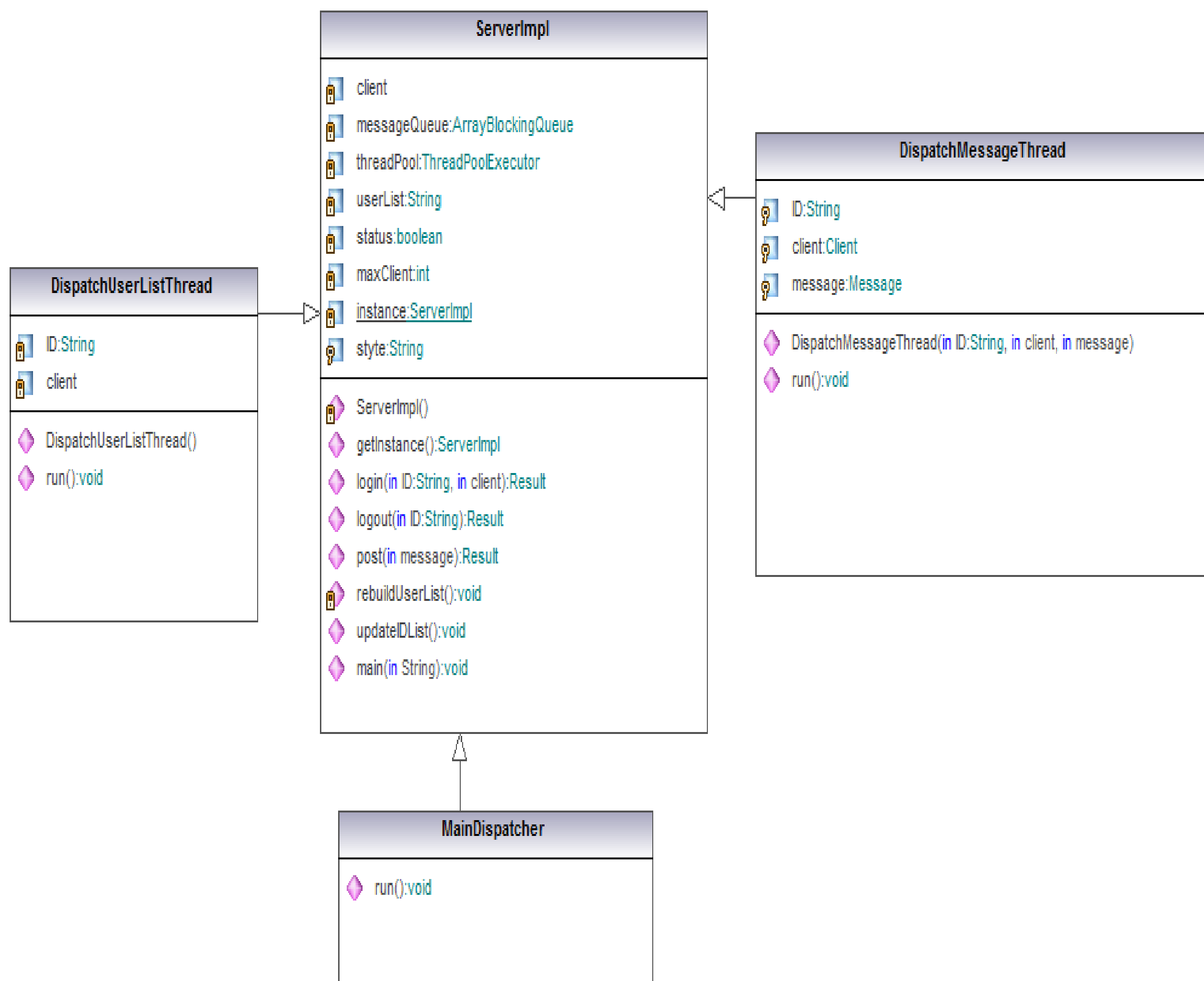

Figure 13. Class diagram 1

Figure 13 describes the attributes and methods of four classes called ServerImpl, MainDispatcher, DispatchUserListThread, and DispatchMessageThread and their interaction with other classes.

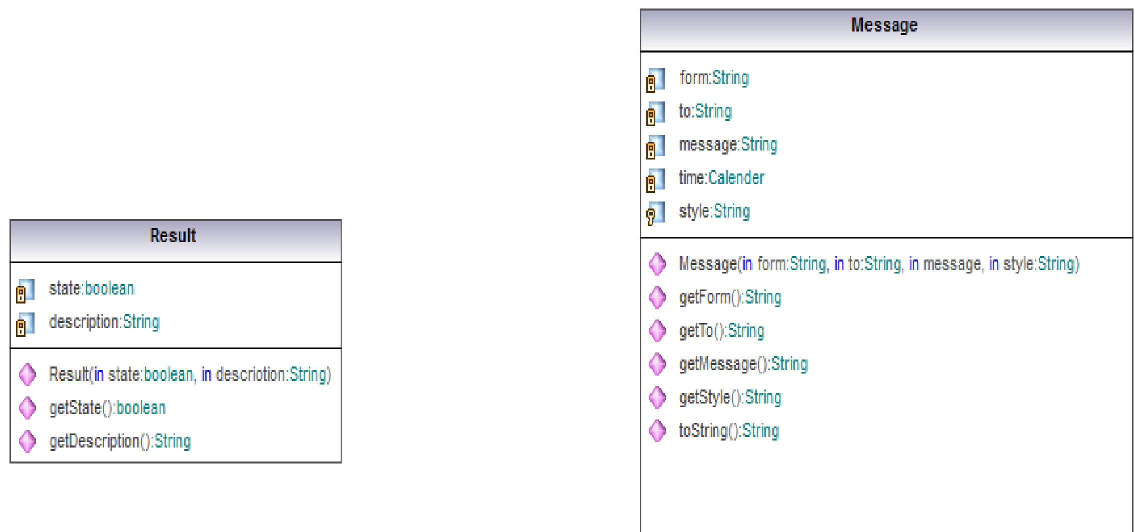

Figure 14. Class diagram 2

Figure 14 illustrates the methods and attributes of two classes called Result and Message as part of the server side program.

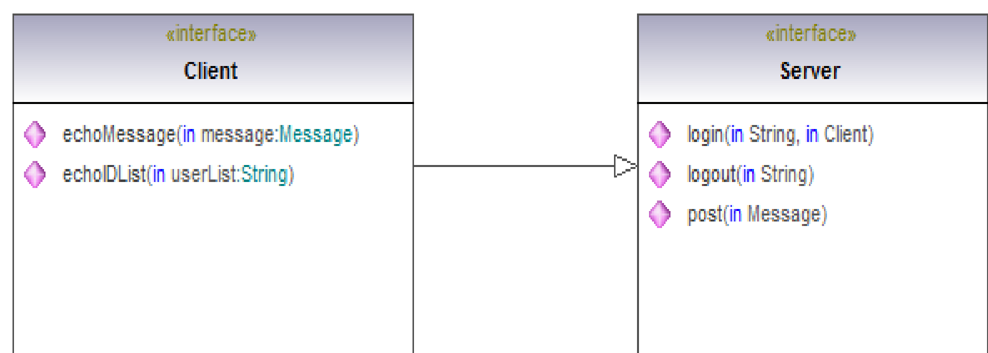

Figure 15. Class diagrams 3

Figure 15 explains the methods of two interfaces called Client and Server as they are common interfaces for the server side and client side.

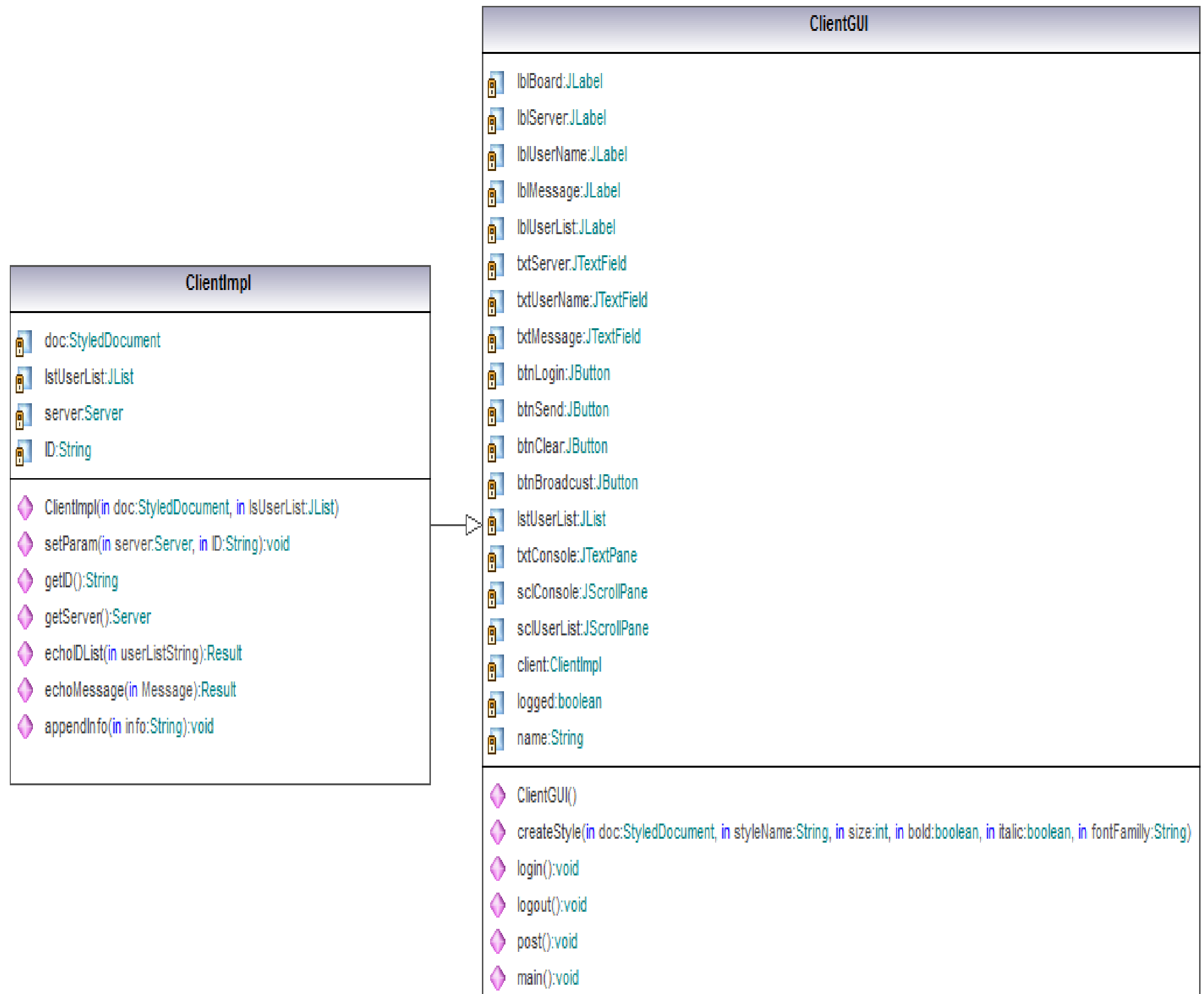

Figure 16. Class diagram 4

Figure 16 describes the methods and attributes of two classes called ClientImpl and ClientGUI on the client side application.

6.4 RMI Registry

The RMI registry acts as a negotiator between the RMI server and the client. Essentially the RMI registry is a place for the server to register the services it offers and a place for the client to query for those services. The RMI Registry is being implemented as a giant hash-table that maps names to remote objects which are always hidden. Each and every remote object has to register with the RMI registry with a unique name. Thus, when a client sends a request, the registry reads the request, looks up the hash table, gets the name of the remote object requested, and returns the stub for the remote object to the client. In a machine, there can be two or more RMI registries, but only one registry can be used per JVM. The RMI registry listens to a specific port. By default, the RMI registry attempts to use 1099 as its service port, but a particular port can be assigned by passing the appropriate argument. The RMI Registry can be launched in two ways: with a stand-alone program or a Java program by accessing the java.rmi.Registry class. Figure 17 shows how to start the RMI registry.

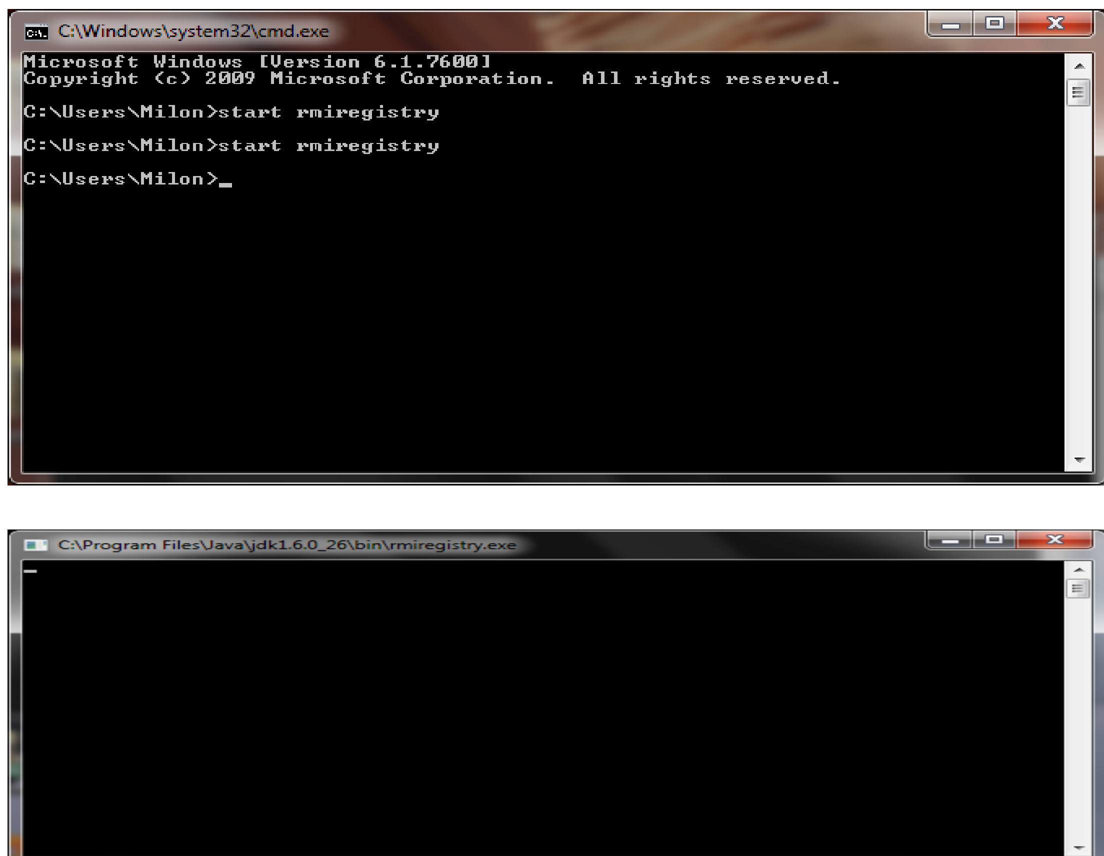

Figure 17. RMI registry

6.5 GUI for Mobile client

The Java platform enables the application developer to create a GUI easily using Java 2 Platform Micro Edition (J2ME). J2ME is a technology that allows software developers to use Java language and related tools to create programs for mobile devices such as a cellular phone. J2ME consists of a programming specification and a special K virtual machine (KVM) that allows a small program to run in mobile devices. The 'K' in KVM stands for kilobyte, signifying that the KVM runs in kilobytes of memory as opposed to megabytes. Two programming specifications for a cellular phone are the Connected Limited Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP). [16] Figure 18 shows the relations between CLDC, MIDP and native software.

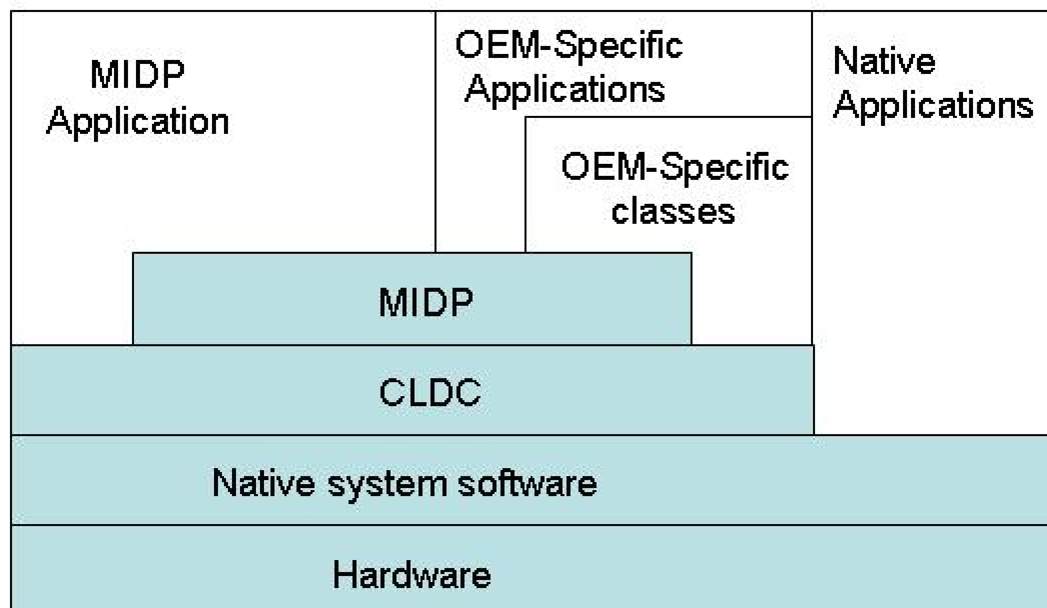

Figure 18. Relations among different layers in J2ME

CLDC lays out the application program interface and virtual machine features needed to support mobile devices. MIDP adds the user interface, networking, and messaging details to the CLDC. MIDP supports the idea of MIDlet which is a small program similar to an applet but one that conforms to CLDC and MIDP and that is intended for mobile devices.

An MIDP application, MIDlet, must extend the abstract MIDlet class, which consists of the abstract methods startApp(), pauseApp() and destroyApp(). These methods also

have to be written into our MIDlet. The application management software (AMS) can create, start, pause, and destroy a MIDlet using a Midlet derived class.

Application management software (AMS) creates the MIDlet object by allocating memory for it. The MIDlet instance is the first in the **paused** state. If an exception occurs during the creation phase of the MIDlet object, the application enters the **destroyed** state immediately. After the exceptionless return from constructor, AMS calls the startApp() method of the MIDlet. The application enters just before the startApp() method calls the **active** state. The application has most of the initialization tasks in the startApp() method. Figure 19 illustrates MIDlet state of the AMS.

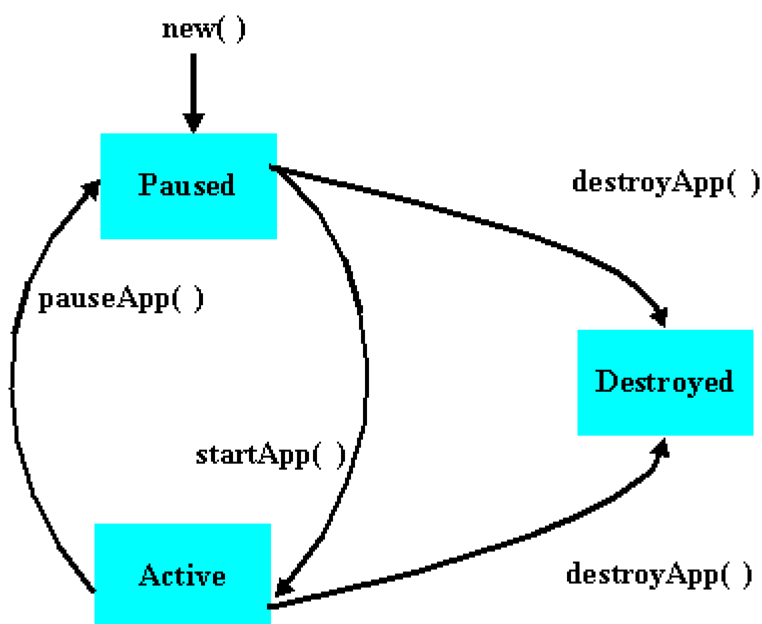

Figure 19. Midlet state

The simple mobile chat client is designed to have several screens where a user can switch from one screen to another according to tasks.

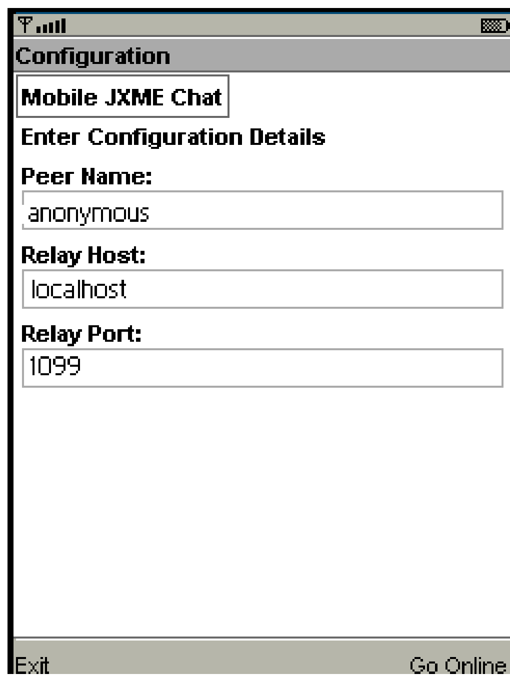

Figure 20. Chat configuration screen

Figure 20 illustrates how the client can setup the variable that will help to connect to the server. The configuration screen consists of three text fields where the client can input the configuration variable and two buttons named Exit and Go Online.

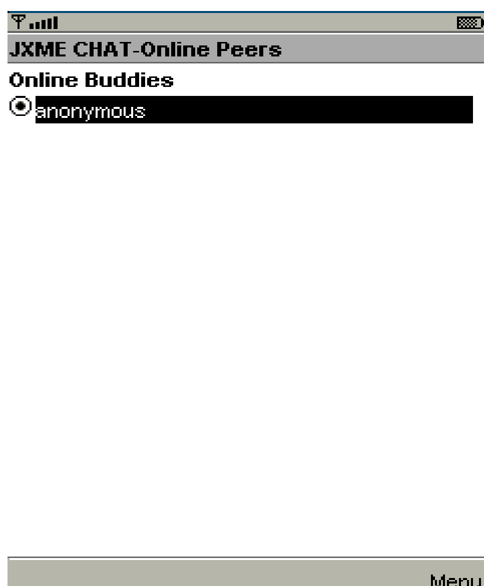

Figure 21. Online presence screen

Figure 21 shows which clients are connected to the server. This screen consists of a list where all user´s names can be displayed. On devices that have a dedicated hardware “Select” or “Go” key, the select operation is implemented with the key. The user can choose other users for chatting by pressing that key.

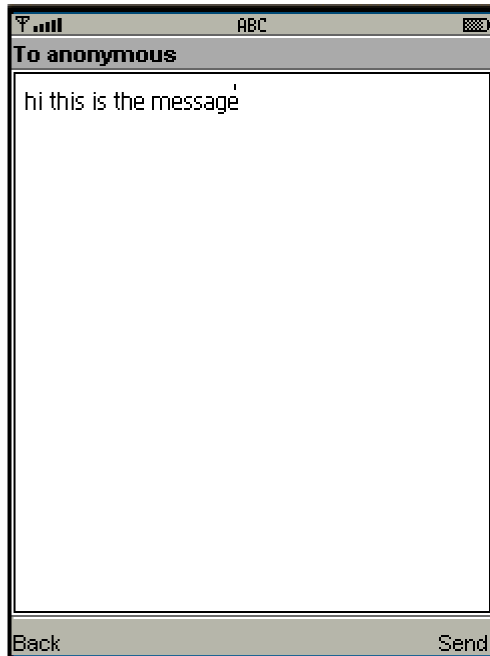

Figure 22. Chat screen

In figure 22, the chat screen allows the user to input a message to be sent to another user. This screen consists of a text box. The text box allows the user to enter and edit text for sending messages to other users by selecting “Send” key.

7 Analysis of Results

7.1 Desktop chat

I designed and implemented a chat application based on RMI technology which allows sending and receiving instant messaging in a network. The application client side and server side were implemented using Java language and Eclipse IDE. The RMI is a full grown mechanism for distributed computing, and is it scalable to more complex tasks than request response style of programming. When a client calls on an object that resides on a remote machine, it must have a reference to it. The client has to know more about the object to retrieve and use it. The only way to do this is to embed that lookup information in the client code. I coded this lookup information into the client as follows:

```
String url = "rmi://" + host + "/JSPaceChating";
Server server = (Server)Naming.lookup(url);
```

When the lookup succeeds, the RMI server returns the remote object´s stub for the remote object´s class and methods. Then the stub sends the request to the skeleton which is on the server side. The skeleton retrieves the operations of the client´s interested object and maintains a dispatch() method to coordinate the object´s response back through the stub. This process occurs internally in the RMI mechanism. At the beginning, it was difficult for me to look up the server and I got an exception called *java.security.AccessControlException*, but after difficulties, I succeeded in binding client and server. Initially, I could not bind the client and server because of a security restriction. RMI applications have no permission to the internet by default. Therefore, I added the following code in the security policy file.

```
grant codeBase  {
     permission java.security.AllPermission;
};
```

It is possible to generate skeleton and stub classes that are used to manage the communication endpoints between the client and server. The tool named **rmic** takes the implementing class and produces class files using the same name. The server code that sponsors the request object is responsible for binding it to an RMI server. This

service is launched through a tool bundle with JDK called **rmiregistry**. I started the RMI registry by calling the following code in command prompt.

Start rmiregistry

The client was able to write a message in the message text field from the client GUI. After clicking on the *send* button, the message was successfully sent through the unicast pipe. The message has four elements: the name of the sender, the message content, the style of the message, and the name of the recipient. This part was first tested on the emulator and the message was sent successfully through the socket.

I used a Windows server machine as a test server provided by Amazon EC2. I made an account and created a Windows server in this service provider. Then I started the server in its own window and telneted to it through my computer. I configured all TCP/IP stacks and activated derivers among the list of network. All the processes went smoothly but when I tried to connect the client to the remote server, unfortunately I got the following exception:

`java.net.ConnectException`*java.net.ConnectException: Connection timed out*

Finally, it was discovered that the client did not find the right hostname. Therefore, I edited the ../etc/hosts file and added a line, <107.21.222.137 hostname>, to that file. After that the client was able to connect to the server and the application worked well.

7.2 Mobile Chat

I also designed and implemented a very simple chat client application which allows sending and receiving messages by using mobiles. The objective of implementing this chat client was to demonstrate how easily users can communicate with other users on the network by using a mobile device. The chat client application was developed with the help of J2ME technology. The operation of sending and receiving messages was straightforward. The client connects to the server directly. It does not need to connect RMI registry such as RMI technology.

The client connects to the server using a Hypertext Transfer Protocol (HTTP) URL. HTTP is an application protocol for a distributed, hypermedia information system. HTTP contains the rules by which clients and servers exchange information. When a client initiates a request to a server by opening a TCP/IP connection, the request consists of a request line, a set of request headers and an entity. Then the server sends a response that consists of a status line, a set of response headers, and an entity. When I sent a request to the test server, the actual HTTP request was the following:

```
 GET / HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (compatible; Opera/3.0; Windows 95/NT4
)

 Accept: text/*, image/jpeg, image/png, image/*, */*

Accept-Encoding: x-gzip, gzip, identity
Accept-Charset: Any, utf-8, *
Accept-Language: en, en_US
Host: ec2-107-21-222-137.compute-1.amazonaws.com
```

I got the following response from the server.

```
HTTP/1.1 200 OK
Date: Thu, 24 Jan 2012 17:33:52 GMT
Server: Apache/1.3.14
Last-Modified: Mon, 21 Jan 2012 22:08:33 GMT
Etag: “47bc6-25e0-3c4c9161”
Accept-Ranges: bytes
Content-Length: 9696
Connection: close
Content-Type: text/html
```

8 Conclusion

The objectives of this project were to clarify and briefly discuss all methods of the Java platform that support networking communication and to design and implement a multi-client chat application for desktop computers and mobile systems which allows instant messaging by using the internet. Java was developed initially as a language that would control a network of interactive consumer devices. Connecting devices was the main purpose when the application was designed.

The Java package makes it possible to communicate over a network, providing cross-platform abstraction to make connections using common protocols. Java provides various methods for communicating over the internet such as socket and server socket, RMI, JSP and servlet. The socket class provides a client-sided socket interface similar to a standard UNIX socket and this opens the connection. A server socket works similarly to a client socket, with the exception of the accept() method. The RMI mechanism binds the client and server together and invokes methods from the client to the server or the server to the client. The process of making JSP on the internet for communication is very simple and it enables to mix regular, static HTML with dynamically generated content from the servlet.

The RMI mechanism was used to design and implement a desktop chat application. Further J2ME MIDlet technology was used to create for a mobile chat client. This application did not have many features such as being secure which in real case would be one of the top priorities. To extend this application in the future, security features would need to be implemented. In this project it was assumed that it is hard to debug a RMI distributed application, particularly when the client and server are multithreaded. In general, the RMI mechanism cannot guarantee that a client can always use the same thread in consecutive calls. The RMI application needs deep knowledge and better understanding for design and implementation. Peer-to-peer technology and its mobile counterpart have a very big potential in changing this.

References

1 Beal V. Internet-based communications [online]. Webopedia; 9 January 2009. URL: http://www.webopedia.com/DidYouKnow/Internet/2009/ internet_communications.asp. Accessed 6 August 2011.

2 Dylan B. E-mail, instant messaging and chat rooms: The new letter? [online]. 6 March 2008.
URL: http://www.123helpme.com/view.asp?id=25790. Accessed 6 August 2011.

3 Karthikeyan M. Client server architecture [online]. Exforsys Inc; 30 June 2008. URL: http://www.exforsys.com/tutorials/client-server/client-server-architecture.html. Accessed 8 August 2011.

4 Stephanie D. Disadvantage of client server architecture [online]. Demand Media; 16 July 2010.
URL: http://www.ehow.com/list_6498527_disadvantages-client-server-architecture.html. Accessed 8 August 2011.

5 Schildt H. The complete reference. 7th ed. New York, NY: McGraw-Hill; 2006.

6 Oaks S, Wong H. Java threads. 3rd ed. Sebastopol, CA: O´Reilly Media; 2004.

7 Cadenhead R, Lemay L. Java[TM] 6. 5th ed. Indianapolis, Indiana: Sams Publishing; 2007.

8 Miller R, Williams A. Java sockets 101 [online]. IBM; 30 August 2009.
URL: http://www.ibm.com/developerworks/java/tutorials/j-sockets/section2.html. Accessed 10 October 2011.

9 An overview of RMI applications [online]. Oracle.
URL: http://download.oracle.com/javase/tutorial/rmi/overview.html. Accessed 11 October 2011.

10 jGuru: remote method invocation (RMI) [online]. Oracle.
URL: http://java.sun.com/developer/onlineTraining/rmi/RMI.html#IntroRMI. Accessed 11 October 2011.

11 Venjaramoodu M. Online crimefile management [online]. Scribd Inc; 15 March 2011.
URL: http://www.scribd.com/doc/50787543/42/RMI-Architecture. Accessed 11 October 2011.

12 Mukhar K, Zelenak C, Weaver J, Crume J. Beginning Java EE 5 platform. Berkeley, CA: Apress Inc; 2006.

13 Hall M, Brown L. Core servlets and Java server pages. 2nd ed. Santa Clara, CA: Sun Microsystem Inc; 2004.

14 Raymond B, Erich M. A study of Internet messing [online]. IEEE Network; August 2006.
URL:https://www.gprt.ufpe.br/~rafael.antonello/articles/analysis. Accessed 6 August 2011.

15 Merritt E. Client server relationship [online]. SDGIS; 15 September 2011.
URL: http://evarigisconsulting.wordpress.com/. Accessed 4 November 2011.

16 Mobile game development [online]. KidsOnMobile.
URL: http://library.thinkquest.org/06aug/01303/english/glossary.html.
Accessed 7 February 2012.

Server Implementation

```
package chat_server;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;
import java.util.concurrent.*;

public class ServerImpl extends UnicastRemoteObject implements Server {

	/**
	 *
	 */
	private static final long serialVersionUID = 1L;
	private HashMap<String, Client> clientList;
	private ArrayBlockingQueue<Message> messageQueue;
	private ThreadPoolExecutor threadPool;
	private String[] userList;
	private boolean  status;
	private final int maxClient = 100;
	private static ServerImpl instance = null;



	private ServerImpl() throws RemoteException {
		super();
		Registry registry =  LocateRegistry.getRegistry();
		registry.rebind("JSPaceChating", this);
		System.out.println("server is ready");
		clientList = new HashMap<String, Client>();
		messageQueue = new ArrayBlockingQueue<Message>(100);
		threadPool = new ThreadPoolExecutor(maxClient / 2, maxClient, 10,
TimeUnit.SECONDS,
						new ArrayBlockingQueue<Runnable>(10), new
ThreadPoolExecutor.DiscardOldestPolicy());
		status = true;
		new Thread(new MainDispatcher()).start();
	}

	public static ServerImpl getInstance(){
		try{
			if (instance == null){
				instance = new ServerImpl();
			}
			return instance;
		}catch(RemoteException re){
			re.printStackTrace();
			return null;
		}
	}
```

```
	public Result login(String ID, Client client){

		if (clientList.size() >= maxClient){
			return new Result(false, "Log in failed, this chating room is already
full");
		}

		if (clientList.get(ID) == null && !ID.equalsIgnoreCase("system")){
			clientList.put(ID, client);
			updateIDList();
			post(new Message("SYSTEM", null, ID + " has joined in.",
"SYSTEM"));
			return new Result(true, "Log in successfully");
		}
		else{
			return new Result(false, "Log in failed, this user name already
exists");
		}
	}

	public Result logout(String ID){
		clientList.remove(ID);
		updateIDList();
		post(new Message("SYSTEM", null, ID + " has left.", "SYSTEM"));
		return new Result(true, "Log out successfully");
	}

	public Result post(Message message) {
		if (message.getTo() != null){
			if (clientList.get(message.getTo()) == null){
				return new Result(false, "User \"" + message.getTo() + "\"
was not found");
			}
			if (message.getTo().equals(message.getFrom())){
				return new Result(false, "Not to talk to yourself");
			}
		}
		// Block if the message queue is full
		try{
			while(messageQueue.remainingCapacity() == 0){
				Thread.sleep(100);
			}
			// Enqueue the message
			messageQueue.put(message);
		}catch(InterruptedException ie){
			ie.printStackTrace();
		}
		return new Result(true, "Post successfully");
	}

	private void rebuildUserList(){
		Set<String> ID = clientList.keySet();
		userList = new String[ID.size()];
```

```
		int i = 0;
		for (Iterator it = ID.iterator(); it.hasNext(); i++){
			userList[i] = (String)it.next();
		}

	}

	private void updateIDList(){
		rebuildUserList();
		Set<String> IDs = clientList.keySet();
		for (Iterator it = IDs.iterator(); it.hasNext();){
			String ID = (String)it.next();
			threadPool.execute(new DispatchUserListThread(ID,
clientList.get(ID)));
		}
	}

	public static void main(String[] args) {
		ServerImpl.getInstance();
	}

	private class MainDispatcher implements Runnable{
		public void run(){
			try{
				while(status){
					if (messageQueue.isEmpty()){
						Thread.sleep(100);
					}

					Message message = messageQueue.take();
					if (message.getTo() != null){// A whisper
						threadPool.execute(new
DispatchMessageThread(message.getTo(), clientList.get(message.getTo()), message));
						threadPool.execute(new
DispatchMessageThread(message.getFrom(), clientList.get(message.getFrom()), message));
					}else{// A broadcast
						for(Iterator i = clientList.keySet().iterator();
i.hasNext();){
							String to = (String)i.next();
							threadPool.execute(new
DispatchMessageThread(to, clientList.get(to), message));

						}
					}
				}
			}catch(InterruptedException ie){
				ie.printStackTrace();
			}
		}
	}
```

```
	private class DispatchMessageThread implements Runnable{
		private String ID;
		private Client client;
		private Message message;

		public DispatchMessageThread(String ID, Client client, Message message){
			this.ID = ID;
			this.client = client;
			this.message = message;
		}

		public void run(){
			try{
				client.echoMessage(message);
			}catch(RemoteException re){
				// Assuming the client is shut
				//re.printStackTrace();
				logout(ID);
			}
		}
	}
	private class DispatchUserListThread implements Runnable{
		private String ID;
		private Client client;

		public DispatchUserListThread(String ID, Client client){
			this.ID = ID;
			this.client = client;
		}

		public void run(){
			try{
				client.echoIDList(userList);
			}catch(RemoteException re){
				// Assuming the client is shut
				//re.printStackTrace();
				logout(ID);
			}
		}
	}
}
```

Graphical User Interface for Client

```
package chat_client;

import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;

import chat_server.Message;
import chat_server.Result;
import chat_server.Server;

public class ClientGUI extends JFrame{

	private JLabel lblBoard, lblServer, lblUserName, lblMessage, lblUserList;
	private JTextField txtServer, txtUserName, txtMessage;
	private JButton btnLogin, btnSend, btnClear, btnBroadcust;
	private JList lstUserList;
	private JTextPane txtConsole;
	private JScrollPane sclConsole, sclUserList;
	private ClientImpl client;
	private boolean logged;
	private String name ;

	public ClientGUI() throws RemoteException{
		super("JSPace Chating Room");


		lblBoard	= new JLabel("Board");
		lblServer	= new JLabel("Server");
		lblUserName	= new JLabel("User name");
		lblMessage	= new JLabel("Message");
		lblUserList	= new JLabel("User list");
		txtServer	= new JTextField();
		txtUserName	= new JTextField();
		txtMessage	= new JTextField();
		btnLogin	= new JButton("Log in");
		btnSend			= new JButton("Send");
		lstUserList	= new JList();
		btnClear	= new JButton("Clear");
		btnBroadcust = new JButton ("Broadcast");
		txtConsole	= new JTextPane();
		sclConsole	= new JScrollPane(txtConsole);

		sclUserList = new JScrollPane(lstUserList);
		client		= new
ClientImpl((StyledDocument)txtConsole.getDocument(), lstUserList);
		logged		= false;
```

```
ButtonCallBack btnCallBack = new ButtonCallBack();
btnLogin.addActionListener(btnCallBack);
btnSend.addActionListener(btnCallBack);
btnClear.addActionListener(btnCallBack);
btnBroadcust.addActionListener(btnCallBack);
KeyboardCallBack keyboardCallBack = new KeyboardCallBack();
txtServer.addKeyListener(keyboardCallBack);
txtUserName.addKeyListener(keyboardCallBack);
txtMessage.addKeyListener(keyboardCallBack);
txtMessage.getDocument().addDocumentListener(new DocumentCallBack());

lstUserList.addMouseListener(new MouseCallBack());

setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
addComponentListener(new ComponentCallBack());
addWindowListener(new WindowCallBack());

txtMessage.setEnabled(false);
btnSend.setEnabled(false);
btnSend.setBackground(Color.RED);
lblBoard.setBackground(Color.GREEN);

txtConsole.setEditable(false);
txtConsole.setBackground(Color.GREEN);

txtConsole.setFont(new Font("å®‹ä½“", Font.PLAIN, 24));
txtConsole.setText("Welcome to chat");
StyledDocument doc = txtConsole.getStyledDocument();
createStyle(doc, "SYSTEM", 12, false, false, new Color(0x00, 0x88, 0xFF),
"å®‹ä½“");
createStyle(doc, "SPEACH", 12, false, false, Color.BLACK, "å®‹ä½“");
createStyle(doc, "WHISPER", 12, false, false, new Color(0xCC, 0x00, 0x00),
"å®‹ä½“");
createStyle(doc, "INFO", 12, true, false, new Color(0x88, 0xCC, 0x88),
"å®‹ä½“");

setLayout(null);
add(lblBoard);
add(lblServer);
add(lblUserName);
add(lblMessage);
add(lblUserList);
add(txtServer);
add(txtUserName);
add(txtMessage);
add(btnLogin);
add(btnSend);
add(btnClear);
add(btnBroadcust);
add(sclUserList);
```

```
        add(sclConsole);
        setPreferredSize(new Dimension(560, 380));
        pack();

        txtServer.requestFocus();

        setVisible(true);
}



private void login(String host, String ID){
        try{
                System.setSecurityManager(new RMISecurityManager());
                String url = "rmi://" + host + "/JSPaceChating";
                Server server = (Server)Naming.lookup(url);
                System.out.println("server name: " + url);
                client.setParam(server, ID);
                Result rs = server.login(ID, client);
                if (rs.getState()){
                        txtServer.setEnabled(false);
                        txtUserName.setEnabled(false);
                        btnLogin.setText("Log out");
                        txtMessage.setEnabled(true);
                        txtMessage.requestFocus();
                        txtConsole.setBackground(Color.WHITE);
                        txtConsole.setText(" ");
                        logged = true;
                }
                client.appendInfo(rs.getDescription() + "\n");
        }catch(Exception e){
                e.printStackTrace();
                JOptionPane.showMessageDialog(this, "Connection to server failed");
        }
}

private void logout(){
        if (!logged){
                return;
        }
        try{
                Result rs = client.getServer().logout(client.getID());
                if (rs.getState()){
                        txtServer.setEnabled(true);
                        txtUserName.setEnabled(true);
                        btnLogin.setText("Log in");
                        txtMessage.setEnabled(false);
                        lstUserList.removeAll();
                        txtServer.requestFocus();
                        logged = false;
                }
        }catch(Exception e){
```

```
			e.printStackTrace();
			JOptionPane.showMessageDialog(this, "Log out failed");
		}
	}

	private void post(){
		try{
			String[] txtMsg = txtMessage.getText().split(" ");
			String from = client.getID();
			String to = null;
			String msg = "";
			String style = null;
			int msgIndex = 0;
			if (txtMsg[0].equals("/to")){
				to = txtMsg[1];
				msgIndex = 2;
				style = "WHISPER";
			}
			else{
				style = "SPEACH";
				to = name;
			}


			for (int i = msgIndex; i < txtMsg.length; i++){
				msg += txtMsg[i] + " ";
			}

			Message message = new Message(from, to, msg, style);
			Result rs = client.getServer().post(message);
			if (!rs.getState()){
				client.appendInfo(rs.getDescription() + "\n");
			}else{
				txtMessage.setText("");
				btnSend.setEnabled(false);
			}

			//client.echoMessage(message);
		}catch(RemoteException re){
			re.printStackTrace();
		}
	}

	public static void main(String args[]){
		try{
	UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
			new ClientGUI();
		}catch(Exception e){}
```

Client Implementation

```
package chat_client;
import java.rmi.*;
import java.rmi.server.*;

import javax.swing.*;
import javax.swing.text.*;

import chat_server.Client;
import chat_server.Message;
import chat_server.Result;
import chat_server.Server;

public class ClientImpl extends UnicastRemoteObject implements Client{

	private StyledDocument doc;
	private JList lstUserList;
	private Server server;
	private String ID;

	public ClientImpl(StyledDocument doc, JList lstUserList) throws RemoteException {
		super();
		this.doc = doc;
		this.lstUserList = lstUserList;
	}

	public void setParam(Server server, String ID){
		this.server = server;
		this.ID = ID;
	}

	public String getID(){
		return ID;
	}

	public Server getServer(){
		return server;
	}

	public Result echoIDList(String[] userList) throws RemoteException {
		lstUserList.setListData(userList);
		return null;
	}

	public Result echoMessage(Message message) throws RemoteException {
		try {
			doc.insertString(doc.getLength(), message.toString(),
doc.getStyle(message.getStyle()));
		} catch (BadLocationException e) {
			e.printStackTrace();
```

```
		}
		return null;
	}

	public void appendInfo(String info){
		try {
			doc.insertString(doc.getLength(), info, doc.getStyle("INFO"));
		} catch (BadLocationException e) {
			e.printStackTrace();
		}
	}


}
```