**1. SYNOPSIS**

Chatting is a method of using technology to bring people and ideas “together” despite of the geographical barriers. The technology has been available for years but the acceptance it was quite recent. Our project is an example of a chat server.

It is made up of 2 applications the client application, which runs on the user’s Pc and server application, which runs on any Pc on the network. To start chatting client should get connected to server where they can practice two kinds of chatting, public one (message is broadcasted to all connected users) and private one (between any 2 users or clients only)

**2. PREAMBLE**

**2.1 General Introduction**

Communication over a network is one field where this system finds a wide ranging application. Chat application establishes a connection between 2 or more systems connected over an intranet or ad-hoc. This application can be used for large scale communication and conferencing in an organization or campus of vast size, thus increasing the standard of cooperation. In addition, it converts the complex concept of sockets to a user friendly environment. This software can have further potentials, such as file transfer and voice chatting option which can integrated in the system later

**2.2 Statement of Problem**

The client-server communication model is used in a wide variety of software applications. Where normally the server side is sufficiently protected and sealed from public access, but client applications running on devices like notebooks and desktops are considered insecure and exposed to security threats.

The main weakness of client-server chat application is that there is no security provided to data which is transferred between clients. Any unauthorized client can hack the client account and can change the data. This is the main objective of this project is to develop a secured Client-Server Chat Application where Messages shared between users should be encrypted to maintain privacy.

**3. REVIEW OF LITERATURE**

**3.1 Project Description**

Several network systems are built to communicate with one another and are made available through service-oriented architectures. In this project, we use the client server architecture to develop a secured Client-Server chat application. A chat application is created based on Transmission Control Protocol (TCP) where TCP is connection oriented protocol and in the end, multithreading is used to develop the application

A client-server chat application consists of a Client Module and a Server Module and there exists a two-way communication between them. Here, Message Processor is used to interpret message from the user, Message Interpreter is used to extract and pass the received message. Message Maker is used to construct back the message and Client Manager is used to maintain the clients list which the sender and receiver at both sides use to interact with each other.

In general, the server process will start on some computer system; in fact, the server should be executed before the client. Server usually initializes itself, and then goes to wait state or sleep state where it will wait for a client request. After that, a client process can start on either the same machine or on some other machine. Whenever the client wants some service from the server, it will send a request to the server and the server will accept the request and process it. After the server has finished providing its service to the client, the server will again go back to sleep, that is, waiting for the next client request to arrive. This process is repeated as long as the server processes is running. Whenever such request comes, the server can immediately serve the client and again go back to the waiting state for the next request to arrive.

**3.2 Modules Included in The Project**

A software module is an independent and deployable software component of a larger system that interacts with other modules and hides its inner implementation. It has an interface that allows inter-modular communication. The interface defines which components it provides for external use and which components it requires for internal use. Module determines a boundary by specifying which part of the source code is inside the module. It also provides flexibility and increases the reusability of the software system. Our project will be having three modules which are: Server Module, Client Module and Admin Module

**3.2.1 Server Module**

A server Module is a program which runs on a specific computer and has a socket that is bound to a specific port. The server waits and listens to the socket for a client to make a connection request. When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

**3.2.2 Client Module**

Client Module is used to implement reliable, bidirectional, persistent, point to point stream-

based connection between hosts on the internet. A socket can also be used to connect java’s I/O system to other programs that may reside either on the local machine or on another machine on the internet

The Socket class is designed to connect to server sockets and initiate protocol exchanges, the creation of a socket object implicitly establishes a connection between the client and server

**3.2.3 Administration Module**

**3.3 Project Modules detailed description**

The client–server model is a standard model for network applications. A server is a process that is continuously running and waiting to be contacted by a client process. A client process initiates contact with the server by connecting to it at a specified port

**3.3.1 Socket Overview**

**A NETWORK SOCKET** is a lot like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can plug in to the socket and communicate. With electrical sockets, it doesn’t matter if you plug in a lamp or a toaster; as long as they are expecting 60Hz, 115-volt electricity, the devices will work.

**INTERNET PROTOCOL (IP)** is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.

**TRANSMISSION CONTROL PROTOCOL (TCP)** is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit your data. A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

There are two kinds of sockets: server sockets and client sockets.

* A server socket waits for requests from clients.
* A client socket can be used to send and receive data

**PORT** a server socket listens at a specific port. port is positive integer less than or equal to 65565. The port number is necessary to distinguish different server applications running on the same host. Ports 1 through 1023 are reserved for administrative purposes (e.g. 21 for FTP, 23 for Telnet, 25 for e-mail, and 80 for HTTP)

**3.3.2 Server Socket**

The Server Socket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. Server Sockets are quite different from normal Sockets. When you create a Server Socket, it will register itself with the system as having an interest in client connections. Server Socket can be created with the following constructors.

ServerSocket(int port): Creates server socket on the specified port with a queue length of 50 ServerSocket(int port, int maxQueue): Creates a server socket on the specified port with maximum queue length of maxQueue.

**PORT:** Port number at which the server will be listening for requests from clients

**MAXQUEUE:** the maximum length of the queue of clients waiting to be processed default is 50

**ServerSocket** has a method called accept( ), which waits for a connection request. The thread that executes the method will be blocked until a request is received, at which time the method return a client socket. The ServerSocket also has a method called close() which stops waiting from requests from client.

try {

ServerSocket s = new ServerSocket(port); while (true) {

Socket incoming = s.accept();

«Handle a client» incoming.close();

}

s.close();

} catch (IOException e) {

«Handle exception»

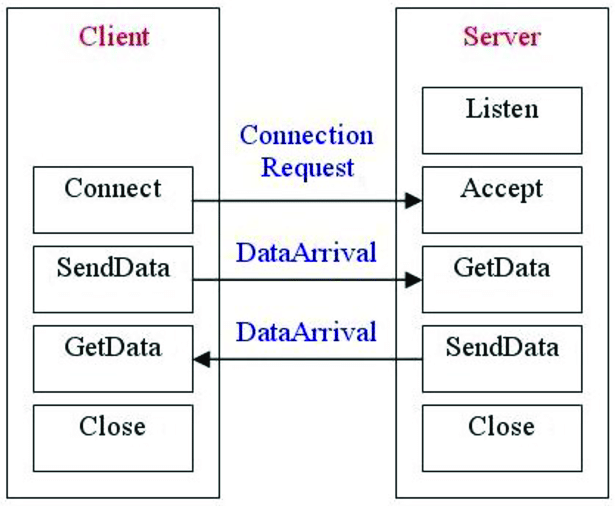
}

**3.3.3 Client Socket**

The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

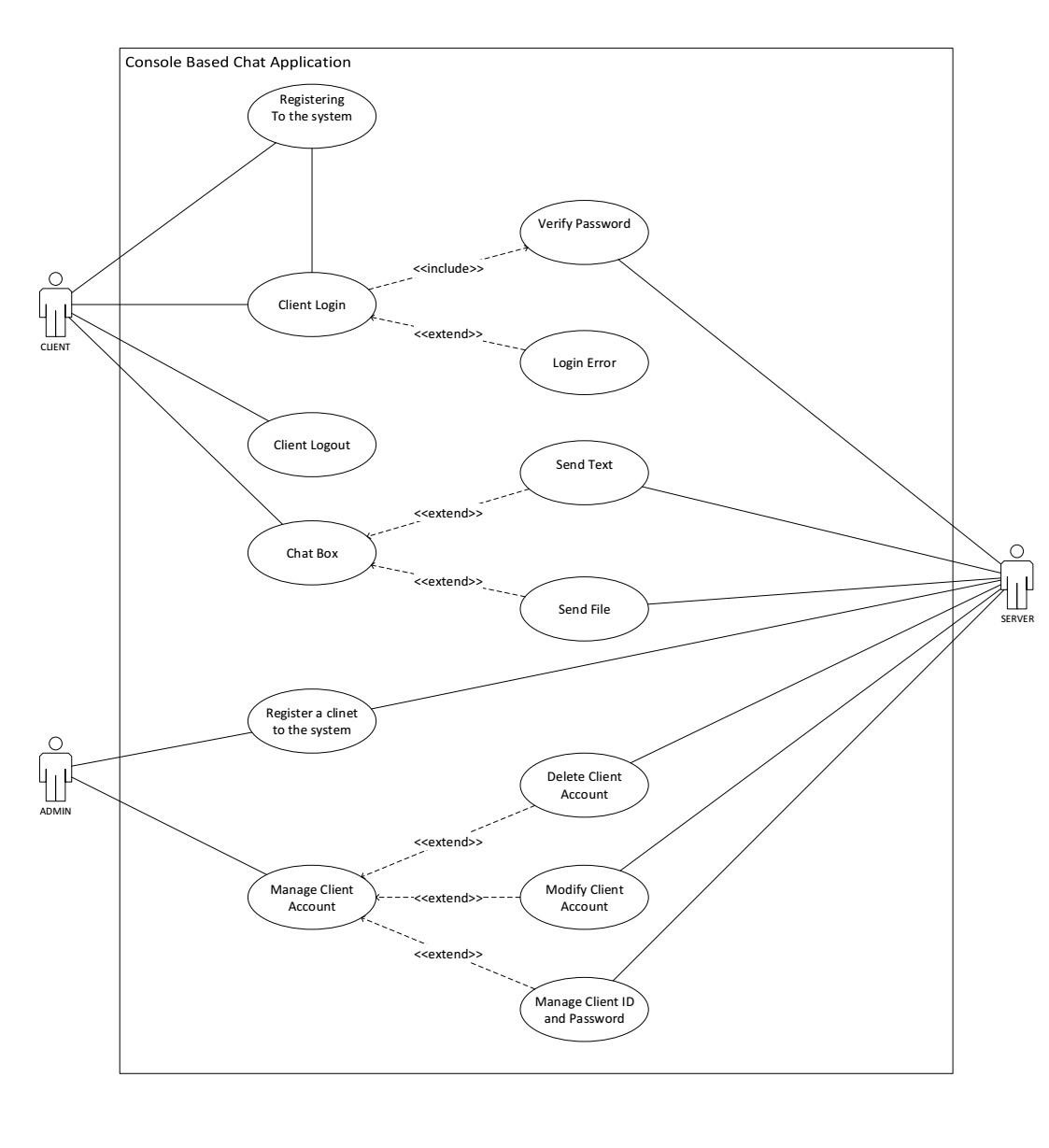
On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can now communicate by writing to or reading from their sockets.



**4. SYSTEM DESIGN AND DEVELOPMENT**

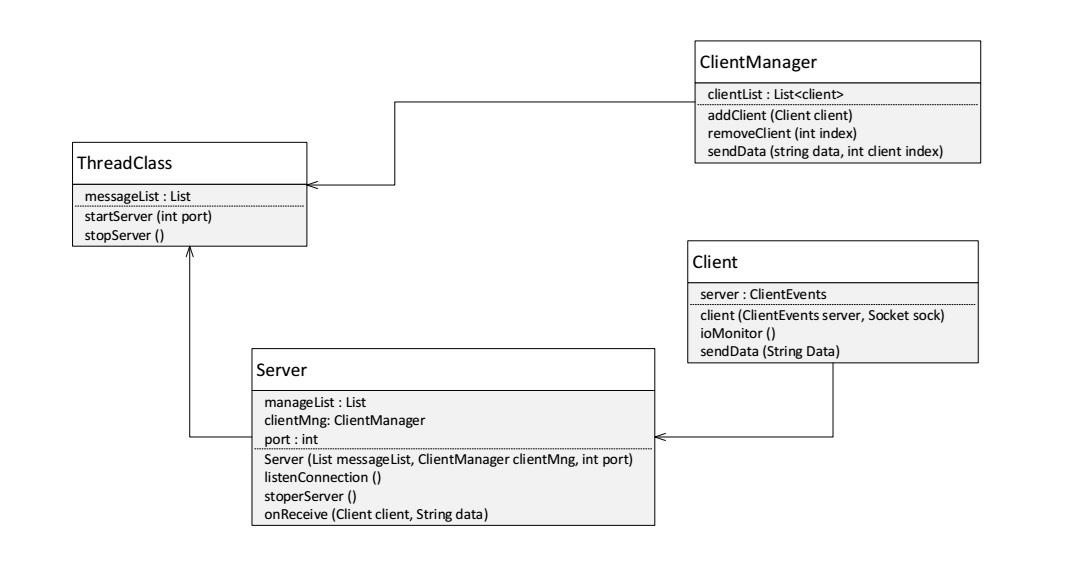
**4.1 Use Case Diagram**

A use case is a usage scenario for an external entity, known as actor, and the system-to-be. A use case represents an activity that an actor can perform on the system and what the system does in response.



**4.2 Class Diagram**

A class diagram in the Unified Modeling Language is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations, and the relationships among objects. Class diagram is created simply by reading the class names and their operations off of the interaction diagrams.



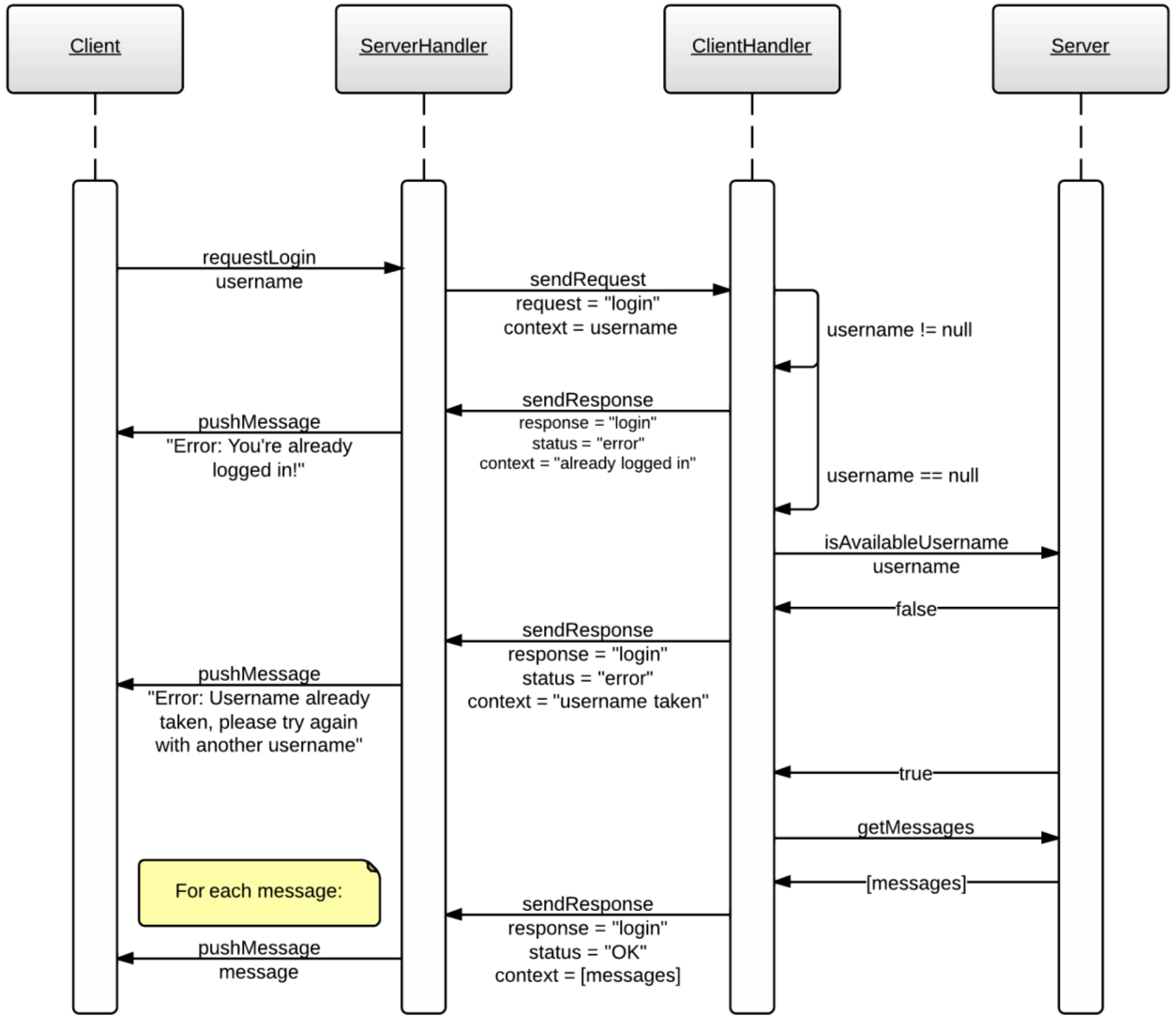
**4.3 Login Sequence Diagram**

The login sequence is initiated by the Client by calling the requestLogin method on the ServerHandler. A request is sent to the ClientHandler using the request “login” with the desired username as context.

The ClientHandler then checks if any username has been assigned to the client. If it has, it means the client is already logged in, and we need to create an error response notifying about how the login was rejected. The ClientHandler pushes the error.

If the user was not logged in, the ClientHandler checks with the Server to see if the username is available. If it’s not, an error response is created and sent to the Client, and the ClientHandler pushes the error.

However, if the username is available, a response with status “OK” is created and sent along with a list holding all the messages (a backlog) as context. Each of the messages are then pushed to the Client by the ClientHandler.



**4.4 Client Server Communication Sequence Diagram**

The message sequence is initiated by the Client by calling the requestSendMessage method on the ServerHandler. The request is then sent to the ClientHandler as a “message” request, using the message as context.

If the user is not logged in, no username has been assigned to the client, in which case the ClientHandler immediately returns an error response.

However, if the user is logged in, the ClientHandler creates a response with the status “OK”, before it pushes the message to the Server. The Server then pushes the message to all ClientHandler objects given that they are logged in, this of course includes the initiating ClientHandler. This triggers the ClientHandler to send a new response, called “new message”, which carries the message as context.

