Github: https://github.com/edev/CS300-term-project Dylan Laufenberg

Term Project: *Chat Application*

Design Document

Table of Contents

1 Introduction

This document presents the software design specification for the CS 300 Spring 2017 term project, AKA Chat Application. The application consists of two executables, a server and a client. The former will connect to the latter, authenticate, and perform various chat-related activities.

The design specification below consists primarily of text descriptions and pseudocode, with diagrams to supplement these details and illustrate the connections and overall structure of the project.

1.1 Purpose and Scope

The design specification is drafted with the intention of being ready for execution by a single developer, which is appropriate given the size of the project. As such, certain decisions, such as the exact protocol to be implemented through Google Protobuf, are not specified in detail. These details, in the view of the author, are easily and most readily defined through implementation, and given that the project has only one implementer, implementing these details consistently is trivial.

1.2 Target Audience

The target audience for this document is primarily the developer who will implement the code, Dylan Laufenberg (the document's author). The secondary audience is the grading team.

1.3 Terms and Definitions

Netty – "Netty is an asynchronous event-driven network application framework
for rapid development of maintainable high performance protocol servers & clients." See http://netty.io/

Protobuf – shortening of Google project Protocol Buffers. “Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.” See https://developers.google.com/protocol-buffers/

2 Design Considerations

This section discusses design constraints, implementation dependencies, and design methodology as they pertain to the Chat Application project.

2.1 Constraints and Dependencies

Both server and client need to be implemented entirely in Java SE 1.8, intended for use primarily on desktop x86-compatible architecture.

The design holds two primary dependencies: Netty (http://netty.io/) and Google Protobuf (https://developers.google.com/protocol-buffers/). Together, these handle large sections of the application’s data transmission and storage.

2.2 Methodology

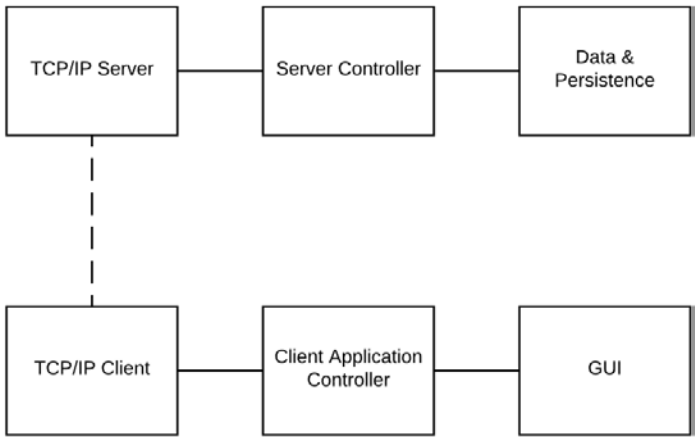
The chat application employs object-oriented design to achieve orthogonality and separation of concerns where possible. The extensive nature of the Netty framework limits this somewhat. In general, the most significant classes can be divided into three categories: Netty-related classes, controller classes, and GUI classes.

The application also practices maximum practical code reuse. As part of this, much of the lower-level work is delegated to either Netty or Google Protobuf.

3 System Overview

The system consists of six basic components, three pertaining to the server and three pertaining to the client. The TCP/IP server and TCP/IP client communicate with one another, and these are the only lines of communication between server and client programs. In both the server and the client, a controller component interprets messages received from the TCP/IP server/client component, handles the primary business logic pertaining to requested actions, and sends messages to other components of each respective system. In both server and client, the controller component is the central hub through which other classes primarily communicate.

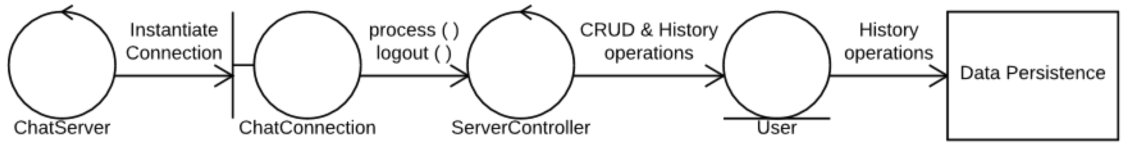
3.1 System Overview Diagram



4 System Architecture - Server

This section gives a high-level overview of the classes that comprise each server component listed in the System Overview section (see above). It describes the interfaces by which components communicate with one another and the basic jobs for which each class is responsible. Below, in brief, is a sequence diagram outlining the high-level interactions and interfaces between the server-side classes.

4.1 Server-Side Sequence Diagram



4.2 TCP/IP Server

The TCP/IP Server component consists of the ChatServer and ChatConnection classes. Together, they implement required Netty server functionality and provide an interface to the Server Controller component.

4.2.1 ChatServer

The ChatServer class provides the entry point for program execution and immediately bootstraps the network server on a port that can be specified at runtime.

4.2.2 ChatConnection

The ChatConnection class defines the handler that each channel instantiates, i.e. the handler that the server instantiates for each client connection. It stores a reference to the User to which the connection pertains and ChatConnection communicates to client messages and status changes to the Server Controller component via the process and logout methods.

4.3 Server Controller

The Server Controller component consists entirely of the static-only ServerController class and is responsible for coordinating the other components of the server as well as mapping the primary business logic of the server. ServerController provides an interface to receive messages via process and logout methods, decodes messages, and passes them to methods that correspond directly to use cases, e.g. login, register, and publicMessage. The ServerController class holds a thread-safe map of all active Users and Channels to aid in processing business logic.

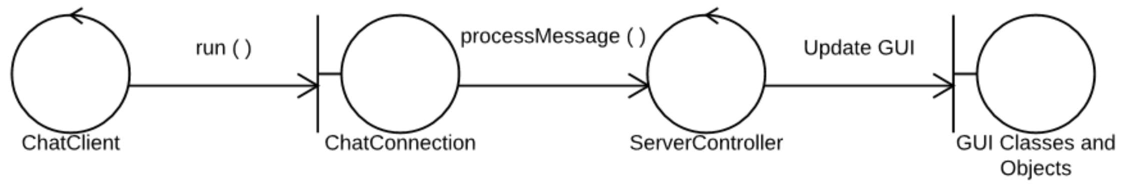
4.4 Data & Persistence

The Data & Persistence module consists of the User class and, informally, includes Google protobuf-generated classes, which are used both by User and by the Server Controller component. User is primarily responsible for managing a user’s identity information and persisting a user’s chat history with the help of Protobuf.

5 System Architecture – Client

This section gives a high-level overview of the classes that comprise each client component listed in the System Overview section (see above). It describes the interfaces by which components communicate with one another and the basic jobs for which each class is responsible. Below, in brief, is a sequence diagram outlining the high-level interactions and interfaces between the client-side classes.

5.1 Client-Side Sequence Diagram



5.2 TCP/IP Client

The TCP/IP Client component consists of the ChatClient and ChatHandler classes. Together, they implement required Netty client functionality and provide an interface to the Client Application Controller component. The Client Application Controller component communicates with ChatClient by setting its host and port fields and calling the run method. The Client Application Controller component holds a reference to the Netty Channel object through which it can talk to the server and communicates directly with it to write. The ChatHandler responds to incoming messages and status changes and communicates them to the Client Application Controller via processMessage method.

5.2.1 ChatClient

The ChatClient class bootstraps the client's connection to the server. It consists of host and port fields as public instance variables and a run method that attempts to connect to the server at said host and port.

5.2.2 ChatHandler

The ChatHandler class primarily responds to messages from the server and status changes (e.g. lost or closed connection). It passes messages from the server on to the Client Application Controller by invoking the processMessage method of ChatApplication.

5.3 Client Application Controller

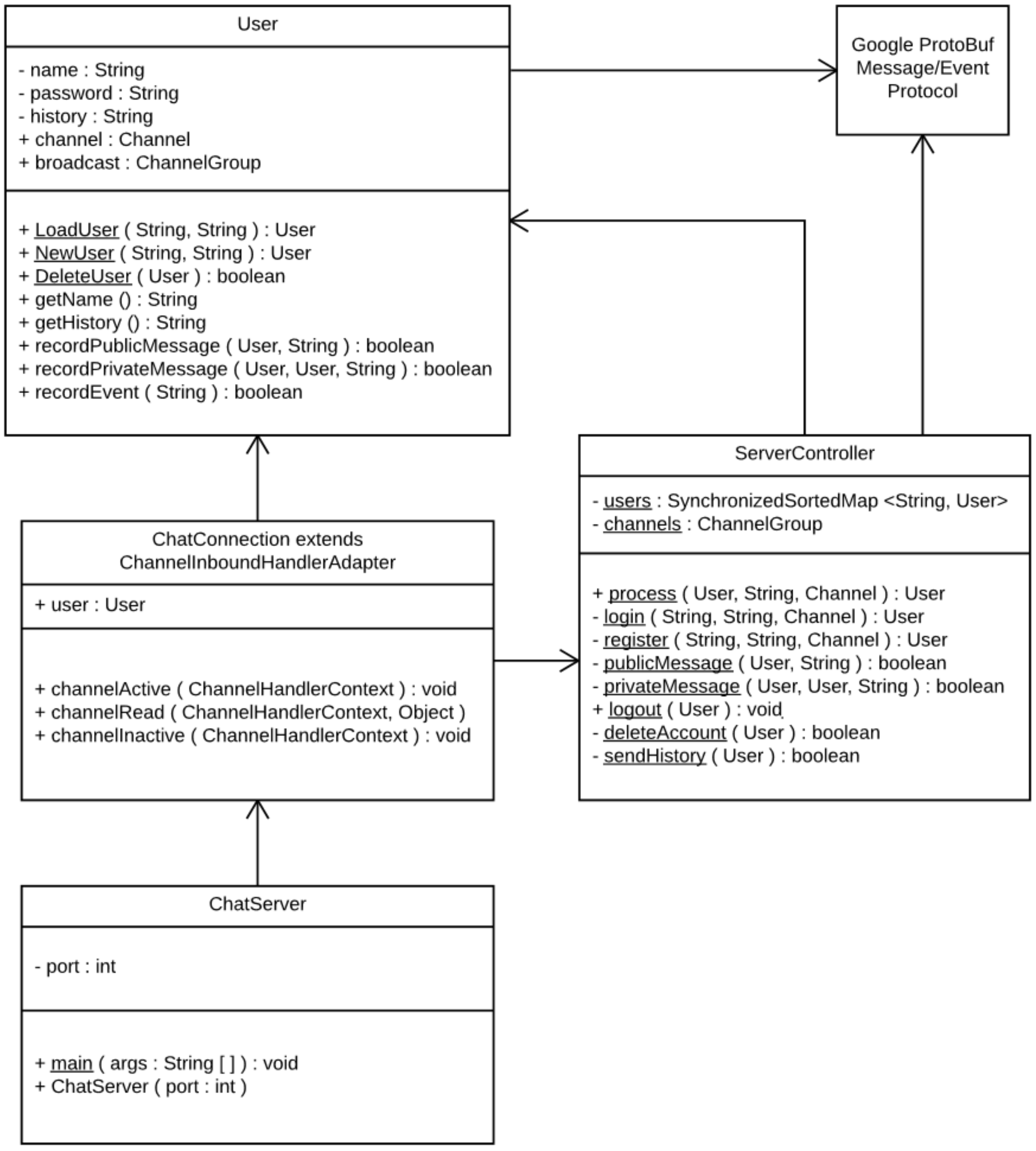
The Client Application Controller consists of the ChatApplication class, ChatApplication receives messages from the server to which it is connected via the processMessage method and sends messages to the server via the Channel object reference it stores. It processes the primary business logic of the client application through named functions corresponding to use cases, e.g. login, register, and sendPublicMessage. ChatApplication invokes Protobuf-generated classes to encode and decode mssages.

5.4 Graphical User Interface (GUI)

The client GUI is a collection of trivial classes such as LoginScreen, ChatScreen, and ChatHistoryScreen. Each instantiates a JFrame or other appropriate Swing object and populates it as needed. Each provides the functionality required through very simple methods that are left to the implementer or, where appropriate, by simply exposing GUI components to ChatApplication as public instance variables.

6 Detailed System Design - Server

6.1 Server-Side UML Class Diagram



User
- name : String
- password : String
- history : String
+ channel : Channel
+ broadcast : ChannelGroup
+ LoadUser (String, String) : User
+ NewUser (String, String) : User
+ DeleteUser (User) : boolean
+ getName () : String
+ getHistory () : String
+ recordPublicMessage (User, String) : boolean
+ recordPrivateMessage (User, User, String) : boolean
+ recordEvent (String) : boolean
Google ProtoBuf Message/Event Protocol
ServerController
- users : SynchronizedSortedMap <String, User>
- channels : ChannelGroup
+ process (User, String, Channel) : User
- login (String, String, Channel) : User
- register (String, String, Channel) : User
- publicMessage (User, String) : boolean
- privateMessage (User, User, String) : boolean
+ logout (User) : void
- deleteAccount (User) : boolean
- sendHistory (User) : boolean
ChatConnection extends ChannelInboundHandlerAdapter
+ user : User
+ channelActive (ChannelHandlerContext) : void
+ channelRead (ChannelHandlerContext, Object)
+ channelInactive (ChannelHandlerContext) : void
ChatServer
- port : int
+ main (args : String [ ]) : void
+ ChatServer (port : int)

6.2 ChatServer

ChatServer serves two purposes. First, it provides the entry point for server-side code execution, which parses the port passed in as a command-line argument or defaults to a port to be determined during implementation. Second, the main function creates a new ChatServer object, which bootstraps the Netty server, including directing Netty to create new ChatConnection objects to handle new Channels. ChatServer will set up a delimiter-based message decoder in the Channel pipeline that will be compatible with Protobuf encoding or otherwise configure the server to work with Protobuf for protocol handling. For source reference, see Netty User Guide.

6.3 ChatConnection

Instances of ChatConnection handle a network connection to one client. While the channel underlying the ChatConnection object is responsible for handling write requests, the ChatConnection handles incoming messages and status changes (i.e. lost or closed connections). Each ChatConnection records the User who is logged in on that particular connection as an instance variable and passes it along to ServerController when needed. ChatConnection provides the following functions:

public void ChannelActive(ChannelHandlerContext)
Performs any required actions as soon as the channel is active, as determined during implementation. (None are yet foreseen.)

public void ChannelRead(ChannelHandlerContext, Object)
Sends complete messages to ServerController:

```
user = ServerController.process(User, Message,
Channel)
if user == null:
     // Channel does not correspond to a User.
     Close connection.
```

public void ChannelInactive(ChannelHandlerContext)
Performs close-out clean-up when a channel closes. In addition to any Netty-required functionality, the function also does the following:

```
if user is not null:
     ServerController.logout(user)
     user = null
```

6.4 ServerController

ServerController is a static-only class that performs the primary business logic of the server via the following functions:

public static User process(User, String, Channel)
Uses Google Protobuf to decode a message (unless ChatServer incorporates a Protobuf decoder into the Channel pipeline). Then delegates the decoded message to the appropriate member method of ServerController. Returns the User associated with the channel, which may change in case of login, register, logout, and delete account.

private static User login(String name, String password, Channel channel)
Attempts to log the specified user in with the provided password:

```
user = User.LoadUser()
if user is null:
     return null
// Else, User found a valid, existing record.
user.channel = channel
user.broadcast = channels
Write login message to channels using Protobuf
Add channel to User.broadcast for every user in users
Add user to users for future lookup
Add channel to channels
```

```
    return user
```

private static User register(String, String, Channel)

Similar to login, except invokes User.NewUser instead of User.LoadUser.

private static boolean publicMessage(User user, String message)

Broadcasts the message to all other uses by writing to user.broadcast using Protobuf and adds the message to all users' histories by invoking recordPublicMessage on ServerController.users. Returns true on success and false on failure. On failure, also writes an error message to user.channel using Protobuf.

private static boolean privateMessage(User sender, User receiver, String message)

Adds the message to both sender and receiver's histories by invoking recordPrivateMessage on both and writes the message to receiver.channel using Protobuf. Returns true on success and false on failure. On failure, also writes an error message to sender.channel using Protobuf.

public static void logout(User user)

Logs user out as follows:

```
    Remove user from users
    Write logout message to user.broadcast using Protobuf
    Return null
```

private static boolean deleteAccount(User user)

Invokes logout(user) followed by User.DeleteUser(user). Returns true on success and false on failure.

private static boolean sendHistory(User user)

Invokes user.getHistory() and writes the associated data to user.channel using Protobuf. Returns true on success and false on failure.

6.5 User

The User class models a chat user and handles persistence of the associated data using the Protobuf codec. The actual persistence scheme, and any tools used to aid in persistence, are left to the discretion of the implementer. Static methods allow the ServerController to load, create, and delete Users, while instance methods provide the interface to individual users. The channel and broadcast public instance variables are managed externally (by ServerController).

public static User LoadUser(String name, String password)
Finds the user's on-disk record by name, loads it into a new User object using Protobuf, and checks the provided password against the stored password. If no record is found or the password does not match, returns null. On success, returns the populated User.

public static User NewUser(String name, String password)
Checks for the existence of a user's on-disk record by the given name. If one exists, returns null. Otherwise, checks the password against password requirements (see Requirements specification – Terms & Definitions). If the password fulfills the password requirements, creates a new User with the given name and password, stores it to disk with the help of Protobuf, and returns the User object. Otherwise, returns null without changing on-disk state.

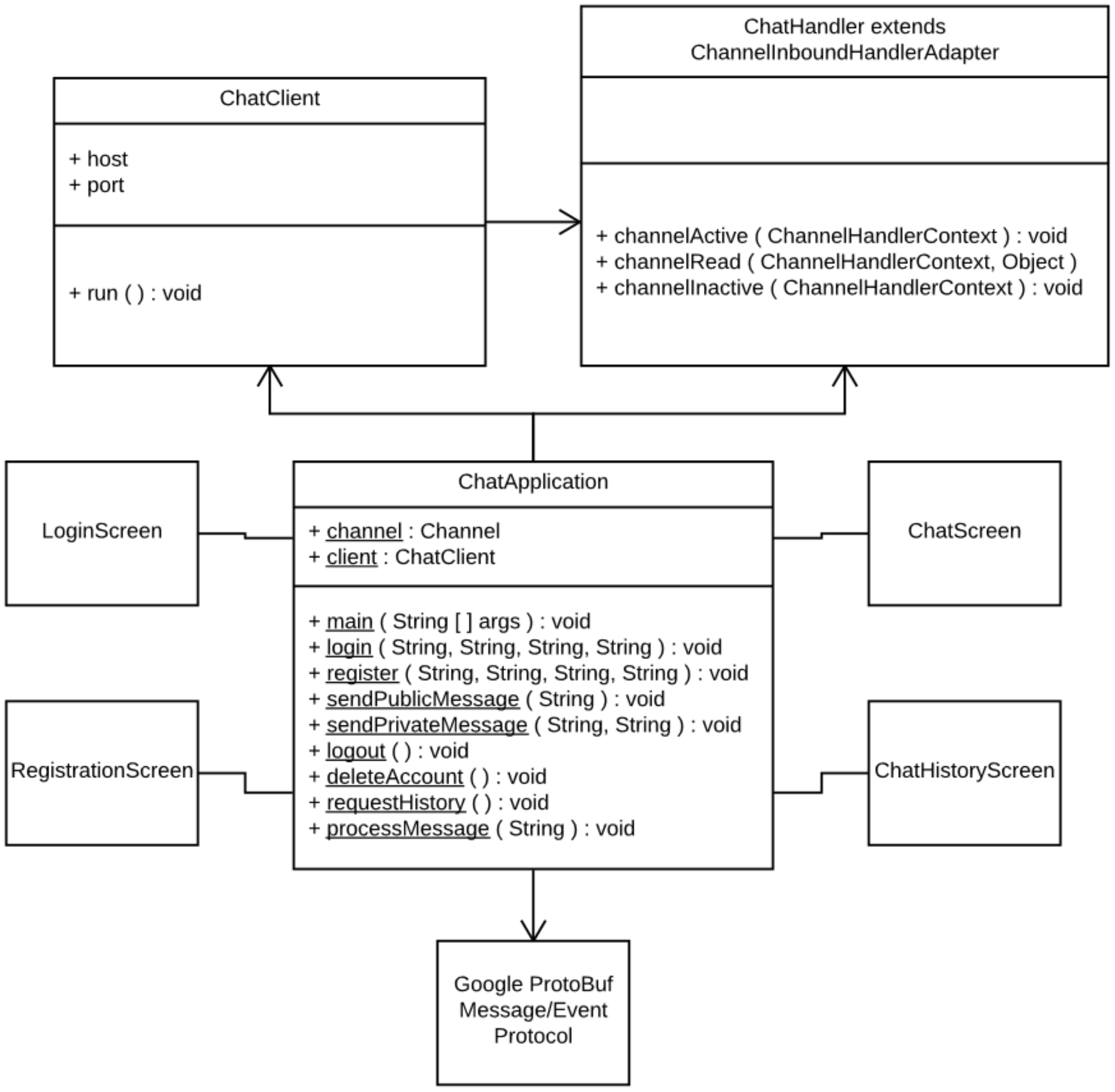
public boolean DeleteUser(User user)
Checks for the existence of a user's on-disk record based on user.name. If one exists, deletes it and returns true. Otherwise, returns false. If deletion fails, returns false.

public boolean recordPublicMessage(User sender, String message)
Encodes message from sender using Protobuf, appends to this.history, and writes the change to disk.

recordPrivateMessage(User, User, String), recordEvent(String)
These methods work similarly to recordPublicMessage.

7 Detailed System Design – Client

7.1 Client-Side UML Class Diagram


ChatHandler extends ChannelInboundHandlerAdapter
+ channelActive (ChannelHandlerContext) : void
+ channelRead (ChannelHandlerContext, Object)
+ channelInactive (ChannelHandlerContext) : void
ChatClient
+ host
+ port
+ run ( ) : void
LoginScreen
ChatApplication
+ channel : Channel
+ client : ChatClient
+ main (String [ ] args) : void
+ login (String, String, String, String) : void
+ register (String, String, String, String) : void
+ sendPublicMessage (String) : void
+ sendPrivateMessage (String, String) : void
+ logout ( ) : void
+ deleteAccount ( ) : void
+ requestHistory ( ) : void
+ processMessage (String) : void
ChatScreen
RegistrationScreen
ChatHistoryScreen
Google ProtoBuf Message/Event Protocol

7.2 ChatClient

ChatClient bootstraps the Netty server, including directing Netty to create a ChatHandler to handle the Channel that ChatClient opens. ChatClient will set up a delimiter-based message decoder in the Channel pipeline that will be compatible with Protobuf encoding or otherwise configure the client to work with Protobuf for protocol handling. For source reference, see Netty User Guide.

7.3 ChatHandler

ChatHandler handles the network connection through which the client communicates with the server. While the channel underlying the ChatHandler object is responsible for handling write requests, the ChatHandler handles incoming messages and status changes (i.e. lost or closed connections). ChatHandler provides the following functions:

public void ChannelActive(ChannelHandlerContext)
Sets ChatApplication.channel to the current channel. Performs any other required actions as soon as the channel is active, as determined during implementation.

public void ChannelRead(ChannelHandlerContext, Object)
Sends complete messages to ChatApplication.processMessage.

public void ChannelInactive(ChannelHandlerContext)
Performs close-out clean-up when a channel closes. In addition to any Netty-required functionality, the function also calls ChatApplication.logout.

7.4 ChatApplication

ChatApplication is a static-only class that provides the main entry point to the program, creates and manages the GUI (details left to the discretion of the implementer), and performs the primary business logic of the client via the following functions:

public static void main(String[] args)
Initializes the GUI, showing LoginScreen. Creates a ChatClient and assigns it to ChatApplication.client.

public static void processMessage(User, String, Channel)

Uses Google Protobuf to decode a message (unless ChatClient incorporates a Protobuf decoder into the Channel pipeline). Then performs the appropriate action:

1. In response to connection activation, displays ChatScreen.
2. In response to connection closure, calls logout().
3. In response to a chat history message, passes the message to ChatHistoryScreen and displays ChatHistoryScreen.
4. In response to a text message (public message, private message, or notification message), sends the message to ChatScreen for display.

private static void login(String name, String password, String host, String port)
Attempts to log in as the specified user with the provided password. Note: does not await a response or change the GUI except possibly to display a waiting message (at the discretion of the implementer).

```
Client.host = host
Client.port = port
client.run()
On channelActive:
    Encode login message with name and password using
    Protobuf
    Write message to channel
```

private static void register(String name, String password, String host, String port)
Similar to login.

private static void sendPublicMessage(String message)
Encodes message using Protobuf and writes it to channel.

private static void sendPrivateMessage(String recipientName, String message)
Similar to sendPublicMessage.

public static void logout()

Logs user out as follows:

```
If channel is active:
     Close channel
Show LoginScreen
```

private static void deleteAccount()

Encodes account deletion message using Protobuf and writes it to channel.

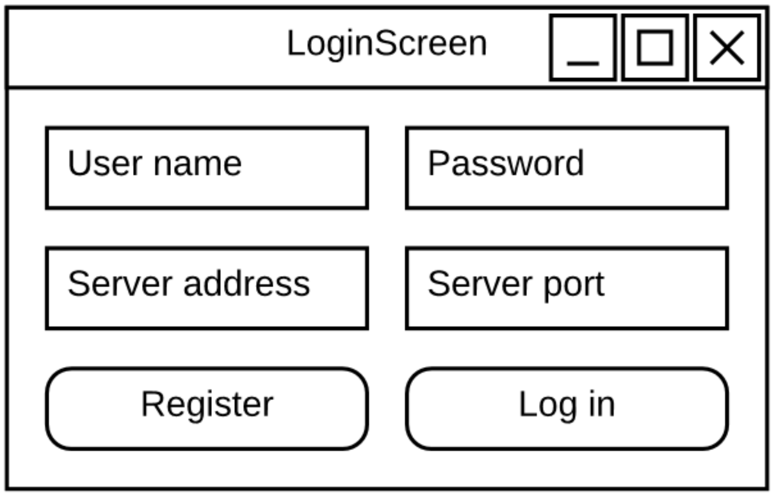
private void requestHistory()

Encodes history request message using Protobuf and writes it to channel.

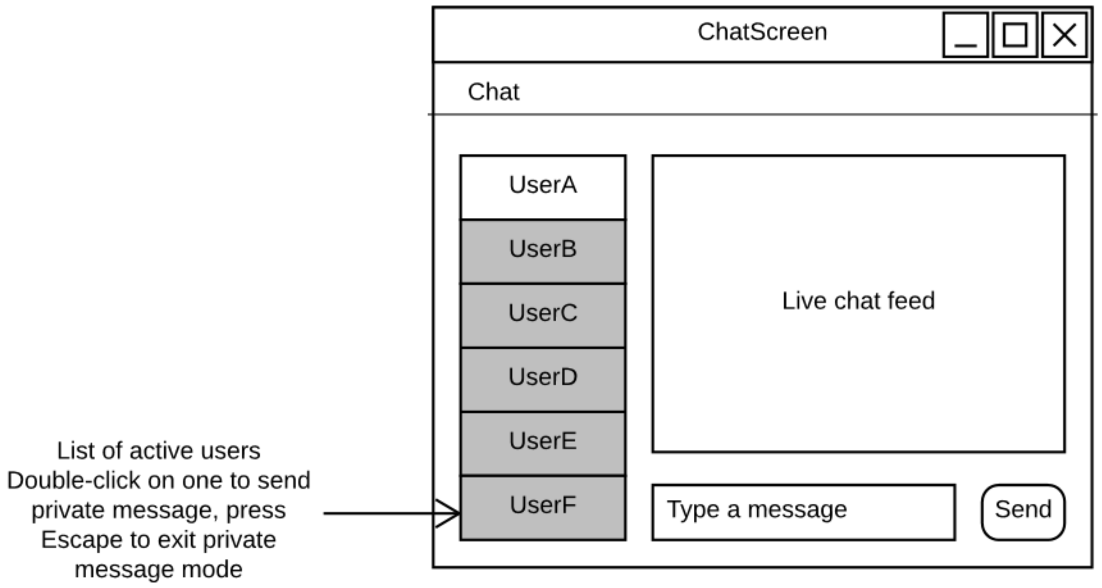
7.5 GUI Classes

The GUI classes are very simple and are left to the discretion of the implementer. In this document they are referred to as LoginScreen, ChatScreen, and ChatHistoryScreen, but these may be changed at the time of implementation, and if appropriate, some may be implemented simply as JFrames or other Swing objects rather than as classes encapsulating these objects. Some, trivial logic relating to GUI functions is omitted from this document. Approximate mockups of the three primary screens appear below, but they are subject to modification by the implementer.

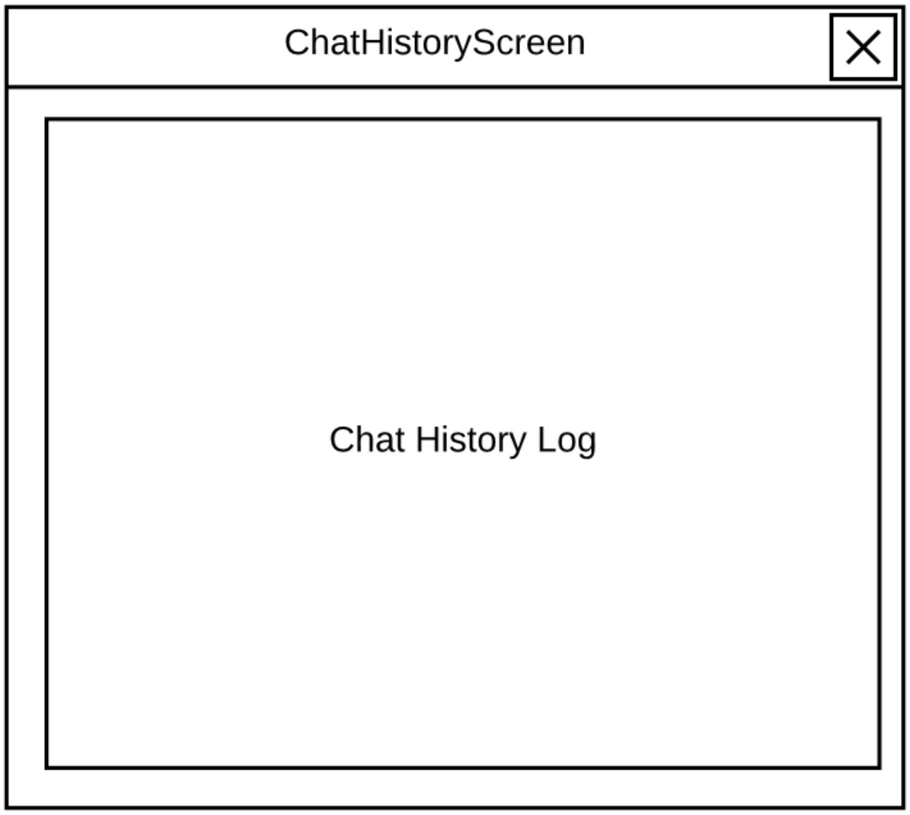
7.5.1 LoginScreen Mockup



7.5.2 ChatScreen Mockup


ChatScreen
Chat
UserA
UserB
UserC
UserD
UserE
UserF
Live chat feed
Type a message
Send
List of active users Double-click on one to send private message, press Escape to exit private message mode

7.5.3 ChatHistoryScreen Mockup


ChatHistoryScreen
Chat History Log