William Trefethen

Talk

Design Document

Table of Contents

1. Introduction 3

1.1. Purpose and Scope 3

1.2. Target Audience 3

1.3. Terms and Definitions 3

2. Design Considerations 4

2.1. Constraints and Dependencies 4

2.2. Methodology 4

3. System Overview 5

4. System Architecture 6

4.1. Networking 6

4.1.1. Connection 6

4.1.2. Packets 6

4.2. UI 7

4.2.1. Screens 7

4.2.2. Screen Stack 7

4.3. User Backend 8

4.4. Chat Backend 8

5. Detailed System Design 9

5.1. Networking 9

5.1.1. Packets 9

5.2. UI 10

5.2.1. Screens 10

5.3. User Backend 10

5.4. Chat Backend..*11*

1. Introduction

This is the design document for Talk, a chat application. It's purpose will be to communicate the details of the applications architecture and design.

1.1. Purpose and Scope

The purpose of this document will be to express a detailed design for the Talk chat application. It will cover high-level architecture as well as granular subsystem specifications.

1.2. Target Audience

The target audience of this document is developers and clients alike. It is to serve as a basis for development, a documentation of the statement of work, and as a guide for maintenance in the future.

1.3. Terms and Definitions

UI, GUI: Graphical User Interface

Packet: An object used to send data cross-network

User: Any client using the Talk chat application to chat with others

2. Design Considerations

The purpose of this section will be to outline the major considerations in undertaking the development of the application's design. Of these considerations, there is constraints and methodologies.

2.1. Constraints and Dependencies

The major constraints of the application are that it be developed entirely in Java. Beyond this there are its functional and non-functional requirements as outlined in the requirements document.

2.2. Methodology

The major software engineering methodology being executed is the waterfall methodology. After having played the groundwork in the requirements document, this document is to serve as the groundwork for the implementation, and so on through the rest of the waterfall stack.

3. System Overview

The system is composed of two major components: Client and Server. These act as independent programs running on separate machines. The server will listen on a public IPv4 TCP socket for client connections, which will initialize their communication through a packet-based networking subsystem.

The client is composed mainly of two parts - networking and UI. The server is in turn also composed of two major parts - networking and backend. On the highest level of abstraction, the system's architecture functions like one single unit - the server's backend connected seamlessly to the client's fronting (UI) - the UI serving as computer-human interface for the backend data processing on the server's backend system.

On the next level lower, we introduce a networking system on both client and server to implement the practical connection cross-network. Because this system flows from a higher level abstract concept of client-server unity, the networking systems will be *symmetrical* - meaning that the systems will communicate with a common language - that language is the packet system.

The client's UI will be handled much like an intricate stack. Screens, which comprise the user's experience, can be pushed and popped on and off of the UI stack as users go deeper or reverse out of use case flows. This FIFO structure makes a stack perfect for the UI implementation.

The server's backend system will function mainly in the use of two major control classes - UserManager and ChatManager. UserManager is the controller for all user-related operations. It handles loading accounts from disk, creating new accounts on register, login and logout, and user-info-related packet requests.ChatManager is the controller for all chat-related operations. It handles new chat creation, new chat messages, chat history requests, and loading and saving chats from disk.

4. System Architecture

We will define our system level architecture in 4 main components - UI, Networking, User, and Chat. A focussed class diagram is shown below which illustrates cross network communication and the basic process of reading and writing packets. Please refer to the appendices for the encompassing data-flow diagram and class diagram that accompany this section.

4.1. Networking

The networking package is defined in terms of its interface. In short it provides a base data-driven class, called a Packet, and a function to send them.

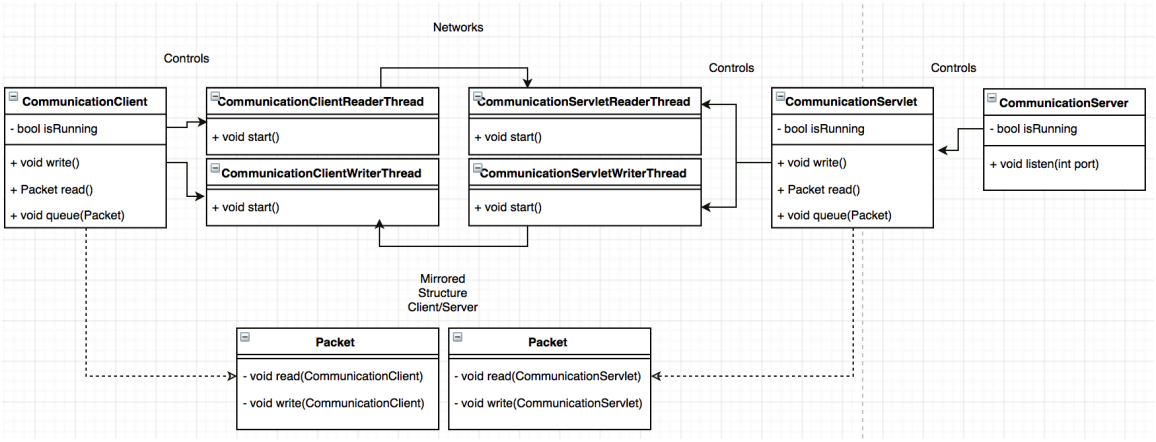
4.1.1. Connection

Each client will establish a TCP connection with the server, which will be listening on a publicly accessible server. Upon connect, the server will transfer the connection into a CommunicationServlet, which will in turn run its reader and writer threads asynchronously on two new threads. The multithreaded approach is built for high speed rather than high scalability, although multithreading isn't always detrimental to long-term scalability. The client, having connected, will in turn set up paralleled reader and writer threads which run asynchronously non-blocking on the socket's input and output streams.

4.1.2. Packets

The main interface of the CommunicationClient and CommunicationServlet (parallel client and server classes respectively), is to queue Packets. Each of these classes will maintain a queue of Packets to send in a thread-safe manner. Upon the writer threads update, the next packet in the queue will be written to the socket output stream. On the other side, upon the update of the reader thread, the socket will attempt to read the packet, if it is successful the packet will be processed. Packets have two main functions -

read and write, which read their data in and out of binary on either side of the send process.



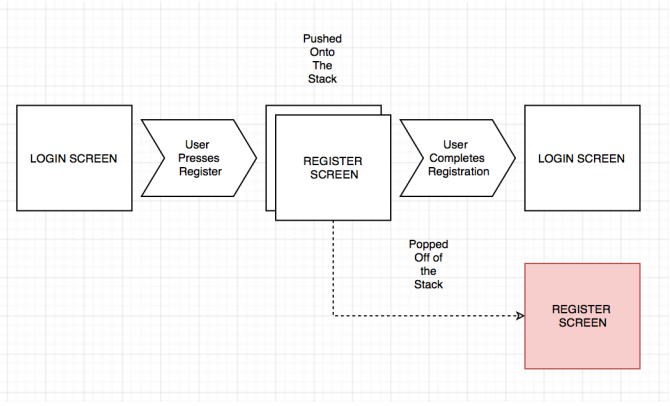
4.2. UI

The Client's UI system will function on the basis of a UI stack. The main object of the UI stack is a Screen.

4.2.1. Screens

Screens are objects which represent a graphical user interface in the form of an object. They provide handlers for all of the event-driven button and input field interaction. These event-driven methods primarily send Packets to the server to retrieve information. They also provide a handy function for displaying alerts at any given time and the top of the screen in a non-intrusive way, which comes in handy multiple times throughout the UI implementation.

4.2.2. Screen Stack



The Screens, organized by the UIManager, are organized on a stack. As users move through the user flow possibilities, it is intuitive to be able to go back - this particular action is very

simple to achieve using a stack, hence why screens are organized in this way. The bottom of the stack will always be the “root controller” as they call it in UI speak. This root will be LoginScreen. If a user hits Register, RegisterScreen will push onto the stack. If they complete Registration, RegisterScreen is then popped off the stack to reveal yet again the Login screen.

4.3. User Backend

The User Backend is located on the server-side of the program and operates from the centralized control of the UserManager. The UserManager operates with the primary ADT of a *User*. Users can be loaded from file by UserManager into a user list, and online user’s are mapped by UserManager to connection ID’s.The manager can create new users, and is also responsible for notifying clients of user logouts and logins.

4.4. Chat Backend

The Chat backend is another server-side backend system with operates on the basis of a centralized control class. The ChatManager deals with Chats in a similar way that UserManager deals with users. Chats are the main ADT of ChatManager, Chat objects can be created by the ChatManager. ChatManager is responsible for loading Chat objects in and out of files. It is additionally responsible for routing some packets to chat objects which they pertain to, but in most cases allows Chat objects to directly send packets to the Networking system. Chats are comprised of Messages and Users, Messages are comprised of a User and a String message.

5. Detailed System Design

5.1. Networking

The networking system will be implemented using Java's java.net.socket package. On the server in CommunicationServer, a java.net.ServerSocket will be created which will listen on a given port. It will then begin a thread which will loop (with blocking) to listen for new connections. Upon receiving a connection, it will instantiate a new java.net.socket.Socket, and create a new CommunicationServlet with this socket. The servlet will in turn then begin its own non blocking reader and writer threads. The server will keep track of connections in the form of an ArrayList of connectionID's (ints). This will form the basis for data transmission. On the client - an almost identical process to the CommunicationServlet takes place in the CommunicationClient. This too creates a socket, however it then connects to a remote port. This socket is then set up as non blocking and runs its own reader and writer threads asynchronously.

5.1.1. Packets

Packets will be implemented as a polymorphic hierarchy. Each packet must be able to be translated in and out of binary. To accomplish this, each will have an ID, and there will be a static register of Packets (HashMap<Integer, Class> where class specifically refers to a subclass of Packet). From this, each packet can be identified after reading the first 4 bytes in the transmitted byte array by interpreting it as the id. Upon getting the correct Packet subclass, a binary reader function is called in the Packet, which utilizes the output stream of the servlet or client socket to read various types of primitives out in a given order. This order is mirrored precisely in a write() function on the other side which utilizes the sockets input stream. Due to the asynchronous nature of the multithreaded sockets, a thread safe queue in the form of an ArrayList is utilized to queue packets for delivery. This minimizes the risk of sending multiple packets at once on possibly different threads.

5.2. UI

The UIManager will be initialized statically, because there is only 1 static frame to bind to. It will provide static functions to push(Screen) a given screen onto the UIStack, as well as to pop the top screen off of the stack. It will additionally provide helper functions for drawing and other UI-related needs that the screens can access. Upon the push or pop of a screen, it will call the respective methods in the screen object is has been manipulating. This will give the screen a chance to do any initialization it may need to do to set up its data. For example. When the main screen is pushed, it will immediately queue a packet to request the User List as well as the current user's past chat id's. Upon receiving this info back from the server, it will then be able to initialize the bulk of its interface. This brings us to Screens.

5.2.1. Screens

Some screens will require a packet or two to initialize their data and display it to the user. For example both the main screen (as described above) and the chat screen need a significant amount of data to begin operation. Such requests are made in the onPush() method. From here, the screen classes generally provide handlers for events in the user interface. Whenever a button is pressed or a text field is edited, handlers in the screen classes will be called. These handlers will in turn often result in packets being sent to the server. For example when a user types a chat message. Upon pressing the enter button, the ScreenChat class will receive a handler call onChatSubmit(). It will then gather the necessary data - namely the message - and create a PacketChatMessage. It will then queue the packet. This summarizes the two main technical features of Screens - stack-related events and user-driven events.

5.3. User Backend

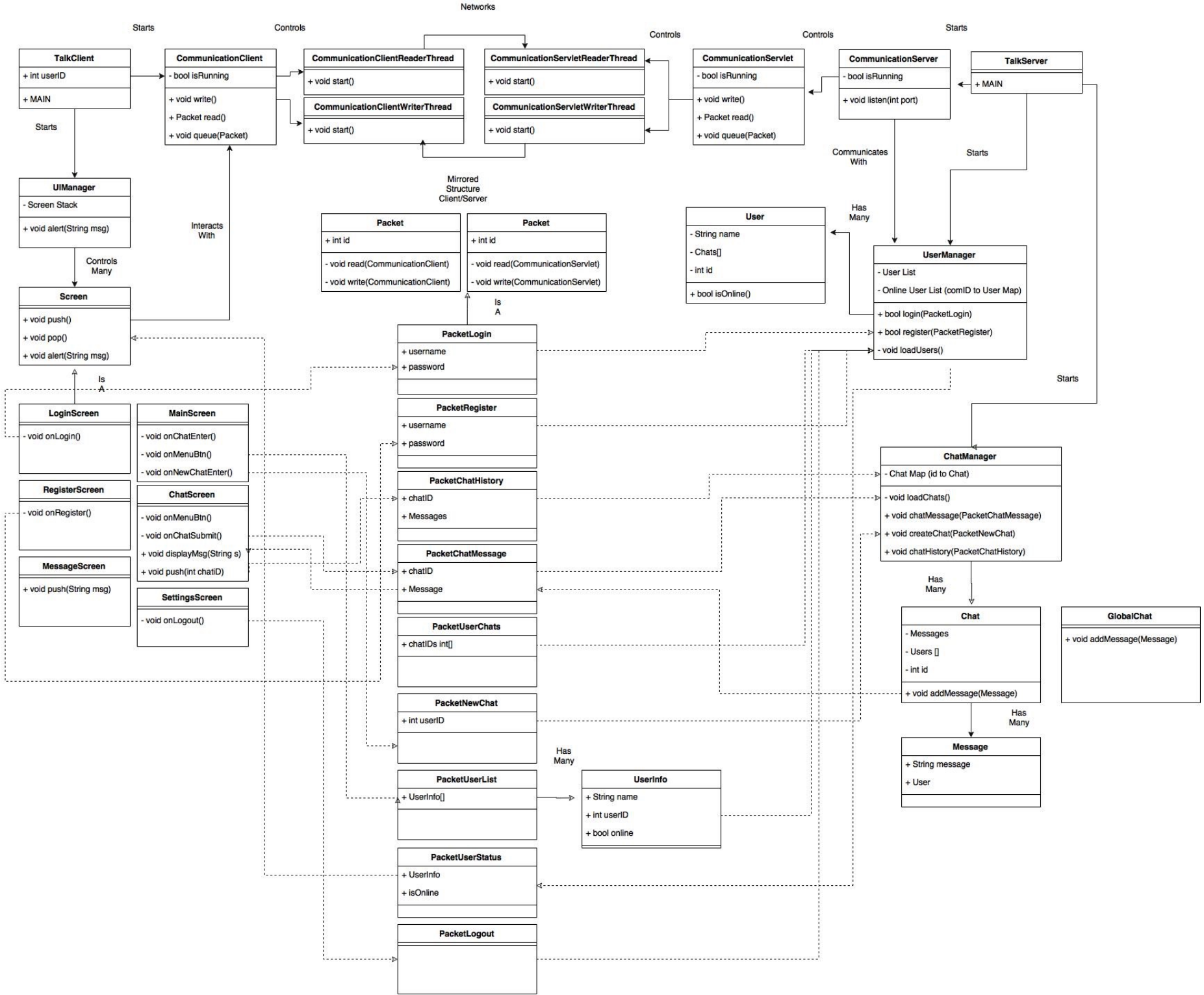
The UserManager will be initialized statically, as there only ever needs to be one centralized controller for users. It will, upon start, attempt to load account data from a

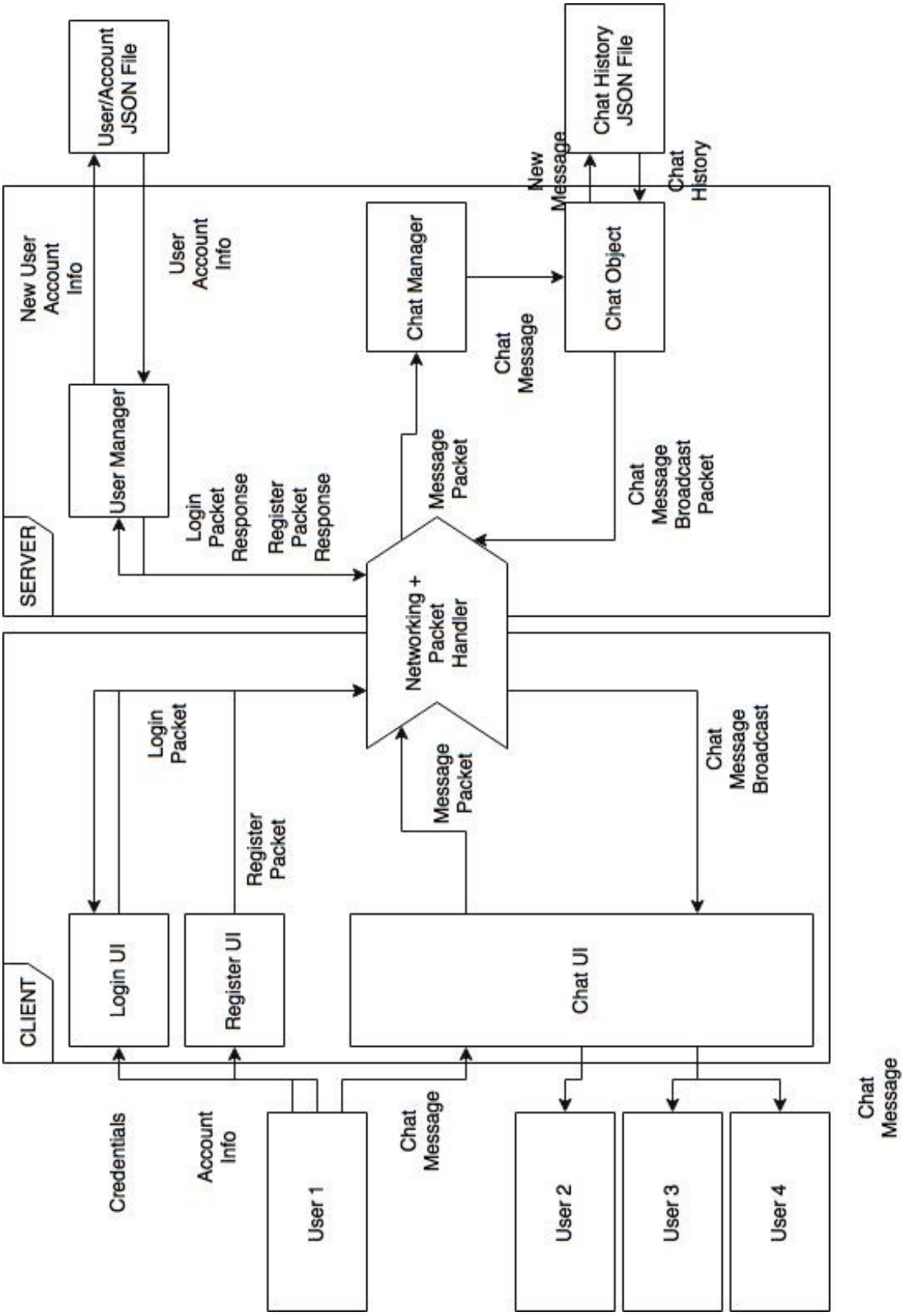
JSON file. It will rip an array of JSON objects into user objects, then place these into its user map (HashMap<string, User> where the key is the username). This is the boilerplate for all other operations the UserManager performs. Upon login, a PacketLogin is sent to UserManager, which then attempts to find the user in its map. If it does, it confirms the password and returns the packet as successful. This then causes the user to be added to the online Users list (HashMap<int, User> where the key is a connectionID). This second map allows for easy conversion between networking and User data. The UserManager will then notify all online users of the login event through a PacketUserStatus. UserManager will also handle packet requests from users for retrieving that user's chats - which come in the form of ID's - which are then used to further communicate with the chat backend upon retrieval.

5.4. Chat Backend

The Chat Manager will be initialized statically as there is only a need for one manager class. Upon start, it will attempt to load chat histories from a directory. Each file in the directory represents a chat in the form of a JSON file. The manager initializes a Chat object with a JSON object from which to rip its data. It then adds this chat to the chat map, (HashMap<int, Chat> where the key is the chatID). The map allows for easy access and reference by packet. The Chat Manager will receive all incoming PacketChatMessage Packets and route them to the correct chat based on chat ID. Upon receiving a new chat message, a Chat object will then broadcast to all listening users a new chat message packet while simultaneously writing the new message to the disk. Other than basic chat networking, ChatManager must handle the creation of new chats. Upon receiving a packet to do so, it will create a new chat object and add it to the chat map. From here, the chat id will be returned to the client, which can then use the id to request to open the chat. Upon double clicking a chat in their main menu, the user's client will send a PacketChatHistory to the server with the chat's id. The chat manager will receive this, find the chat object with the chat map, and add a listener to the chat. The

chat, upon receiving a new listener, will return the PacketChatHistory with a full chat history. The Chat object will additionally add the user to its listeners ArrayList. The final requirement is that upon logging in, a client must receive a list of all chats he/she has been involved in to populate the main screen. This will be done by adding chatIDs to an ArrayList in the User class which will be saved in their JSON file. By using this list, and adding to it every time a user enters his/her first message into the chat, the chat history can always be retrieved.


Networks
Starts
Controls
Controls
Controls
Starts
TalkClient
+ int userID
+ MAIN
CommunicationClient
- bool isRunning
+ void write()
+ Packet read()
+ void queue(Packet)
CommunicationClientReaderThread
+ void start()
CommunicationClientWriterThread
+ void start()
CommunicationServletReaderThread
+ void start()
CommunicationServletWriterThread
+ void start()
CommunicationServlet
- bool isRunning
+ void write()
+ Packet read()
+ void queue(Packet)
CommunicationServer
- bool isRunning
+ void listen(int port)
TalkServer
+ MAIN
Starts
Communicates With
Starts
Mirrored Structure Client/Server
UIManager
- Screen Stack
+ void alert(String msg)
Interacts With
Controls Many
Screen
+ void push()
+ void pop()
+ void alert(String msg)
Packet
+ int id
- void read(CommunicationClient)
- void write(CommunicationClient)
Packet
+ int id
- void read(CommunicationServlet)
- void write(CommunicationServlet)
Is A
User
- String name
- Chats[]
- int id
+ bool isOnline()
Has Many
UserManager
- User List
- Online User List (comID to User Map)
+ bool login(PacketLogin)
+ bool register(PacketRegister)
- void loadUsers()
Starts
Is A
LoginScreen
- void onLogin()
RegisterScreen
- void onRegister()
MessageScreen
+ void push(String msg)
MainScreen
- void onChatEnter()
- void onMenuBtn()
- void onNewChatEnter()
ChatScreen
- void onMenuBtn()
- void onChatSubmit()
+ void displayMsg(String s)
+ void push(int chatiD)
SettingsScreen
- void onLogout()
PacketLogin
+ username
+ password
PacketRegister
+ username
+ password
PacketChatHistory
+ chatID
+ Messages
PacketChatMessage
+ chatID
+ Message
PacketUserChats
+ chatIDs int[]
PacketNewChat
+ int userID
PacketUserList
+ UserInfo[]
PacketUserStatus
+ UserInfo
+ isOnline
PacketLogout
ChatManager
- Chat Map (id to Chat)
- void loadChats()
+ void chatMessage(PacketChatMessage)
+ void createChat(PacketNewChat)
+ void chatHistory(PacketChatHistory)
Has Many
Chat
- Messages
- Users []
- int id
+ void addMessage(Message)
GlobalChat
+ void addMessage(Message)
Has Many
Message
+ String message
+ User
Has Many
UserInfo
+ String name
+ int userID
+ bool online


SERVER
CLIENT
User/Account JSON File
Chat History JSON File
New User Account Info
User Account Info
New Message
Chat History
Chat Manager
Chat Object
Chat Message
User Manager
Login Packet Response
Register Packet Response
Message Packet
Chat Message Broadcast Packet
Networking + Packet Handler
Login Packet
Register Packet
Message Packet
Chat Message Broadcast
Login UI
Register UI
Chat UI
Credentials
Account Info
Chat Message
User 1
User 2
User 3
User 4
Chat Message