*Babble Chat*

Design Document

Table of Contents

1 Introduction 3

1.1 Purpose and Scope 3

1.2 Target Audience 3

1.3 Terms and Definitions 4

2 Design Considerations 5

2.1 Constraints and Dependencies 5

2.2 Methodology 6

3 System Overview 7

4 System Architecture 8

4.1 Netty Network Framework 8

4.1.1 Server, ServerInit, and ServerHandler Classes 8

4.1.2 User, UserInit, and UserHandler Classes 10

4.2 Interface 11

4.3 DB (Dataabase) 11

5 Detailed System Design 13

5.1 Server Class 14

5.1.1 run() 14

5.2 ServerInit Class 14

5.2.1 initChannel(SocketChannel arg0) 14

5.3 ServerHandler Class 15

5.3.1 handlerAdded(ChannelHandlerContext ctx) 15

5.3.2 handlerRemoved(ChannelHandlerContext ctx) 15

5.3.3 channelRead(ChannelHandlerContext ctx, Object msg) 15

5.4 User Class 16

5.4.1 run() 16

5.5 UserInit Class 16

5.5.1 initChannel(SocketChannel arg0) 16

5.6 UserHandler Class 17

5.6.1 channelRead(ChannelHandlerContext ctx, Object msg) 17

5.7 Interface Class 17

5.7.1 register() 17

5.7.2 login(String username, String validate) 17

5.7.3 send(String msg) 17

5.7.4 displayChat() 18

5.7.5 displayHistory() 18

5.7.6 displayFriends() 18

5.7.7 logout() 18

5.8 Database Class 18

5.8.1 createTable() 18

5.8.2 addUser(String username, String key) 18

5.8.3 findUser(String username, String key) 19

5.8.4 chatHist(int userID, int friendID) 19

5.8.5 addHist(int userID, int friendID) 19

**1 Introduction**

This section provides a high overview of the design for the Babble chat application.

## Purpose and Scope

The purpose of this document is to provide a design overview that will be used to support the implementation process later on. The design will cover considerations to ensure all of the use cases previously laid out in the Requirements Document as well as discussing possible constraints, dependencies and methodologies that will provide the best structure for the chat application.

The document will also include:

* the system architecture using an object-oriented programming approach and a high-level visualization of an OOP class diagram
* a more detailed system design with a low-level insight into methods and data supported by a UML diagram.

## Target Audience

The target audience for this document will be Professor Fei for CS300 and TA Bin Lin. From a creative standpoint, I will approach this design as if it is for a client who has hired me to create this chat service for their users who, in turn, what a simple chat interface to communicate for work or personal use.

## Terms and Definitions

|  |  |  |
| --- | --- | --- |
| **Term** |  | **Definition** |
| **Babble** |  | The name of the chat application |
| **Java** |  | The programming language that will be used to build the chat |
| **GUI** |  | Graphical User Interface |
| **Netty.io** |  | A client-serer framework for setting up a server (or servers) with improved networking capability for scaling systems (like a chat service) up |
| **Network socket (aka socket)** |  | “an internal endpoint for sending or receiving data at a single node in a computer network” ([Wikipedia: Network Socket](https://en.wikipedia.org/wiki/Network_socket)) |
| **pipeline** |  | A netty construction that opens a pipe between sockets for smoother byte transfer versus using sockets from java.net |
| **Apache Derby Database** |  | A structure for holding data, specifically oriented for use with Java applications |
| **OOP** |  | Object-Oriented Programming (involving a focus on objects and their relations with one another rather than alternative data- or operation-driven designs) |
| **User** |  | A person joining the chat channel to communicate from their specific IP address |
| **Server** |  | The main host and port that allows the user to connect and speak with users |
| **I.P. Address** |  | A unique identifier (internet protocol address) for different internet-connecting devices |

# Design Considerations

This section will cover constraints, dependencies and methodologies, which must be decided or resolved to have a smoother implementation down the line.

## Constraints and Dependencies

Because this application has many separate parts that must be joined together, one of the greatest constraints in its successful deployment will be combining the parts (e.g. classes) in an efficient manner.

As the environment is expected to be scalable (having one-to-one user communication as well as one-to-many), the netty.io framework will be used to easily scale communication using its *pipeline* model to open sockets between client(s) and server for streamed communication. As much of the netty library is created specifically for the purpose of opening up such channels of communication between two locations, its methods will dominate the foundation of the chat environment.

The Apache Derby Database will be used, as it is specific to Java applications, and store user data such as username, passwords, unique identifiers, friends, and chat histories. This should help maintain the encapsulation in the object-oriented design while making the user data easier to manage, store and restore (e.g. chat histories).

Possibly the most important constraint are the time constraint for when the client is expecting the final product (June 8th). To keep this deadline, this design (May 9th), following test plan (May 30th), and implementation must follow the strict deadlines set in the Requirements Document.

## Methodology

The chat application must be implemented using the Java language per the client’s expectation.

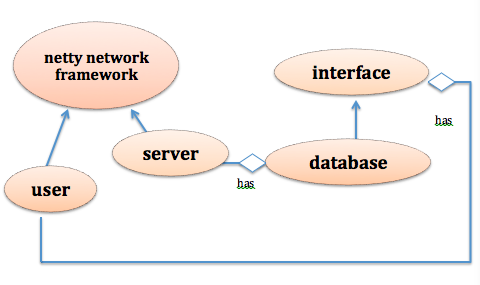
Using the Java language, the design methodology will be object-oriented, meaning classes will be used to represent the various parts of the chat application and each class will have its own methods and data. If an abstract class is used, its subclasses can inherit or override its methods and data for the subclass’s specific usage.

Data from the database will be protected with the encapsulation that is designed into the object-oriented approach. This is in contrast to operation-oriented design, which focuses more on the actions that different parts of an application makes, or a data-driven design, which builds a structure around the data.

Lastly, a graphical user interface will be created to create a user-friendly visual that a user can interact with, without having to understand the underpinnings of the object-oriented design. As an example, if a user presses a button such as “send” when sending a message to their friend, it will initiate the equivalent method sendMsg(), which will take care of sending the message to the friend’s I.P. address, storing the message in the two user’s chat history, and sending an error if the message failed to send.

# System Overview

Using an object-oriented approach, the Babble Chat Application will have five main parts to create the bigger system. Using the netty framework, a user and server class (further detailed in the next section), will be derived. The server has a database in which all of the chat room data is stored. The database is derived from the interface, which will be used to retrieve data and display data as needed (while remaining secure) to the appropriate user. This high-level overview is demonstrated in the figure below:



**Figure 1: High-Overview Class Diagram**

# System Architecture

The following section summarizes the main classes that will be used to create Babble Chat along with UMLs detailing relations between classes, methods and data. These are open to change during the implementation process.

## Netty Network Framework

The netty framework provides very useful classes and methods to create a smoothly working and quickly scalable network connection and channels within the network connection. Using the channel and pipeline capability, two sockets can be opened to send and receive messages between users and quickly display the messages in real-time. The two main classes that will be derived from this frameworks various libraries will be the Server class and User class as follows.

### Server, ServerInit, and ServerHandler Classes

The ServerInit class is derived from the netty ChannelInitializer class. This class creates the pipeline through which data is sent and received. Upon receiving the data, in bytes, the bytes are decoded into a readable string type and, lastly, handled by the ServerHandler Class.

The ServerHandler class is derived from the netty ChannelInboundHandlerAdapter class, which is in turn derived from the netty ChannelHandler class. These provide methods that can be overridden to add and remove channels (each user has a channel). It is within this class that messages can be received from a user and printed to either the Server or User channels.

The Server class provides the port where the server will set-up and run. Upon successful set-up, the server will run until it is intentionally closed. The Server class has a DB (database) object, detailed later.

### User, UserInit, and UserHandler Classes

The UserInit class and UserHandler classes are identically derived from the netty classes as described previously for the serverInit and ServerHandler classes. The main difference of the UserHandler class is that it does not deal with adding or removing channels as that is beyond the scope of the user object. Therefore, it is responsible for printing messages that are received from other channels sent specifically to channels in which the user belongs.

The User class details a specific port (same as the Server port so that the two can actually connect on the pipeline) to connect to, in order to access the chat environment. If the connection is successful (also based on log-in), the user will be able to type and send messages. The user class will have an interface object, detailed next.

**Figure 3: User Classes**

## Interface

The Interface class provides all of the user-facing functionality to generate how the user will see, select, and interact with the chat application. It is the super class to the DB (Database) class so that a user’s friend list, chat history and more can be retrieved or new users can be added.

## DB (Database)

The DB class is the database which will have functionality to create a database table (likely two) to store user data. The first database will be used to contain the username, password and a unique id to represent the user. The second database (if I can figure out how to implement this properly), will use the unique id of the user and a selected friend (and their unique id) to retrieve their chat history.

**Figure 4: Interface and Database Classes**

# Detailed System Design

## Server Class

The Server class initializes a Server object using a parameterized constructor that takes a specified port (likely 8000 or 8080, though the client can change this as desired) to generate a main server for the chat environment to exist on. Outside of the main() method that creates the new object, there is only one other method detailed next.

### run()

This method creates several new objects that are very standard in netty network and server implementation called the bossGroup and workerGroup of type EventLoop Group. Each of these objects represents a thread. Boss threads create a binding socket and pass them onto their worker threads and this is the beginning of channel creation. Because I could only understand the high-level overview of how netty works, the code is written based on several consistent examples on how to get the server set-up using methods and objects that are already built-in to the netty framework and it follows many of the tutorial examples from the netty.io website.

After the objects are created, a Bootstrap object is created within which the ServerInit object is initialized and the port is binded until the Server object receives a signal to close or shutdownGracefully().

## ServerInit Class

When the ServerInit object is created in the Server class, the initChannel method is run.

### initChannel(SocketChannel chan)

A pipeline object is created within this method, and this is one of the main features that distinguishes netty from java.net sockets. This pipeline object allows for very smooth streamed packets of data to consistently run through the sockets that are set-up on either end.

Netty provides several methods of encoding and decoding. As there was no desire to add increased difficulty by dealing with JSON (JavaScript Object Notation), all packets are dealt with similarly to simpler socket libraries – converting bytes to strings and vice versa (see code below). Finally, a serverHandler object is created in the pipeline, detailed next.

**protected** void initChannel(SocketChannel chan) {

ChannelPipeline pipeline = chan.pipeline();

. . .

pipeline.addLast(“decoder”, **new** StringDecoder());

. . .

pipeline.addLast(“handler”, **new** ServerHandler());

}

## ServerHandler Class

The ServerHandler, as the name suggests, handles the data that is running through the pipeline.

### handlerAdded(ChannelHandlerContext ctx)

If a new user logs-in, the handler will create a new channel for the client. This method will announce when new users have entered the chat room.

### handlerRemoved(ChannelHandlerContext ctx)

If a user logs-out, this method announces their departure and removes the channel that was associated with them.

### channelRead(ChannelHandlerContext ctx, Object msg)

Whenever messages are sent through channels, they are written and flushed out by the server using this method.

## User Class

Users must connect to the same port on which the Server has been set-up. This is done in the User class.

**public static void** main(String[] args) **throws** Exception {

**new** User(“localhost”,8000).run();

}

The host and port are passed in to the parameterized constructor and the run method is called.

### run()

This method is nearly identical to the Server class’s run() method, in that it also sets up a channel using similar objects. The main difference in this method is that there is no bossGroup (this is specific to the Server object and the User class should not have access to this thread). A new UserInit() object is initialized and a stream can start reading data from the User object.

## UserInit Class

When the UserInit object is created in the Server class, the initChannel method is run just like it was in the ServerInit class above.

### initChannel(SocketChannel chan)

This method is identical to the one described above except that it creates a UserHandler() object.

**protected** void initChannel(SocketChannel chan) {

ChannelPipeline pipeline = chan.pipeline();

. . .

pipeline.addLast(“handler”, **new** UserHandler());

}

## UserHandler Class

The UserHandler is responsible for printing messages from the pipeline.

### channelRead(ChannelHandlerContext ctx, Object msg)

Whenever messages are received, they are printed out using this method.

## Interface Class

The Interface Class, as the name suggests deals with the graphical user interface that the user interacts with to use the chat application. This class is closely tied with the Database class, and it may be further integrated with the Server Class depending on how the implementation proceeds. This class is constructed when a new channel is created. Upon initialization, a window will pop-up allowing a user to either register or log-in, as follows.

### register()

If a user selects the “register” button, the register method will be implemented. This allows a user to input their desired username, which must be unique. If the input is not unique, they will receive an error message stating the name is not unique and to choose another. After this is successfully done, the user can create a password. After this is successfully done, the user will be redirected to the log-in screen.

### login(String username, String validate)

The log-in screen will be displayed with username and password fields. When the “Login” button is selected, both fields’ data will be passed as parameters to the login method. This will validate the data by checking the database for matches. If a match is found with both username and password, the user will gain access to the chatroom.

### send(String msg)

When a user wants to send a message, they can input data into the message box and select the “Send” button that will take the data in the message box as a parameter to the send method.

### displayChat()

This method displays the chat window with a screen of the ongoing dialogue, and a second text field below where the user can type a message.

### displayHistory()

If there is a previous history with a specific user/friend, the displayHistory method will show it, otherwise, the chat window will be empty or only contain new dialogue.

### displayFriends()

This will be a list of friends that are available to speak with. If time allows, friends will be highlighted and selectable through a mouse click to start a private chat with said individual.

### logout()

A user can leave Babble chat at any point by selecting the logout button. This will call the logout method that will save any new dialogues with other users and close the user’s session.

## Database Class

The database class is responsible for managing users’ data.

### createTable()

This method will be used to create the database tables (starting with test accounts). The main table, *userTable*, will consist of username, password, and a unique ID for the user. If time permits, a second table, *chatTable*, will be created to store the user’s unique ID, friend’s ID and relevant chat history between the two individuals. Otherwise, this chat history should be stored in the first table.

### addUser(String username, String key)

Whenever a user successfully registers, they will be added to the userTable using this method.

### findUser(String username, String key)

When a user logs-in, they will have to be validated using this method. If a username is found, the user’s password will be checked to see if there is a match. If there is a match, a success of 0 will be returned, otherwise, there will be a mismatch or no account error returned using -1.

### chatHist(int userID, int friendID)

When a user selects another user for a chat, the chatTable will be checked to see if there is a pre-existing chat history. By checking the user’s uniqueID and friend’s uniqueID, a chat should be found, otherwise, an empty string will be returned signaling no previous dialogue.

### addHist(int userID, int friendID)

Finally, when a user logs-out of Babble chat, new dialogue will be added to the pre-existing dialogue by using this method. It should add the new dialogue to the chatTable, appending to any previous dialogue.