Social Media App™

Team 7

Jai Menon, William Boulton, Alan Yi, Mukund Venkatesh, Kush Kodiya

12/8/2024

1 Part 1

1.1 Project Functionality

1.1.1 code implementation

Our project involved creating a social media platform with individual classes to manage various components such as clients, messages, and the server database. Each class had a specific role: creating clients, handling messages, storing user data, creating a GUI, and managing interactions between clients. All classes have their own interfaces.

For user creation, we implemented `User.java`, which handles the creation and modification of User objects. This includes managing friendships, blocks, usernames, and password validation. When a User object is created, it receives a username, password, first name, last name, and profile picture, all of which are strings. Due to the nature of how we organize the user inputted data, these strings cannot contain pipe characters. Validation is performed before user creation. Users are stored in a pipe-separated value (PSV) format.

The `UserDatabase.java` class manages user registration, sign-ins, and friendship/block requests, storing all user data in a file named `users.txt`. When instantiated, the class checks for the existence of this file. It then calls the `load()` method to load users from the file into a private `ArrayList`.

Upon logging into the platform, a client socket is created using `UserClient.java`, which communicates with the server. This class handles all requests to the server using `BufferedReader` and `BufferedWriter` objects for communication. The client cannot access the database directly but can request information through the server.

Messages are managed by the `Message.java` class, which stores details like `messageID`, sender username, receiver username, text content, and picture (if applicable).

Messages are stored in individual files for each user using `MessageDatabase.java`, which handles sending, deleting, and editing messages. Messages are sorted by `messageID`. Each user's individual message files contain all the messages that user has ever sent and all the messages that user has ever received.

Server-side operations are managed by `MediaServer.java`, which runs the main server loop. Each time a client connects, a new thread is created for exclusive interaction with that client. The server processes account creation, blocking, unblocking, and direct messages. Messages are updated every three seconds using a `TimerTask`.

Since multiple clients can interact with the server simultaneously, thread-

ing is essential. For our project we implemented threading differently than how we learned it in class. In some cases, we used lambda functions to create new threads, avoiding the need for separate classes to handle client interactions.

1.1.2 features

This app is created to handle all of the features necessary for a standard messaging app. This includes standard features like sending messages, blocking users, adding friends, viewing users, and handling profile information.

The application stores user information like username, password, firstname, lastname, profile picture, and who is allowed to message a user. All of these are required when you create a new account besides who is allowed to message you. By default anyone is allowed to message a user, when logged in you can change this to friends only from the edit profile section. You may also edit your username, password, or profile picture from this section.

The app includes a user scroll list which you can use to select users to message. There is also a search bar included for convenience. When a user is blocked, they will not appear on this scroll list unless you search their exact username. For convenience, your friends appear at the top of this list.

There are two special features included in this application. The first is picture handling. Besides allowing profile pictures, we also allow sending pictures in messages by clicking the add a picture button. The picture will then display in a preview and when sent the other user can click the message to view the image.

The second special feature is the option of RSA encryption. RSA encryption is a standard public vs private key encryption system where clients and the server broadcast their public key for encryption and keep a private key for decryption. With sufficiently large prime numbers (in our case 512 bytes) it is very difficult for a user to find the private key from the public key. This is due to the nature of the encryption and decryption algorithm and the complexity of factoring large numbers. To create the keys, two large prime numbers are chosen p, q and multiplied together to get the first part of the public and private key $N = pq$. Then to create the public key another number e is chosen. Information is then encrypted using the formula $c = m^e \ \texttt{mod}(N)$ where c is the ciphertext and m is the original text. The receiver then calculates the original text using the following formula: $m = c^d \ \texttt{mod}(N)$. Where d is computed as $d = e^{-1} \ \texttt{mod}(\phi(N))$ and $\phi(N) = (p-1)(q-1)$.

1.2 Design Choices

Our project uses two databases: one for Users and one for Messages. Each user has their own `MessageDatabase`, which ensures they only have access to messages they sent or received. This organization keeps things clean and avoids iterating through irrelevant messages.

This `MessageDatabase` format means that each user has their own file containing every message that they have sent or received. This means the files will be much longer, and require more parsing when choosing to display a user's messages with another user. However, it means the server does not need to create as many files to store messages with users and does not need to traverse multiple files for a single user. The `MessageDatabase` object for each user will require more RAM, but since this is being handled entirely by the server this should not create an issue.

Messages are created without pictures by default, but users can add pictures by calling the `addPicture()` method. This method creates a new picture file, assigns it a unique name, and stores it as a JPG file for memory persistence.

We chose to store data in PSV format instead of CSV to avoid conflicts with commas in messages and byte arrays. The `createUser()` method throws a custom `BadDataException` if illegal characters, such as pipes, are included in the input.

Every client request to the server starts with an enumerated type from `Action.java`, ensuring users can only perform valid actions by clicking buttons rather than sending unauthorized commands.

Our approach to threading uses lambda functions to avoid creating additional objects for each thread. We overcame the restriction of needing effectively final variables by defining a new final `socket` for each user connection. We also used a `TimerTask` to update messages in real time, running every three seconds.

We also choose to display pictures from messages in a different window than the JList that displays the text content of theses messages. This was primarily for two reasons, it is easier to handle the pictures this way, and it creates a less cluttered environment while viewing messages. This means when a messages is sent with a picture, the message will have "(click to view image)" added at the end, indicating that if you view the image by double clicking on it an image will appear.

We made multiple noteworthy decisions while handling the images within our app, but most importantly, we chose to store the images for users and messages in a different format. For users, we chose to store profile pictures directly in the `users.txt` file (which stores the user database) to prevent

creating multiple files for the user-database to read from. This is manageable because users do not have much information to store and there will not be that many users to store profile pictures for (i.e. the relative size of the user database is much smaller than all of the messages). However, for messages we store the picture sent with a message in a separate .jpg file. This allows us to send messages to the client without needing to send the picture over unless they try to view it, and it allows for more organization as images are stored in a different location and their file-path is stored with the message.

We also chose to send the pictures as byte arrays through an object output stream. `ObjectOutputStream` objects are more volatile than buffered readers/writers, but they are more flexible, allowing us to send more than just String objects to the server. This makes message sending more seamless.

Finally, we chose to create the effect of real time messaging by updating your messages every three seconds using a `TimerTask` instead of having new messages sent directly to the client when they appear. We chose to do this because it avoids the possibility of messages being missed when the client is running another operation. To make the `TimerTask` create a responsive real-time messaging feel, we chose to update messages every 3 seconds, which is frequent enough to make the client feel fast but not so frequent that it will send too many requests to the server.

2 Part 2

2.1 Jai Menon:

I had different roles throughout the different phases. During Phase 1, when we were creating the foundations of our social media program, I helped write the class Message.java, which dealt with creating the Message objects and had methods that helped us manipulate existing messages. These were called when the user clicked the appropriate buttons in the GUI. I worked on methods like deleteMessage, and helped write test cases for the class. During Phase 2, when we were implementing Client and Server functionality, I wrote all the test cases for that entire phase. This required me to have a basic understanding of most of the functionality of the code written in that phase. And finally, during the last Phase I mainly worked on the report and visual aid for the final presentation. Using the Docs which we had been writing throughout the entire process of creating our social media platform, I helped write the full report explaining the functionality and our design choice throughout the project. For the visual aid, I was in charge of the Frequently Asked Questions sections. I had to come up with the questions and, during the recording of my section of the presentation, answered them at the end of the presentation.

Throughout the project I dealt with a variety of technical difficulties. There were several problems with my integrated development environments and their ability to pull code from github. I also couldn't properly push code from any of my IDEs to github which severely hindered my ability to help write code in Phase 3. Something I wish I changed was figuring out what the problem was earlier, which would have helped me contribute more efficiently to the project. The workaround that I found was writing code in an IDE to make sure it would compile, and then copying that code over to the website version of github to push my changes. Something else I would change on a second run through of the project is adding a feature that would let users give their friends nicknames. I feel like that extra element of customizability would really give users a better experience on this platform.

2.2 William Boulton:

The features that I focused on the most for this social media app were message handling, server creation, and the main GUI. For handling messages, I created two files, Message.java and MessageDatabase.java. In the first, I created the message object that would contain information on message id (which is a counter meaning each new message will have a higher id than the previous),

sender, receiver, text content, and picture content. To ensure that we could recreate old messages and make new ones without having trouble with the ID value, I created two constructors to handle re-instantiating messages or creating new ones. To handle how we store messages and allow the server to make modifications to messages I created the MessageDatabase.java file. This class stores all messages sent and received by a user in a file with the filename "username.txt" (where "username" is the user's unique username). I decided to store messages in this manner to reduce the number of files stored on the server, as creating a file for each thread between two users would require many more files. It would also force the server to read a different file more often because with this infrastructure we are able to retrieve all of a user's messages from a singular file, without having to traverse different files when the user chooses to message someone else. However, this does mean that a singular MessageDatabase object (containing a user's messages) requires much more memory than one based on only message threads. In server creation, I created the main run method for the server and the method that handles message operations. I also created the thread instantiation system that uses a lambda function with a final client variable passed in arguments to start the run() method on a new thread. This avoids creating a new Network object for each client that connects and makes the MediaNetwork.java file simpler. However, it does require all parameters passed to the run method to be effectively final, so we had to carefully handle processing in the main method to allow creation of new threads. Finally, I worked on the primary GUI for user interaction. This GUI (in GUIClient.java) handles viewing messages and altering them, viewing your messages with different users, sending messages, altering your profile, and adding/removing friends or blocked users.

If I were to go back and change what I contributed to the project, I would have spent more time planning in phase 1. I believe if I spent more time planning how we would handle messages and the message database in later phases, I would have been able to design a more robust message handling system rather than needing to go back and make adjustments as the project continued. I may have considered changing the message storage system to be files based on communication threads between two users, however, I feel the system we have currently is not bad. One thing I would not change is documentation, I spent a lot of time throughout each phase creating documentation of our project and organizing it neatly. I feel this helped the team understand what each aspect of the project was doing and allowed us to use each other's code more effectively without needing to go figure out how to use classes that were previously created by someone else. The main thing I would change if I were to start over is putting more emphasis on test-cases as I believe we did not focus on those enough in the first phase, making testing

in the second and third phase more difficult.

2.3 Alan Yi:

In this project, my main focus was on threading, the server side, and the sign-in page GUI. To make the program thread safe, I created a static final Object variable to synchronize certain vulnerable code. In the server file, I worked on the code for the periodic refreshing of direct messages through a TimerTask that runs on a separate thread. For example, the messages will be refreshed every 3 seconds so when a sender sends a message to a friend, the receiver will see the new messages after the messages are refreshed. The longest delay between sending the message and viewing it is 3 seconds. I also developed the handling of client requests that are linked with the user database, such as blocking or adding a user as a friend. Because of our universal message-sending format with the parameters separated with a "|", this was simple as I just needed to parse the string and send them into their respective function in the User class. In the third phase of the project, I focused on building the GUI where I created the login and sign-up page. Because of my prior experience in front-end development, the appearance of GUI is somewhat appealing and effective, allowing the users to create a new account or sign in with an existing one. If an issue occurs when signing in, an error message would pop up.

If I were given the opportunity to start the project all over again, I would spend more time double-checking and looking over our work after we submit it. A lot of the points that were taken off were because of careless mistakes, many of them on my part, because I didn't read through the code and instructions carefully. For example, after I created the threading for the user database, the thread-safe methods got overwritten a week later when we were about to submit. However, because I didn't go back and check my work, I didn't realize it was missing, and we lost several points. In terms of the actual program, one thing I believe we could have improved is not requiring the user to input a profile picture when signing up. Instead, we could utilize a default picture, and the user can change their picture in their profile settings. This would mimic real massaging apps and make creating an account much simpler.

2.4 Mukund Venkatesh:

The aspects of this project I focused on were User handling, client creation and the image handling in the GUIs. To handle users, I created two classes, `User.java` and `UserDatabase.java`. In User.java, I create a user object

whose primary role is creating and storing information pertinent to users. This includes two constructors, one for loading a User from the database, while the other is for creating a new user. Each user has several pieces of instance data: username, password, first name, last name, a list of friends, list of blocked users, a profile picture, and a toggle for allowing messages from all people or only from friends. These users' information is stored in a database, `users.txt` in the format given above. Friends and blocked users are stored in `ArrayLists`, while the profile picture is stored as a `byte[]` for easy access and simple storage. This database is managed and accessed by the other file I made, `UserDatabase.java`. This class manages user data and operations in the project. From user account creation, login verification, profile management (including friends and blocked users), and storing profile pictures. This class also handles writing and updating the database file `users.txt`. To let users communicate with the server, I also created the `UserClient` class. This class implements the client-side functionality for a user interface in the application. It manages tasks such as connecting to the server, user authentication, sending and receiving messages. The client can sign in existing users or create new ones, manage friends and blocked users, and send messages with text and/or images. It communicates with the server with socket connections and `ObjectInputStreams` and `ObjectOutputStreams`. I created this class to also include methods to edit, delete and view messages, while preventing message overflow by limiting their size. I also handled image display and profile picture editing in the GUI, through the `JFileChooser` class seamlessly integrating with our project.

If I could go back and change what I did and how I did it, I would've focused more on organization and planning when we started. If I had done so, we would have been able to integrate users and messages more effectively and prevent a lot of errors in later phases. We could've had a better user profile system that did not require constant change as we moved with the 2nd and 3rd phases. For example, the user profile pictures and message images are stored different due to different ideas between teams, and communication and planning in that regard would have streamlined the process. However, the system we have in place isn't detrimental to the efficiency and ease of use of the project. The aspect I would not change would be the user client. The client's methods are simple and effective in their role of providing and receiving valuable information to and from the server. If I could change only one thing, it would be the emphasis we put on test cases. We paid them no mind until close to the deadline, and then realized how intricate they were to design and had to take valuable last minute time to write and evaluate them.

2.5 Kush Kodiya:

During Phase 1 I worked on storing users for our program. This task included creating a User class, creating a UserDatabase class, making methods for the classes, and making testcases for the classes. The User class had a couple different methods that I implemented including features such as add friend, block user, unblock user, and the creation of a new user. The UserDatabase class was a database for the users. In Phase 2 I worked on the client for our program. This client allowed us to connect to the server and send data back and forth between the program and database. The client sent user sign in data to the server and sent opened messages and it received messages from the database. In Phase 3 I did the testing and worked on the presentation aspect of our project. For testing in phase 3, I mostly manually tested the program. While developing our GUI I tested each feature with extensive stress testing. The features of our GUI include profile pictures, messaging, blocking users, sending images, and a sign in system. I tested the sign in system, the profile picture system, the image sending system, the message sending system, and the blocking system. I also created the slideshow for our video.

If I could redo the project, I would change a couple things but keep everything mostly the same. In the 1st phase I would try to improve on how we organized the project and I would try to plan ahead rather than thinking of the assignment as 3 different things. This would improve how we handled images and messages because the way we have it right now is very confusing and took a lot of changes in the 2nd and 3rd phases. We could have saved a lot of time and effort by planning ahead and organizing our project better. Another thing that got impacted by a lack of organization was our testing. In the 1st phase we didn't pay attention to these because we didn't see the importance but when we needed them in the 2nd and 3rd phases, they were lacking and it took us extra time to be able to test our program.