Internet Technology

Class DHI2V.So

Authors: Alexandru Crudu (524145), Biholar Matei (507512)

Contents

Introduction

This is documentation related to the chat application that we had to develop for Internet Technology course which is part of the curriculum for year 2 in Saxion University's ICT program.

Client-Side

The client of the system represents a user of the chat application. They have access to multiple functionalities which allow them to communicate information to other users using a server.

The following classes were used to implement the user functionalities:

- Client2: This is the class that contains the “main” method that a client needs to run in order to access their functionalities. In here, instances of the other classes that help define client functionalities are created and their needed methods are called when necessary. When ran, this class creates two threads, one that constantly checks for new messages and handles each of them and one that asks users for their input and then sends their commands to the server. Aside from this, on its own the class is able to handle basic functionalities such as sending a “PONG” message back to the server. Some of the more complex functionalities, such as handling surveys are implemented in other classes that “Client2” creates instances of and uses when necessary.
- EncryptionHandler: This class contains all methods and variables that are needed for handling the encryption functionalities. These include: handling a received public key from another user, handling a received session key, sending and decrypting messages.
- FileTransferHandler: This class contains all methods and variables necessary for handling the file transfer functionalities. These include the following: requesting a file transfer, handling a file transfer request, handling an accepted request, handling a rejected request and listening for new messages related to file transfering.
- MessageToClientHandler: This class contains all methods and variables necessary for sending broadcast and private messages.
- SurveyHandler: This class contains all methods and variables necessary for handling survey functionalities. These include: requesting the possibility to create a survey from the server, adding the questions and answers, choosing the users and sending a message containing information about both the questions and answers and the users invited to the survey.

Together, the mentioned classes give users access to 9 main functionalities, that they can choose from by using a console printed menu:

- Sending a broadcast message
- Sending a private message
- Receiving a list of all users that are online
- Creating a survey
- Filling in a survey
- Sending files
- Accepting/Rejecting file transfer requests
- Sending encrypted messages
- Closing the chat application

Sending a broadcast message

This is the first functionality listed in the user menu and it is pretty straight forward. When a user wants to send a message to everyone connected to the chat application at a given moment they need to send a broadcast message. After selecting this functionality from the menu, the user is asked to type in their

message. Its contents are then sent to the server and the client is then informed if everything went through correctly. In this case, a confirmation message is displayed in the menu.

Sending a private message

The second functionality of the menu allows users to send private messages to a user of their choice. After choosing this from the menu, a user will first be asked to type in the exact name of the person they want to reach. They can see all available users by viewing the list of users. After they choose a name, users are then asked to type in the exact message they want to send. After this is done, they private message request is sent to the server. In the case in which everything went well, the user is informed and a confirmation message is displayed in the console. However, if they chose a user that does not exist or tried to send an empty message, they will get an error message.

Requesting a list of all users that are online

The third functionality allows users to see which other users are online at a given time. After they request this information they will either be able to see the names of all who are connected, or be told that they are the only ones online at that moment is there are no other users.

Creating a survey

The forth functionality allows users to create a survey and send it to other users of their choosing. Since there are certain conditions that would not allow for a survey to be started, a user first needs to request permission from the server to create a survey. If that is not possible, they will get an error message, informing them of the reason of failure. If it is possible, the user will then be able to introduce the contents of their survey. For this, a new thread is created in “Class2”. This is because in order to start a survey, a user must first wait for the response of the server telling them they are allowed to do it. Listening for a response takes place inside a listener thread, which needs to be prepared for new messages at all time. However, since creating a survey requires user input, the listening thread would be prevented from catching incoming messages if we were to handle the survey creation inside of it. Using the sender thread is also not a possibility as the handling of a server response, in our case, the handling of the response that allows us to start a new survey is done in the listening thread. Thus, after sending a survey request, we make the sender thread wait. After getting an approving response from the server, we create a new thread that handles the creation of the survey. The reason for temporarily incapacitating the sender thread is because it requires user input, just like the new thread does. If two threads that require user input run at the same time there will be errors. At least that is what happened in our case, before making the sender thread wait. After the survey information is sent by the user or if any error messages are given when requesting the survey creation, the sender thread is released. If everything went through correctly, a confirmation message is displayed in the console menu. A user is also informed if:

- A survey the were invited to was created
- A survey they were not invited to was created. Though this might seem harsh, this was done to let them know they will not be able to start surveys until the ongoing one ends.

5 minutes after the survey was started or after all invited users have filled in the survey, the statistics of the survey are displayed to everyone connected to the chat application.

Filling in a survey

The fifth action allows users to fill in a survey they have been invited to. After hitting this option, they will be presented the first question and all possible answers for it. After they make a choice, they will have the option to do the same for the second question if there is one. This goes on until all questions are answered. If everything goes correctly, a confirmation message will pe displayed in the console menu. If a user tries to submit a survey twice, they will be prompted with an error message.

Sending files

The sixth option allows users to send files to one another. When choosing this option, users are presented with a series of documents they can choose from. After selecting the number associated with the document of their choice, users are asked to type the name of the person they want to send the file to. In the case in which no user with the chosen username is connected to the chat application at the moment, an error message will be displayed in the console. If everything goes fine, a confirmation message is displayed.

If the previous request is accepted by the receiver, the sender creates a new socket connection on a port different from the one that is used for the main protocol functionalities. This new port connection is only used for handling the transfer of bytes from one user to another. DataOutputStream is used for sending bytes across clients and servers. Each time bytes need to be sent, the the DataOutputStream first writes an integer value representing the number of bytes the following messages has. This informs the DataInputStream that is used for reading the sent bytes how many bytes it will have to read next. When doing a file transfer, the sender forwards the following information to the server: transfer UUID, the name of the file, the checksum of the file and the file itself. Moreover, the first message it sends is one that informs the server they are a sender in a file transfer which helps the server handle their request after accepting the new socket. All of the previously mentioned information is sent as bytes. If the transfer goes correctly, a confirmation message is displayed in the sender's menu. However, is the checksum received by the other user does not match the one that was sent by the sender, an error message is displayed.

For the file transfer part it is worth mentioning, that it only works for files that do not take up more than INT_MAX bytes. That is because we attempt to put all file bytes inside one byte array that has the maximum capacity of INT_MAX. In order to make our application also handle larger files, we should have created a buffer byte array in which we read a certain amount of bytes at a time and then write that array using the DataOutputStream of the sender's socket. This process should be done repeatedly, until all bytes all of the file have been read. On the receiver side, each new byte array should be read. Those bytes should be combined together and used to build a file, namely the large file the sender initially tried to forward.

Accepting/Rejecting file transfer requests

The seventh functionality gives users the option to view file transfers they have received and choose whether to accept them or not. Once a sender sends a transfer request, the receiver gets a notification about it. After accepting a request, the receiver application creates a new socket connection, that connects to the same port as the sender does and generate a UUID. They use the DataOutputStream of the new socket to write a message that contains the mentioned UUID, as well as information that tells the server they are a receiver.This client then uses the regular protocol to communicate to the sender that they have accepted their request. A new thread is started that checks for any changes to the new

socket's DataInputStream. When changes occur, we know that we have received new information from the sender. We know the order in which the sender information is sent, that being: file name, checksum, uuid and the file itself. Since the types of information mentioned earlier each require a different way of being handled, we store a variable called "messageType", which is initially set to 1. When its value is 1, we know we must handle the name of the file and then change the value of the variable to two. This number is associated with another type of information, which we handle in a different way. We store information about the file name, checksum and uuid in local variables. When reading the file bytes, we generate the checksum for it using the same algorithm that the sender used for encrypting. If the newly generated checksum matches the one sent by the sender, then the file transfer can be called a success and by using the regular protocol, we send a message to the sender, informing them of the good completion of the transfer. Moreover, the received file is saved inside the "downloads" folder. However, if the checksums do not match, a message is sent to the sender telling them of the transfer failure.

Sending an encrypted message

The eighth functionality of the system allows users to send encrypted private messages to one another. If user 1 wants to send a message to user 2, they must first agree on a session key they will be using for their communication.

When the first user wants to start an encrypted communication with the second, they send their public key. The public key of the first user is first converted into a byte array which is then converted into a String value. That value is then sent to the other user, which is able to get a PublicKey object from that String. The second user generates a session key, which they then encrypt with user 1's public key and send it back to the first user. That user can then use their private key to decrypt the session key, which they can then use to send messages to the second user.

After choosing option 8 and the name of the person the user wants to send an encrypted message to, the session key exchange process is ran automatically if the two users do not already share a session key. After that, the sender is prompted to type the message they want to send to the other side. That message is then encrypting by using the session key and a byte array is obtained. That array is further converted to a Base64 String representation of the encrypted message. The encrypted message is decrypted on the receiver's side by using the session key and the plain text contents of the message is displayed to that user.

Closing the chat application

The last functionality of the user allows them to close their application. This closes their socket connection, thus they will not show on the other users' list of online participants. Moreover, other users will not be able to messages or invite the disconnected user to any activity until they next log in.

Server-Side

The server represents a central place which is responsible for the communication between clients. It is an intermediary used to exchange messages between all the participants of the chat application. It also allows clients to connect to the application in the first place.

The following classes were used to implements the server functionalities:

- Server: This is the class that sets up the connection of clients and start the server itself. It keeps track of the port that the server is running and also a port which the server uses to listen for file transfer requests. Upon running the class, a server socket and a file socket gets created, which run on the respective ports. Also, two threads are instantiated, one for the listening of connection of clients and one for listening for file requests. There is need of a new thread for each client that connects to the server, because the server needs to ensure that multiple clients can be managed concurrently. If we only used a thread, then each client would have to wait for the request of another client to get handled. In other words, only one client would be able to perform an action at a time; the other ones would have to wait, which is not how a chat application works. For file transfers, we need a new thread for each file transfer process for the same reason. The server needs to allow multiple transfers to be done at the same time, in parallel. If the same thread was used for all transfers, then only one file transfer could be possible at the same time.
- ClientHandler: This class is a means to handle the communication between one client and the server. It allows the server to handle each client connection individually and ensures that messages that are supposed to refer to one client do not interfere with messages related to other clients. This class implements the “Runnable” interface, because we pass an instance of this class as a parameter when we create a new Thread for each client connection. In this way, the server can manage everything that is specific for a single client in one place. It is also easier for the programmer to build and maintain the application, because everything is done from the point of view of one client, even if it is server-side. This class is where all the logic is done. When a client performs an action, usually it is through the use of a protocol, in other words, predefined messages that are handled by the server and as a result of reading that predefined message, the server knows what the intension of the client was. It is important to mention that all communication is done through this class, direct communication between two clients is impossible.
- FileHandler: This is the class that is responsible for handling file data and transmitting it from one client to the other client. Each client has input and output streams that allow for data to be read from and written to. This is how bytes of a file is taken from one client and given to another client. This class ensures that the input and output streams of two different clients “find each other”, meaning that it makes sure that data is read from and written to the right places.
- AuthenticationHandler: This class handles all logic that is related to authenticating clients. It makes sure that when a client connects to the server, the server does all the necessary checks, such as username format or whatever a client is trying to connect when in fact he was already connected, etc. When a client successfully connects to the server by login in, this class makes sure to send a confirmation so that the client knows that it has indeed connected to the application. In the case that the client did an error when trying to type it’s username or any other unallowed authentication attempts are encountered, the server sends back a fail message, which then is read by the client application and lets the user know what is the problem.
- ServerEncryptionHandler: This class represents the logic behind the encryption process. When a client wants to send an encrypted message, the application uses a combination of the RSA and AES algorithms to handle that. RSA is used to exchange a session key through the use of a public/private key pair. AES is used to symmetrically encrypt and decrypt messages, meaning that a session key is kept by both clients. This class ensures that the public key is sent to the

right user when a user wants to initiate an encrypted message. It also makes sure that the session key is transmitted from one client to the other and finally, it understands when an encrypted message is sent and to whom it needs to be forwarded to.

- ServerFileTransferHandler: This class handles communication between clients in regards to the status of a file transfer. It receives the request of a client when it wants to initiate a file transfer and it lets the other client know that someone wants to transfer him a file. Then, the receiving client has the choice of accepting or declining the file transfer. This class makes sure that the two clients get the proper responses back and forth so that they are aware of the status of the exchange. So, when a client accepts or declines, the server lets the other party know of the client's decision.
- ServerPrivateMessageHandler: This class is used when a client wants to send a private message to another client. When that happens, this class reads the message and it forwards the message to the specific client that the sender wanted to reach to. This is where error handling in regards to this action is done. So if the sender wanted to send a message to an insisting client or it has sent an empty message, the message does not arrive on the other side, but instead, the sender gets notified of the problem.
- ServerSurveyHandler: This is the class responsible for the logic that is needed to implement the survey functionality in the application. When a client wants to send a survey to other clients, this is where the server performs all the logic that is needed to forward the survey to the right clients and also keep track of the results of the survey that are submitted by each client when they are done answering the questions. As soon as all clients are done or more than 5 minutes have passed, the server communicates the results of the survey to all involved clients.

Handling a broadcast message:

When a client wants to send a message to every connected member of the chat application, he uses the broadcast message functionality. The server gets a message complying with the protocol for this specific functionality. The server immediately lets the sender know that his message has successfully been sent. That being said, the server knows what the actual message is and from whom it came from. With this information, the server can then see who else is connected to the chat application at that moment and forward the message to all users besides the one who sent it. When sending the message to all other users, the server uses the respective part of the protocol where it specifies from whom the message comes from so that all receiving users know the origin of the message.

Handling heart bit

This is a mechanism used to check whatever a client is connected to the server at a given moment in time. This ensures that the communication between the client and the server is still ongoing and whenever the connection is lost, the server knows about it (or at least it finds out soon). In this specific chat application, the client sends a "ping" message to the server, who then responds with a "pong" message to the client, letting him know that the ping has been received. This is done every 10 seconds. In this way, the server rechecks what client is connected to the server and who isn't every 10 seconds, ensuring that it can take action whenever a connection unexpectedly gets interrupted. The Ping and Pong messages are received and sent according to the protocol.

Handling a private message:

When a client wants to send a message to one specific other client that is connected to the chat application, he uses the private message functionality. The server gets a message complying with the

protocol for this specific functionality and reacts accordingly. The first action taken by the server is sending a confirmation message back to the client so that he knows that the message was sent successfully. The server gets information about who the sender is, who the receiver is and what is the actual message. By doing so, it can then perform the action of forwarding the message to the right client according to the protocol.

Handling a user list request:

When a client wants to see what other users are connected to the server at a given moment, they use the “see list of other users” functionality. The server receives a message complying with the protocol that lets the server know that a user requested to see a list of all connected users. In doing so, the server can then react accordingly and send the list of all connected users back to the client according to the protocol. This request is only one message away, because the server knows at all times who is connected to the chat application.

Handling the creation of a survey:

When a user wants to send a survey to other clients, he needs to know who else is online so that he knows who he can include in his survey. So, the first action that the server takes is sending a list of connected users to the client upon his request. Then, the client sends a message according to the protocol that contains the content of the survey and a list of all participants. When that is done and the message from the client gets read by the server, the server immediately reacts by letting the creator of the survey know that the survey has been successfully created. At this point, the server has access to the following information: who the creator of the survey is, what are the contents of the survey and who should be the participants of the survey. This being said, the server is now ready to send a message to all participants of the survey, letting them know who the creator is and what are the contents of the survey.

Handling results of the survey:

When one client is done completing the survey, the server gets a message according to the protocol containing the answers of that specific client. The server keeps receiving messages like this when each client submits their results and it waits no longer than 5 minutes. When every participant has completed the survey or more than 5 minutes have passed, the server closes the survey and calculates the results. Because the server keeps track of all answers submitted by each client, by the end of a survey, it has information about the number of chosen answers for each individual question. The server uses this information then to send a message to all clients in the chat application containing the results of the survey so that everyone can see what answers have been chosen and by how many participants.

Sending Files:

When a client wants to send a file it uses the “send file” functionality to let the server know about his intention. The server gets a message containing the following information: the file name that the sender intends to transfer, the file size, the receiver and a unique identifier that is associated to one specific file. Upon receiving this information, the server immediately lets the client know that the file transfer request has been successfully sent. Then, the server forwards this request to the receiver. The message sent to the receiver holds the same information that was intercepted by the server, with one modification, it includes the username of the sender so that the receiver knows who wants to transfer

him a file. After this, the server can get one of two messages from the receiving client. Either an accept message or a reject message. An accept message lets the server know that the receiver has accepted to receive a file from the sender. In doing so, the server reacts accordingly by letting the sender know that the receiving client has accepted a file transfer and that he is ready to upload the file and start the transfer. When that happens, a new socket gets created from the client side which gets connected to another port on the server side. That other port is where file transfers are handled. This needs to happen on a different socket connection, accepted on a different port, so that the process of file transfer does not interfere with the other communication between clients. When a file is sent, a significant amount of time might be needed to complete the transfer, so that is why this needs to be done in the background, not blocking all of the other functionalities of the chat application. Each file transfer is done in its own thread.

The actual transfer of the file on the server-side is done by reading the data from the input stream of the sender client and writing it to the output stream of the receiving client. Everything is written and read as bytes from the streams. The server reads data piece by piece from the input stream and it knows the order that data is sent through. So, the first thing that the server expects to read from the input stream is data about the file name. The server needs to know the number of bytes that it needs to allocate for the file name, so the first piece of data in the stream, is the length of the file name in bytes. Now that the server knows how many bytes the file name will contain, it is ready to read the actual filename. This process is repeated for all data that is read from the input stream by the server. First the number of bytes, then the actual data. The next thing the server expects to read from the input stream is the checksum of the file (it repeats the same thing with the number of bytes first, then the actual checksum). After this, the uuid of the file is expected and then the file itself. (P.S Everything is read in memory as the application is built at the moment, but, in order to send large files, the application should read a small portion of bytes and send them to the client, chunk by chunk. This is more memory efficient and it does not allow the server to run low on memory.) After everything has been read, the server writes this data to the output stream of the receiver so that he can access all this data. After the receiver finished reading all the transmitted data, the server makes sure to let the sender know that the file transfer was successfully completed. It can also happen that the checksum in the message sent by the sender did not match with the actual checksum of the file. In that case, the server gets notified by the receiver and then lets the sender know that there was a mismatch and the file transfer did not succeed.

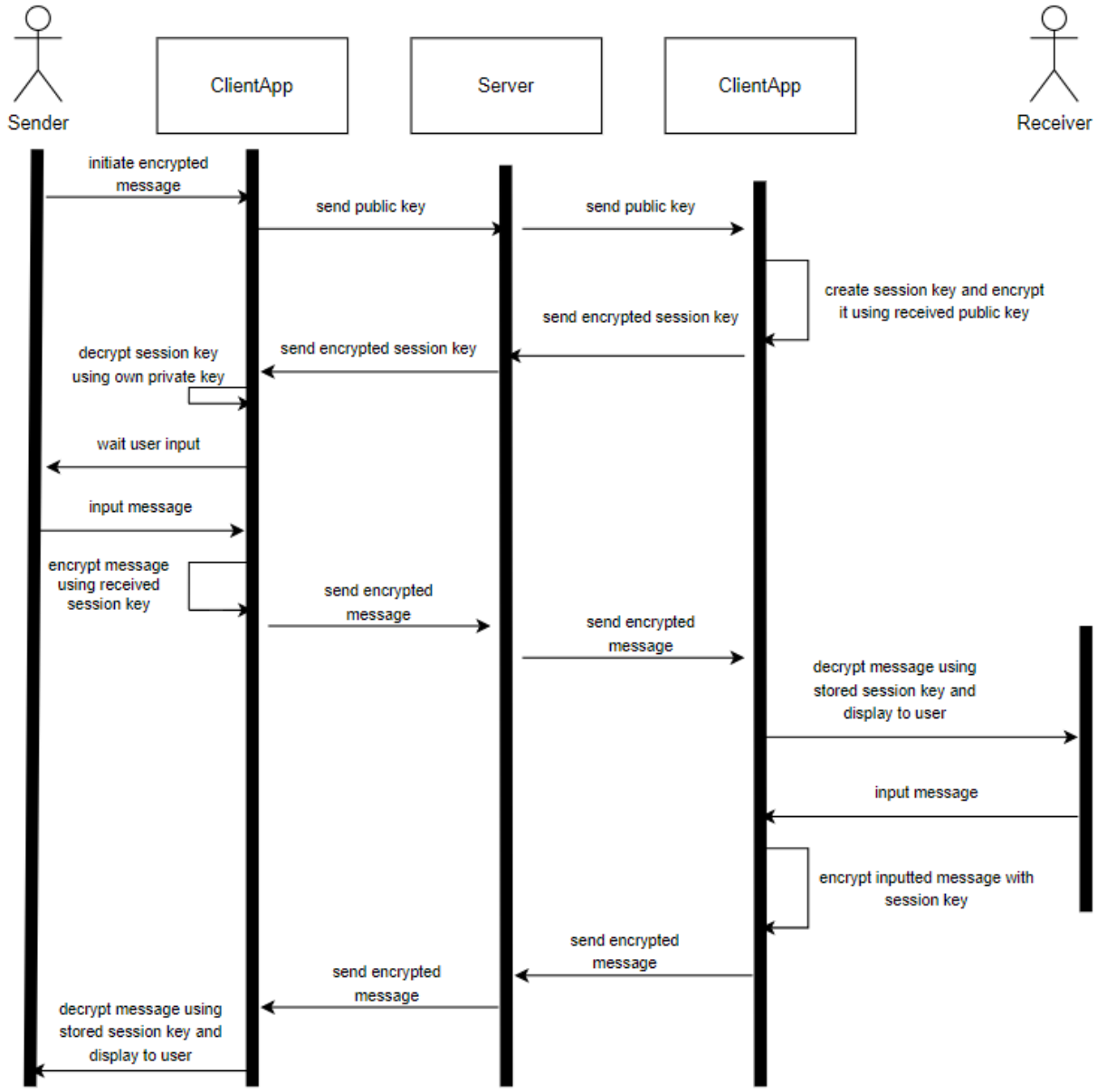
In case that the receiver rejected the file transfer request, the server gets notified about the receiver's decision. The server then forwards the decision of the receiver to the sender, letting him know that the file transfer will not go through.

Handling an encrypted private message:

When a client wants to send a private encrypted message to the server, he first sends the public key. The public key is needed so that the other client can use it to encrypt a session key, which will be later used by both clients to continue communicating with each other. When the server receives the public key according to the protocol, it forwards it to the other client. The next steps that the server does is receive a session key from the receiver and forward the session key to the sender. Now that both clients have the session key, which is the same on both sides, the server receives messages that are encrypted

by the session key. With the use of the protocol, the server can distinguish when the messages are encrypted, and it can safely allow the exchange of messages between clients.

Sequence Diagram of the encryption process:



This diagram points out to all the actions that need to be performed by the actors of the system in order to complete an exchange of encrypted messages between two clients. All the steps have been described in detail in the above documentation.

Handling the closing of application by a client:

When a client wants to disconnect form the chat application, the server receives a message from the client complying with the protocol, which allows it to react accordingly. The server closes the socket connection of that client and updates all of the information that it keeps track with involving that client.

Tests

In order to check the good functioning of the protocol we have implemented, some unit tests were created. In total, there are 10 test classes, each one handling different parts of the protocol:

- AcceptedUsernames: This class of tests was already given to us and it tests the good functionality of our server's filtering of the usernames that are not allowed, as well as allowing the usernames that meet the standards imposed by the assignemnt.
- EncryptionTests: This class of tests was designed by us and it serves the purpose of testing whether all functionalities related to the encryption part of the assignment work as intended.
- FileTransferTests: This is another example of a test class designed by us, this one serving the purpose of testing the good functionality of all file transfer related functions.
- LineEndings: This test class was given to us already and it checks whether or not our server is able to handle multiple different line endings placed at the end of request messages.
- MultipleUserTests: This test class was given to us and it checks the good functionality of the application in the case in which multiple users are connected to it at once. A thread.sleep method was added to one of the tests in between two users logging in one after another. For the tested functionality it is important that user1 logs in before user2, however that does not always happen, thus we found ourselves in the situation where this test sometimes works and sometimes it does not. The method thread.sleep() was called to ensure the order in which users log in always statys the same.
- PacketBreakup: This test class was already given to us and it tests if the response is the expected one in the case of multiple calls to the flush() method.
- PrivateMessTests: This test class was created by us and it checks whether or not all functionalities related to private messaging work as intended.
- RequestListTests: This test class was designed by us and it checks if all functionalities related to requesting a list of users work as intended.
- SingleUserTests: This test class was already given to us and it tests the good functionality of all features that invole a user. These refer to logging in correctly, not being able to log in multiple times and only sending "PONG" when necessary. The "PONG" related test was slightly modified by first authenticating the user, as our protocol does not allow users to do anything other than authenticating themselves before logging in.
- SurveyTests: This test class was designed by us and it ensures that all functionalities related to surveys work as intended.

For the next part, a brief explanation will be given for each of the tests we have designed. The given tests will not be explained.

EncryptionTests:

- sendingPublicKeyArrivesAtTheOtherUser: This test checks if the public key sent from one user is successfully received by the intended receiver who is connected to the application.
- receiverSessionKeyArrivesAtTheSender: This test checks if the session key generated by the receiver successfully arrives at the sender.
- sendingPublicKeyToUnknownUserReturnsError: This test checks if the correct error message is returned by the server when trying to send a public key to an unknown user.
- sendingEncryptedMessageReturnsOk: This test checks if a user is able to successfully send an encrypted message to another user who is connected to the application.
- sendingAnEncryptedMessageToAnUnknownUserReturnsError: This test checks if the correct error message is returned when trying to send an encrypted message to an unknown user.

FileTransferTests:

- FileTransferRequestReturnsOk: This test checks if the correct response is returned from the server when a user tries to do a valid transfer request.
- TheReceiverReceivesTheFileTransferRequest: This test checks if a valid transfer request is successfully received by the intended receiver who is connected to the chat application.
- SendingAFileTransferRequestToAnUnknownUserReturnsError: This test checks if the correct error message is returned in the case in which a file transfer is made to a user that is not connected to the application.
- AfterReceiverAcceptsTheRequestTheSenderSeesTheAcceptForwardedByTheServer: This test checks if the sender is notified after the receiver accepts the transfer requests.
- AfterReceiverRejectsTheRequestTheSenderSeesTheRejectForwardedByTheServer: This test checks if the sender is notified after the receiver rejects the transfer requests.
- AfterReceiverDownloadsTheFileTheSenderSeesAFileSuccess: This test checks if the sender is notified after the receiver successfully downloads the file.

FileTransferTests:

- PrivateMessReturnsOk: This test checks if the correct response is sent by the server when a user makes a valid private message request.
- PrivateMessReceivedWorks: This test checks if a private message successfully arrives from a user to the intended receiver who is connected to the application.
- SendPrivateMessWithEmptyStringReturnsError: This test checks if the correct error message is returned in the case in which an empty message is sent by a user.
- SendPrivateMessToUnknownUserReturnsError: This test checks if the correct error message is returned when a user tries to message an unknown suer

RequestListTests:

- RequestListReturnsListOfUsers: This test checks if the correct list of users is sent after multiple users log in
- RequestListWhenYouAreTheOnlyOneOnlineReturnsNoUsernames: This test checks if the correct response is sent in the case in which the requesting user is the only user online.

SurveyTests:

- RequestSurveyReturnsOkSurveyWithListWhen3UsersAreOnline: This test checks if a valid survey request is approved by the server.
- InitiateSurveyWhenLessThan3UsersAreOnlineReturnsError: This test checks if the appropiate error message is returned when trying to start a survey with less thatn 3 users onine.
- StartingANewSurveyWhenThereIsAlreadyOneOngoingReturnsError: This test checks if the appropiate error message is returned when trying to start a survey when there already is an ongoing survey.
- ReceivedSurveyByEveryoneWorks: This test checks if all users invited to a survey receive its correct details.
- CreatingSurveyReturnsOk: This test checks if the correct response is sent by the server when trying to create a valid survey.
- SubmittingSurveyDetailsReturnsOk: This test checks if the correct response is sent by the server when users do a valid survey submission.
- AfterAllUsersCompleteSurveyEveryoneGetsStatistics: This test checks if the statistics are correctly sent after all users invited to a survey finish filling it.
- SubmittingASurveyTwiceReturnsError: This test checks if the correct error message is returned in the case in which a user tries to submit a survey twice.

General notes about the tests:

- The thread.sleep() method was used in multiple places, being placed between two users log in as in some cases it mattered which of the users was the one to log in first. Without the use of thread.sleep(), the order in which users log in would sometimes differ, which affected our tests.
- The @AfterEach method in most of our test classes first sends a “QUIT” to the server before closing the socket connection. This was done because for a reason we were unable to discover, just closing the socket was not enough to get the tests running properly. All test would work individually, but when running multiple tests at the same time, all except for one would fail. By adding the println(“QUIT”) methods, that problem was solved.
- The server should be ran again every time after testing an entire test class.
- For reasons we were unable to discover, sometimes a few tests do not work, when trying to run an entire test class at once. If this problem occurs, try restarting the server and running that test individually. All tests should work correctly if ran individually right after restarting the server.

Final team Reflection

Situation

For the Internet Technologies course of our programme we were tasked with creating a chat client application by implementing a server and a client application that people could use in other to communicate with one another by using our protocol.

Task

As previously mentioned, our task was to create a chat client application. The functionalities of the application should have are the following:

- Sending broadcast messages
- Sending private messages
- Seeing a list of all active users
- Creating a survey
- Filling in a survey and submitting it
- Sending a file to other users
- Accepting/Rejecting a file transfer request
- Sending an encrypted message

The application consists of two main parts: the server and the client interface. The former has the purpose of transforming our designed protocol into code. For that we needed to make sure that all incoming requests are handled accordingly and that the correct information is sent to the clients based on either their request or another user's request that involves them.

The client interface needed to contain a menu that people could use to access the functionalities of the protocol in a user-friendly manner. In order for this to be possible, some logic had to be implemented, that connects each user to the server, handles user input and sends the right requests to the server based on the users' choices and last but not least, correctly handles the messages sent from the server.

Actions

For each functionality, we did some research and brainstormed on how the specific functionality should work and also thought about the protocol. First we implemented the protocol, made sure that it works and then we continued with working on the code. Mostly it worked as planned, but sometimes we needed to change the protocol as we realized some steps were either extra or missing. We encountered various obstacles throughout the project, but through research and spending hours and hours of trying to come with a suitable idea, we managed to complete all tasks.

Results

The chat application is complete. Almost all functionalities are well-implemented. Some messages that were intended for the user could have been more descriptive and more user friendly. Also, the file transfer is not perfect, it does not allow very large files. We know what was the problem, but we ran out of time. We learned a lot from this project, a lot of new technical things in software engineering, as well as how to work as a team. We improved our skills when it comes to working with Java, building a complex codebase and applying programming conventions in order to have a quality product.

Reflection

Even though the application is mostly well done, there are still many things we can improve on. Keeping the code clean and not postponing procedures that keep the code readable and maintainable is one of the biggest things we've learned during the development of this product.

One of the things that proved to be very useful was implementing unit tests for our application. It allowed us to quicky see what our application lacks and what bugs need to be fixed. This is something we will definitely keep in mind when working on future projects.

All in all, the development of this chat application was one of the most complex codebases we have built, which allowed us to improve not only our coding skills, but also our teamwork skills and a lot of good practices that we as programmers need to follow.