Chat App - Socket Programming

Nicholas Limbert, Jonathan Tooke, Shane Weisz

3/2/2019

Contents

Introduction 1
List of Features and Description of Implementation 1
Protocol Specification - Message Formats and Structure 3
Network Constraints 7
Concluding Remarks 8

Introduction

This report documents our network application that makes use of TCP sockets in order to implement a client-server chat application. We design and make use of an application layer protocol to support the client-server architecture. All interactions between clients are managed by the server. The program has been setup to work through a Graphical User Interface*(GUI)* to provide ease of access to the user. Some important and notable requirements dealt with are: **i)** the use of protocol design, **ii)** bandwidth constraints, **iii)** multiple host support, **iv)** user accessibility, **v)** chat archiving, and **vi)** file transfer. The overall design, implementation and functionality is also included.

List of Features and Description of Implementation

Basic Functionality

- The app has been designed to facilitate 1-on-1 communication between pairs of users, as required by the project specification.
- We allow for users to participate in multiple chats at once, and switch between them when they want to chat to different people. We included this since allowing multiple concurrent chats was deemed important functionality that a successful chat app must allow for.
- To make sure users know who they can currently chat to and don't make any mistakes in this regard, users are only allowed to send messages to other users that are online. Users are logged out before closing the app, and all other users will then no longer be able to communicate with the logged out user until they come back online.
- Users have the option of either sending simple text messages to another user, or to transfer files such as photos and videos to the recipient.

Creative Additions and Extra Functionality

- We included a user-friendly GUI to enhance the user experience. The GUI is intuitive and easy-to-use, and additionally supports multiple error-checking facilities in line with a defensive programming approach. As an example of an application of our defensive programming technique that made possible by use of a GUI, is how the user must select the user to chat with from a combo box (displaying only the currently online users) - this preemptively prevents errors such as incorrect spelling of the name of the recipient or incorrect message formatting that could become a problem if a GUI had not been used. Other features provided by the GUI all aid to an enjoyable user experience - such as a user being able to see their chats even if they log out, being alerted if a user is the only user currently online, being sent a notification if a new message has been received, and so on.

- We have also provided chat archiving facilities so that if a user logs out and closes the app, and then logs back into the app, all of their previous chats with currently online users are restored. This is an extremely useful and important addition as users will then never lose their chat history with another user - which is sure to greatly increase user satisfaction with the app. A dashed line is displayed below the last message of the previous session in order for the user to differentiate between different chat sessions, with time stamps also included with each message as an extra benefit for the user.
- Another one of our creative additions was to implement a File Picker and File Saver that allows for very easy-to-use and user friendly transferring of files. When a user wants to transfer a file, they can browse through the documents stored on their computer and click on the desired one; and similarly when a user accepts a file they can browse and choose where on their computer they want the file to be stored.

Screenshots to Display Features

The below screenshots illustrate a typical course of events in the running of the application, to reveal its features:

This first screenshot displays our logging in procedure where the user is asked to enter their name, and it is checked for validity before the user is logged in.

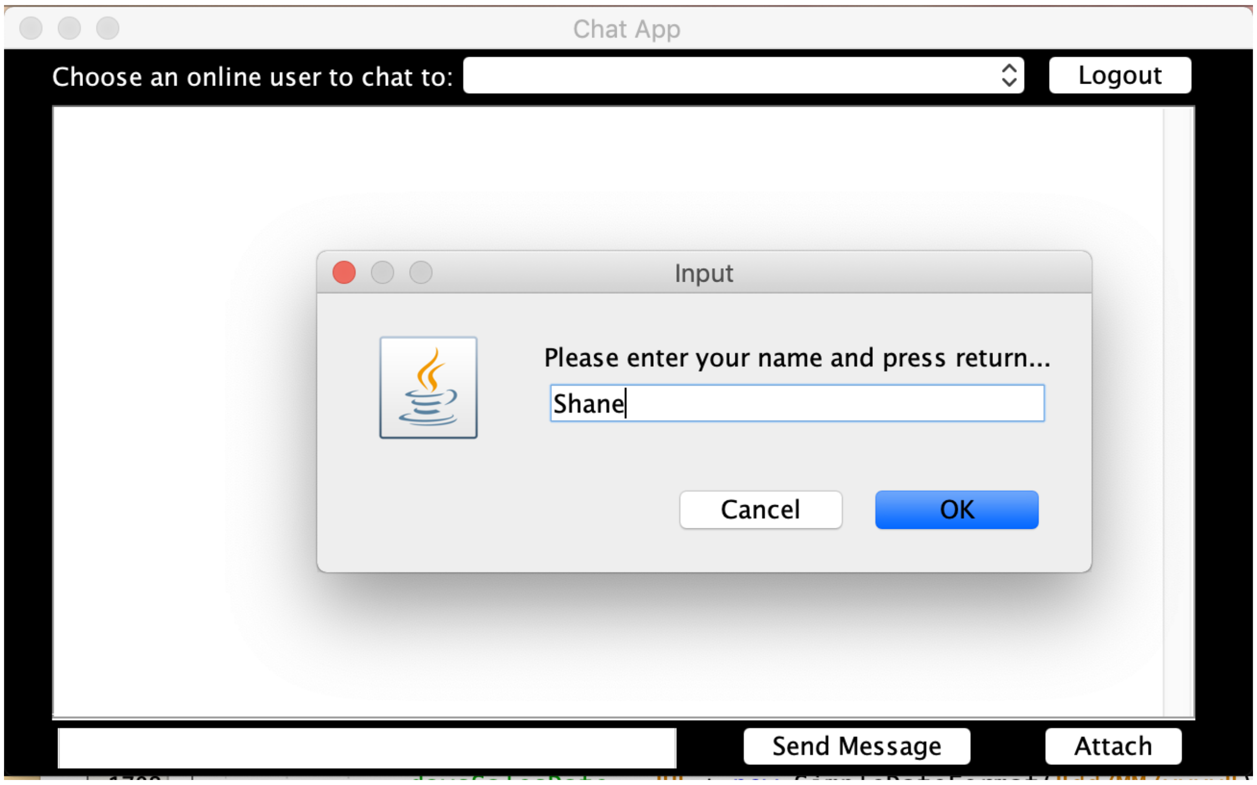
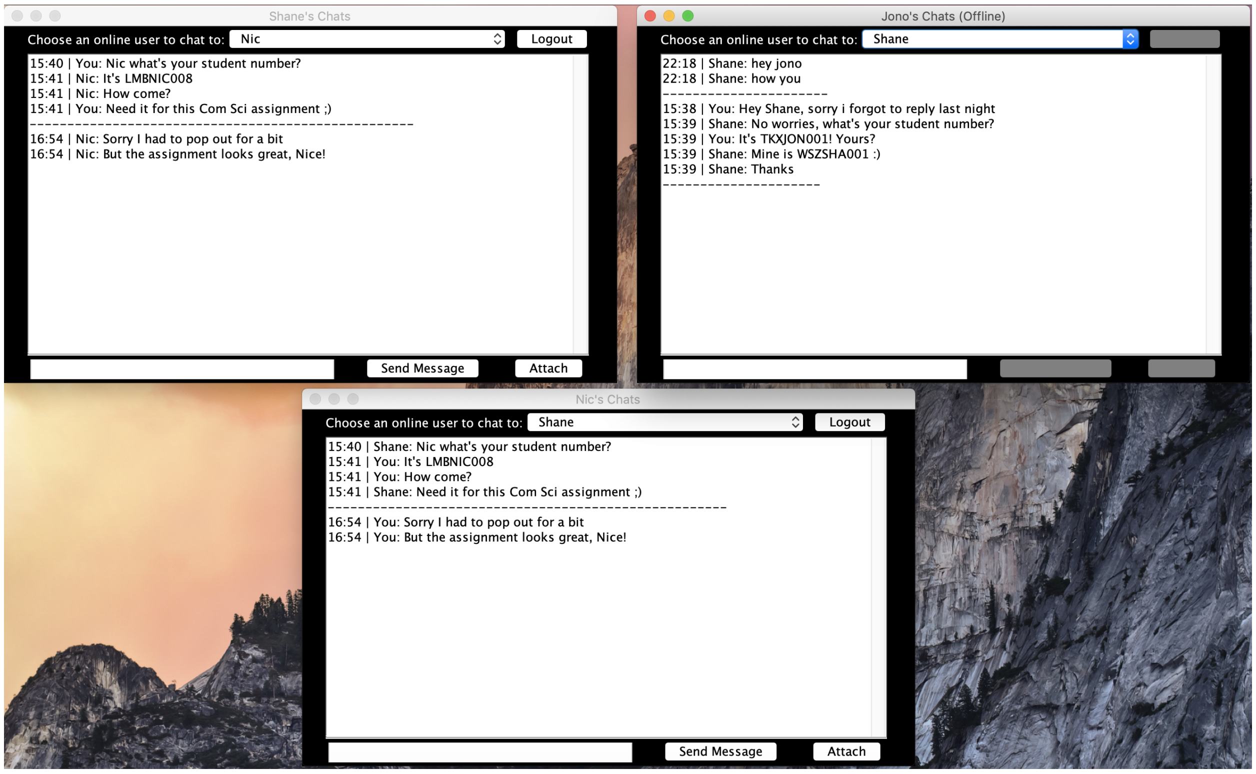

Chat App
Choose an online user to chat to:
Logout
Input
Please enter your name and press return...
Shane
Cancel
OK
Send Message
Attach

Figure 1: “Entering your name to log in”

The below screenshot captures many of the features of our chat app. Firstly, it shows that each user can have concurrent one-on-one conversations with other users, with the option of both sending a message and transferring a file. It shows the different chat sessions separated by dashed lines indicating when one of the respective users went offline and ended the specific chat session. It displays the combo box from which the user to chat with can be selected. It also shows our offline functionality: in the screenshot, Jono has logged out and gone offline - but can still view his old chats as can be seen in the screenshot.


Shane's Chats
Choose an online user to chat to: Nic
Logout
Send Message
Attach
Jono's Chats (Offline)
Choose an online user to chat to: Shane
Nic's Chats
Choose an online user to chat to: Shane
Logout
Send Message
Attach

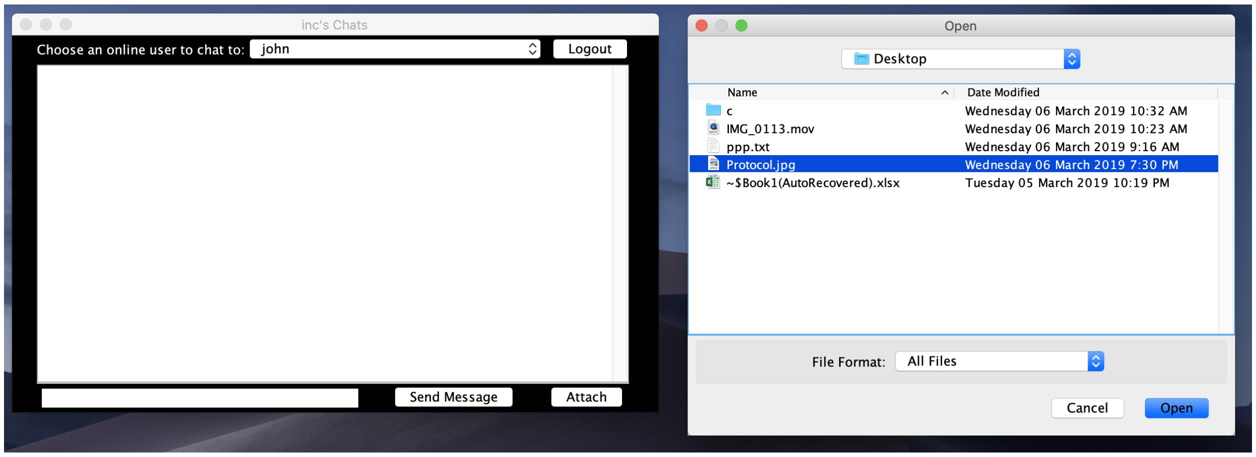
```
15:40 | You: Nic what's your student number?
15:41 | Nic: It's LMBNIC008
15:41 | Nic: How come?
15:41 | You: Need it for this Com Sci assignment ;)
---------------------------------------------------
16:54 | Nic: Sorry I had to pop out for a bit
16:54 | Nic: But the assignment looks great, Nice!
```

```
22:18 | Shane: hey jono
22:18 | Shane: how you
---------------------
15:38 | You: Hey Shane, sorry i forgot to reply last night
15:39 | Shane: No worries, what's your student number?
15:39 | You: It's TKXJON001! Yours?
15:39 | Shane: Mine is WSZSHA001 :)
15:39 | Shane: Thanks
---------------------
```

```
15:40 | Shane: Nic what's your student number?
15:41 | You: It's LMBNIC008
15:41 | You: How come?
15:41 | Shane: Need it for this Com Sci assignment ;)
---------------------------------------------------
16:54 | You: Sorry I had to pop out for a bit
16:54 | You: But the assignment looks great, Nice!
```

Figure 2: "Multiple concurrent chats"

The next screenshot below depicts the file transfer process whereby a user has clicked 'Attach' and then selects the file they wish to send; while the receiver would then accept the file into a directory of their choosing.



Name	Date Modified
c	Wednesday 06 March 2019 10:32 AM
IMG_0113.mov	Wednesday 06 March 2019 10:23 AM
ppp.txt	Wednesday 06 March 2019 9:16 AM
Protocol.jpg	Wednesday 06 March 2019 7:30 PM
~$Book1(AutoRecovered).xlsx	Tuesday 05 March 2019 10:19 PM

Figure 3: "File Transfer"

Protocol Specification - Message Formats and Structure

A protocol is used to control define the order and format of communication between clients and the server on a network. Thus in order to explain and describe how our protocol works, we will explain the *order* of communication that the protocol expects (with the sequence diagram below as a visual depiction of the order and procedure followed), along with the specific message structures (*format*) that is expected.

Protocol Description for Login Procedure

When a client opens the app, a connection is made between the client and the server with a TCP socket assigned for this communication. The protocol then dictates that the client is requested to enter their name, to which the client needs to respond with their name. The server then checks that the name is a valid choice - if so, then the client is made aware that they have been made online, otherwise the client is requested to enter another name.

Protocol Description for Sending and Receiving a Message

We decided that the protocol should require the message to be sent from the client to the server as a string in the format **[M]#[recipient_name]#[message]**. The 'M' stands for message, 'recipient_name' must be a string of 24 characters or less with the name of the person to send the message to and finally 'message' needs to contain a string with the message to be sent. The reason for using an 'M' flag is so that the server can differentiate between whether it needs to deliver a text message or transfer a file (which will require a 'F' flag as explained below). The other fields also need to be contained in the message as the server needs to know exactly what the message to be sent is and who to send the message to. We chose for a string to be sent with the message information as opposed to just a byte stream as it is a simpler means of transmitting the information and the decoding of the information is a lot easier. Lastly, the hashes were chosen to separate the fields since the hash is acknowledged as a universal delimiter.

When the server distributes the message to the specified client, it does so by sending a string in the format **[M]#[sender_name]#[message]**. The reasons for this choice mirror the reasons for the similar message required for sending a message.

The order of events that the protocol expects then is as follows. When a client wishes to send another client a message, they enter the message in the text field, select the user from the combo box and then click enter. So the after the user has selected the recipient and entered and sent the message, the ChatClient class captures this information to the server in the above format ([M]#[recipient_name]#[message]) and sends it through to the server. The server then receives and breaks the message up into its necessary components, and then makes sure the recipient is amongst the online users. It then sends a string to the specified client in the format [M]#[sender_name]#[message], who then decodes this message received and alerts the user that a new message has been received.

Protocol Description for Transferring and Receiving a File

When transferring a file, the protocol will require the message containing the necessary information about the file to be sent in a byte stream as opposed to a string, since the file could be an image or video which is better captured as bytes as opposed to a string. The message sent from a client to the server when they want to transfer a file is a byte stream in the following format: **[F][recipient_name][file_name][file_size][file_data]**. The 'F' flag has 1 byte allocated to it, the 'recipient_name' has 31 bytes allocated to it, the 'file_name' has 32 bytes, 'file_size' has 8 bytes and the 'file_data' is restricted to be 20MB maximum. The 'F' flag is used to indicate that the message is for a file rather than a text message (which would have an 'M' flag). The 'recipient_name' and 'file_name' are necessary inclusions associated with the file, whilst the 'file_size' contains a string with the exact byte size of the file - this will be used when asking the recipient whether they want to accept or decline the file transfer. Lastly, we have decided to limit the maximum file size that can be transferred to 20MB because this was deemed suitable for our purposes as a basic chat application. Should larger files need to be accommodated for, our application could be scaled up and we could switch to a more powerful server to accommodate handling larger files.

When a file transfer request message is distributed to the server, the server then sends the recipient client a message in the format **[R]#[sender_name]#[file_name]#[file_size]**, where the 'R' flag stands for request for file transfer. This information is taken from the file transfer message the server received from the sender client, and it is presented to the user so that the user can decide whether or not to accept the file based on who the sender is, what the file name is and how big the file is (thus accommodating for bandwidth constraints). If the client decides to accept the file, it sends the server a return message in the format [**A**]#[**sender_name**] (with the 'A' flag denoting accepting a file). The server then sends the file to the recipient client using a byte stream with the following format [**F**]#[**file_name**]#[**file_size**]#[**file_data**]. The client receiving the file then knows that a file is being received, what the file name should be saved as, as well as the actual file data to be saved.

The order of events that the protocol then expects in the running of the program is as follows. A user clicks the Attach button and proceeds to use the file picker to choose the file they wish to transfer. The client program then sends this information to the server in the form [**R**]#[**sender_name**]#[**file_name**]#[**file_size**]. The server transmits this information to the receiving client in the format specified above, who then either accepts or declines the file transfer. If the file transfer is accepted, the server is sent a message detailing as such in the format above, and follows this with the actual file being sent to the client with the corresponding information as we have specified in the protocol structure above. The recipient user then uses a file saver to browse through where on their computer they want their file to be saved, and the file is saved to that directory as desired.

Sequence Diagrams

The below diagrams - **Figure 1** indicate the main scenarios for which our protocol controls and dictates the format and order of communication: **i)** the user makes a connection to the server and sends messages or files to another user online, and **ii)** the user makes a connection to the server and receives a message from the server, or the server requests to transfer the sent file to the destination user who will have the option to download or decline the file.

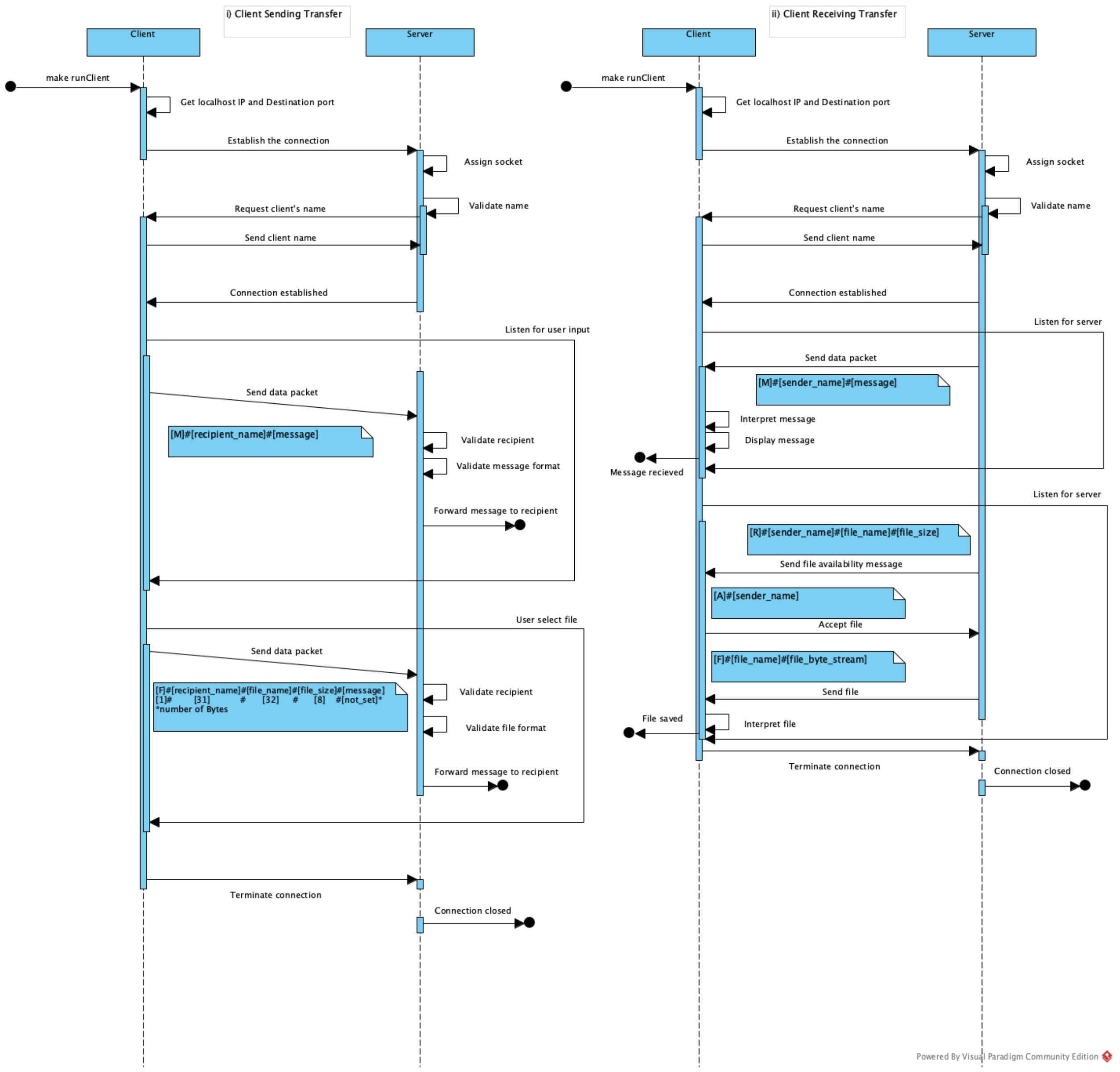


Figure 4: "Sequence Diagrams"

Robustness of Protocol Implementation

The following are examples of error-checking components that were included in our protocol in order for it to be robust and handle erroneous/malformed messages:

Logging in

- If the name entered is too many characters (larger than 24), the user is requested to enter another name. This prevents users entering names that are too long which causes formatting issues with regards to the display of the name.
- If no name is entered at all but the enter button clicked, the protocol ensures that the user is informed

that no name has actually been entered and the user is requested to try again.
• If a name is entered but there is already a user on the system with that name, the user is told that the name has already been used and a new unique name must be entered.

Chatting

• The user must select a user to chat to from a combo box of online users. Selecting from a combo box ensures only a valid name can be entered which ensures the correct message in the correct format is passed through the protocol.
• The user cannot send an empty message - the send button only works if the message field contains at least one character entered.
• The user cannot click the exit button until the user has logged out. This controls the order of procedure for a user leaving the chat and makes sure that the protocol is in a position to alert all the necessary parties of the user's departure.

File Transfer

• We have incorporated a file chooser in our GUI for navigating files and directories thus limiting the user input to only files that are available. The user cannot select a file that is not available.
• The user also selects where the file will be saved upon approval of file download. Again this is navigated through a file chooser. • If the file size or file name length is too long, the user will be alerted as such and will be taken back to their chat window.

Network Constraints

Bandwidth Constraints Regarding File Transfer

Bandwidth constraints are an important aspect of any network application. In particular, they are pertinent to client-sever applications where the client may be working on a low-spec device that has limited capacity, or where the client needs to manage the data received/sent to avoid high data usage charges. Before we explain how this constraint is accounted for, we provide a brief reminder of the protocol for our file transfer process: we will allow the sending user to select a file (the user will be able to send any file type) to be sent to the recipient, who will be given the option of whether to accept the file or not, and then have the file sent to a directory of the user's choice if accepted. The most important way in which we account for the bandwidth constraint is through providing the client the option of accepting or declining the file. The client is made aware of the name of the sender, name of the file as well as the size of the file. Based on this information, the client can then make an informed decision about whether or not they are willing to use their bandwidth (and potentially incur data charges) to download the file. For example, if the user sees the message is from a certain friend who often sends annoying files, or sees the file size is very large, they may choose to decline rather than accept the file. Additionally, the maximum file size that we allow to be sent is 20MB for the purpose of this application (i.e. text files, photos and short video clips), and this restriction also helps prevent users from incurring high data charges for large file transfers.

Chat Archiving and Security

One important constraint that needed consideration in designing our protocol and implementation of this network chat app was that of archiving chat histories with different user. If the user logs out of the app, it is important that if they go back online they are able to retrieve the chats that they had with other users in the network. This is useful both as a form of backup, as well as for interest and future use by the user. These chat histories are stored locally as text files in the same directory as the chat app class file, with a text file for each user the client has had a chat with.

This chat archiving implementation also required careful treatment with regards to the bandwidth used in retrieving archived chat histories. We decided to store chat histories locally on the client host's device rather than on the server, which prevents the chat histories from having to be downloaded from the server each time the user logs in. This approach thus means the client's bandwidth is prevented from having to be regularly spent on retrieving these files from the server.

In addition, the chat archives being stored locally provides an important security constraint. If the chat history text files were stored on the server, this could be a security leak as whoever is running the server would have access to every user's chats. By storing locally rather, it is guaranteed that only the sender and the recipient have a permanent record of the chat histories which is far more secure.

Realtime Interaction and Offline Functionality

Another important network constraint was to allow for both realtime interaction and offline functionality features. The protocol communication procedures ensures that there realtime chat interaction can take place between users. But we also decided to add an additional feature to manage the offline functionality constraint, whereby a user can still see all of their chats even once they have logged out. This is important because if the user needs to disconnect and go offline through logging out, they are still able to access their chat histories with other users - this offline functionality can be of major use to users. Additionally, the send and attach buttons are disabled so that users cannot attempt to contact people whilst they are offline.

Concluding Remarks

Our chat application has been a successful implementation of a client-server architecture using TCP sockets for communication. We designed and implemented a clear protocol for managing client-server interactions to facilitate the successful functioning of the app. We considered various network constraints in our protocol that needed to be carefully managed. It is also clear to see that our chat app includes a variety of core and additional features that provides for a user-friendly and successful chat application.