Majestic MesSENGer

Milestone 4

SENG 299, Summer 2017

Scott Andreen	sandreen	V00816349
Richard Cruikshank	rcruiksh	V00844041
Andrew LaRivere	ajlarivi	V00853200
Logan Macdonald	loganm	V00832812

Table of Contents

1.0 Introduction

This section introduces the Majestic MesSENGer chat system and describes its purpose, highlights relevant background information and important definitions, and gives an overview of the entire report.

1.1 Purpose

The purpose of this document is to provide an in depth description of all work completed in service of implementing the Majestic MesSENGer chat system. It will include a summary of all milestones submitted as deliverables for this project as well as a detailed description of the implementation, a review of the initial requirements and an external review, and a reflection on the success of the project and lessons learned.

1.2 Background

In this document, several technical topics will be discussed. Readers should have a functional grasp of object-oriented programming, UML Diagrams, and server-client interactions and terminology.

1.3 Definitions

Term	Definition
User	Person interacting with the chat system via the command line
Client	The interface that accepts text from a user and sends it to the server
Server	The system that controls flow of user input via chat rooms, and sends the input to all users in the same room
Chat Room	Means of organizing users into groups that can see the same messages
Mbps	Internet connection speed metric, megabits per second
Socket Module	A python module designed to facilitate low-level networking
Select Module	A python module that allows you to filter lists of readable, writable, and error sockets
UML	Unified Modelling Language, provides standards for visualizing software systems

1.4 Report Overview

Section 2 of this report provides a high-level overview of the system, using several different views to describe key functionalities. Section 3 discusses the implementation process, including design patterns, user interfaces, and other details relevant to the implementation of the final product. Section 4 reviews our original design requirements, and the changes we ultimately made to them. Section 5 discusses the recommendations given by an external review, and explains whether each suggestion was implemented or not. Section 6 describes the design process used throughout the project, and gives a timeline for the work that was completed. Section 7 outlines the roles of each group member throughout the project. Section 8 details different problems that were encountered throughout the project, and actions taken to resolve each problem.

2.0 High-Level Overview and Changes

This section contains all UML diagrams for the system, highlights their specifics, describes changes that were made to them throughout the design, and describes the system's high-level design.

2.1 Logical View

The system utilizes several major classes that interact with each other to allow the chat client to function properly. One of the most important classes is the ClientInfo class, which contains all information relevant to each person connecting to the client. A new instance of the ClientInfo class is created for each new user connecting to the system. There is a Room class, which contains lists of which users are in the room and another list for users blocked from that room. A new instance of the Room class is created whenever a user requests a new room, and that user is listed as the room creator and given the ability to block users and delete the room.

The system also uses several other modules to abstract the work. A connection handling class takes care of requests from new users to connect to the chat system. The RequestHandler class receives messages from clients and passes them to a TextHandler class, where they are interpreted as commands or as standard text. If the received text is a command, the class executes the appropriate command. If it is standard text, it sends the message to all users in the same room as the sender. The class diagram below (Figure 1) includes all classes, their attributes and methods, and the relationship they have to each other.

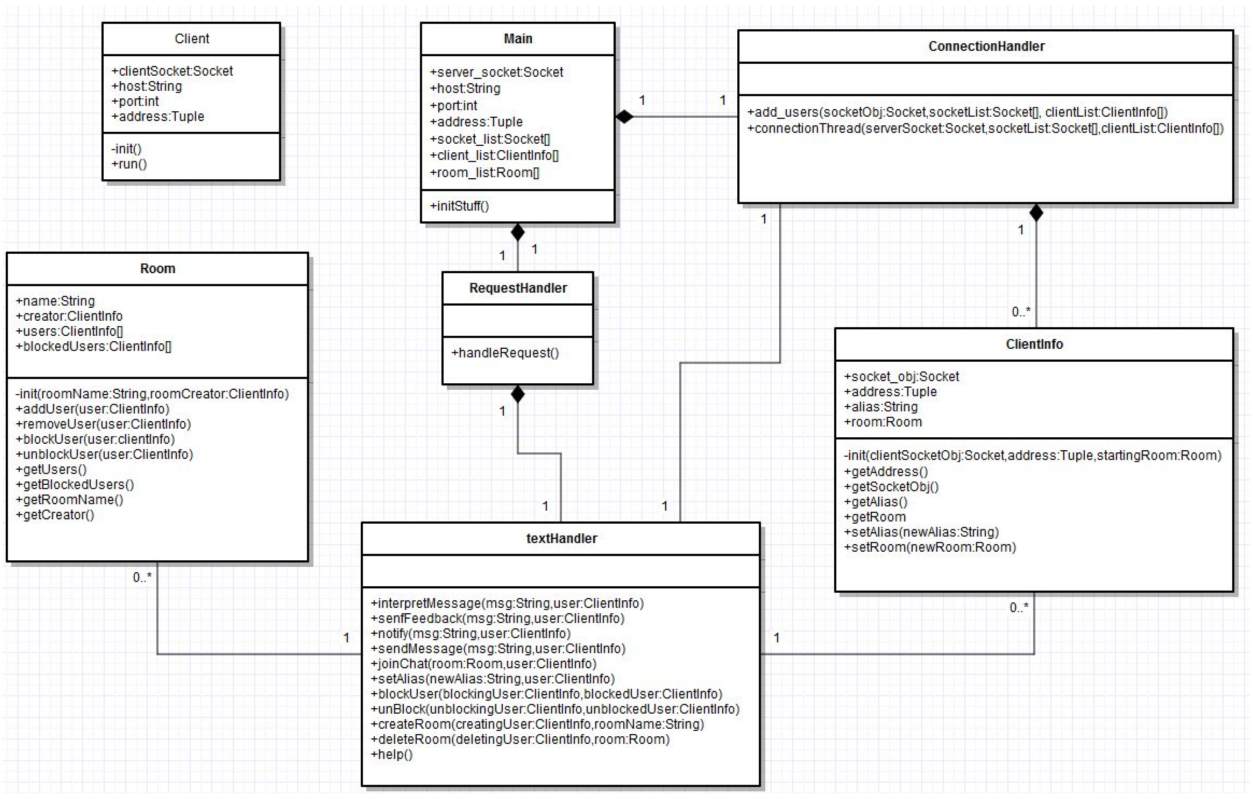

Client
+clientSocket:Socket
+host:String
+port:int
+address:Tuple
-init()
+run()
Main
+server_socket:Socket
+host:String
+port:int
+address:Tuple
+socket_list:Socket[]
+client_list:ClientInfo[]
+room_list:Room[]
+initStuff()
ConnectionHandler
+add_users(socketObj:Socket,socketList:Socket[], clientList:ClientInfo[])
+connectionThread(serverSocket:Socket,socketList:Socket[],clientList:ClientInfo[])
Room
+name:String
+creator:ClientInfo
+users:ClientInfo[]
+blockedUsers:ClientInfo[]
-init(roomName:String,roomCreator:ClientInfo)
+addUser(user:ClientInfo)
+removeUser(user:ClientInfo)
+blockUser(user:clientInfo)
+unblockUser(user:ClientInfo)
+getUsers()
+getBlockedUsers()
+getRoomName()
+getCreator()
RequestHandler
+handleRequest()
ClientInfo
+socket_obj:Socket
+address:Tuple
+alias:String
+room:Room
-init(clientSocketObj:Socket,address:Tuple,startingRoom:Room)
+getAddress()
+getSocketObj()
+getAlias()
+getRoom
+setAlias(newAlias:String)
+setRoom(newRoom:Room)
textHandler
+interpretMessage(msg:String,user:ClientInfo)
+senfFeedback(msg:String,user:ClientInfo)
+notify(msg:String,user:ClientInfo)
+sendMessage(msg:String,user:ClientInfo)
+joinChat(room:Room,user:ClientInfo)
+setAlias(newAlias:String,user:ClientInfo)
+blockUser(blockingUser:ClientInfo,blockedUser:ClientInfo)
+unBlock(unblockingUser:ClientInfo,unblockedUser:ClientInfo)
+createRoom(creatingUser:ClientInfo,roomName:String)
+deleteRoom(deletingUser:ClientInfo,room:Room)
+help()
1
1
1
1
1
1
0..*
1
1
1
1
0..*
1
1
0..*

Figure 1: Class Diagram

Several changes were made to our classes as the project progressed. The first change was creating the connectionHandler class. The connectionHandler class split up the responsibility of RequestHandler. RequestHandler is still responsible for receiving data from clients however Connection Handler is now responsible for accepting new connections from clients and assigning them to clientInfo. The connectionHandler had to be connected to textHandler in order to send feedback to the user when connected.

The second change was altering the class diagram to reflect the connection of Room and ClientInfo to textHandler. This change reduced the need for request handler to pass arguments between these classes whenever commands changed their attributes, for example setting user alias or blocking users.

Some minor changes made were additional methods for returning and changing attributes, get() and set(), and three methods for sending messages in textHandler. SendMessage sends basic messages to users in the room, sendFeedback sends information about changes made by a command to the user that executed the command, and notify broadcasts feedback messages to other users in the room. Finally all attributes used by each class were added or modified to make the diagram accurate to the final implementation.

2.2 Process View

Upon startup, the server is initialized, setting up the host and creating a general chat room. The server begins running ConnectionHandler and RequestHandler, waiting for incoming connections. When a client connects, a new instance of the ClientInfo class is created to track the user's info, and they are placed in the general chatroom. At all times the general chat room exists. From there, users are able to send text to the server through the RequestHandler class to then be relayed to the TextHandler class.

The system interprets any input starting with a '/' as a command, and then checks to see if the command is valid. If it is an invalid command, the help method is executed. If it is a valid command, the system then executes the command, changing either the ClientInfo or Room class. If the request is to create a new room, a new instance of the Room class will be created, and the user who created the room will be moved to it and added as the room user. Any changes made to the Room or ClientInfo class are returned to the textHandler to be sent as feedback. Any text not starting with a '/' is treated as a message. The textHandler sends data, be it raw text messages or command feedback to the client. The Sequence diagram below (Figure 2) outlines the interactions between the classes during normal use of the chat server as described.

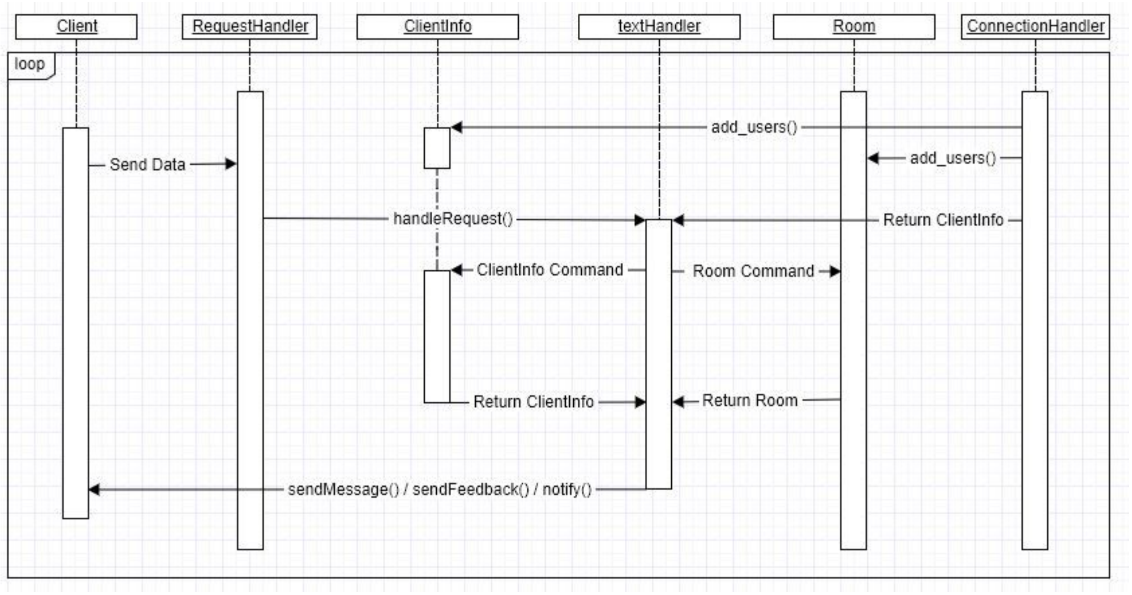


Figure 2: Sequence Diagram

Our sequence diagram actually proved quite accurate in representing the processes in our design, however some changes still needed to be made. The addition of the ConnectionHandler class had to be represented in the sequence diagram. This creates a new clientInfo instance and returns the results to textHandler. The final design also changed the data flow so now information is always returned to textHandler to be relayed to the client via sendMessage(), sendFeedback(), or notify().

Some smaller changes included adding appropriate method calls for clarity and adjusting the activity of the client to reflect the final implementation. Any init() calls were removed from this diagram to reduce clutter as any classes initialised are active in the diagram from the start, so labeling them as such would not provide additional insight.

2.3 Data View

This section will reference classes from the Class Diagram in the Logical View. Data received from the server can follow one of three paths. After a user has input text, the client sends it to the server. From there, the server's textHandler class determines whether or not the message is a command, indicated by a '/' as the first character. If it is an improperly formatted command, it executes the help command, which will send a help message to the client, displaying proper command formatting, and a list of valid commands. If the message is a valid command, textHandler executes one of the commands from its list of methods. After executing the command the server will send a feedback message outlining any changes. If it is not a command and is raw text, the sendMessage method is called. The message will be sent to all clients belonging to the chat room of the client that sent the message. The Activity Diagram below (Figure 3:) follows the path of text entered by the user through the server.

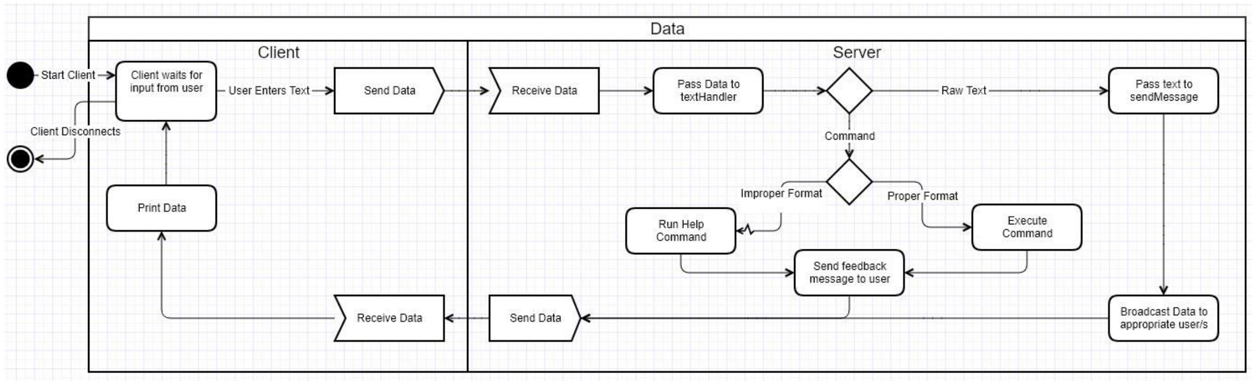


Figure 3: Activity Diagram

We received some recommendations about this diagram during the external review. During the design process the diagram did not accurately represent the data flow, terminating after text was processed by the server. To more accurately follow the data flow of our final implementation, data is always returned by the server to the client, either as a message or feedback. This more closely resembled the infinite loop of the client, only terminating if the client disconnects.

Apart from these changes the data flow within the server remains largely the same, with slight changes to the activities to reflect additional methods, like sendMessage and sendFeedback.

3.0 Implementation

This section covers the design patterns that were taken into consideration for this system and shows the main user interface views.

3.1 Design Patterns

A number of design patterns were implemented on the server side of this system, at a base level, we attempted to maintain high cohesion and low coupling, by using a module system, and trying our best to keep each module focused, independent from other modules. ConnectionHandler acts as a factory for ClientInfo objects in our design, each time a new connection is established with a client, ConnectionHandler instantiates a ClientInfo object in order to correspond to that connection and passes it all the appropriate socket information required to reference that connection and its corresponding info such as room, alias, and so on.

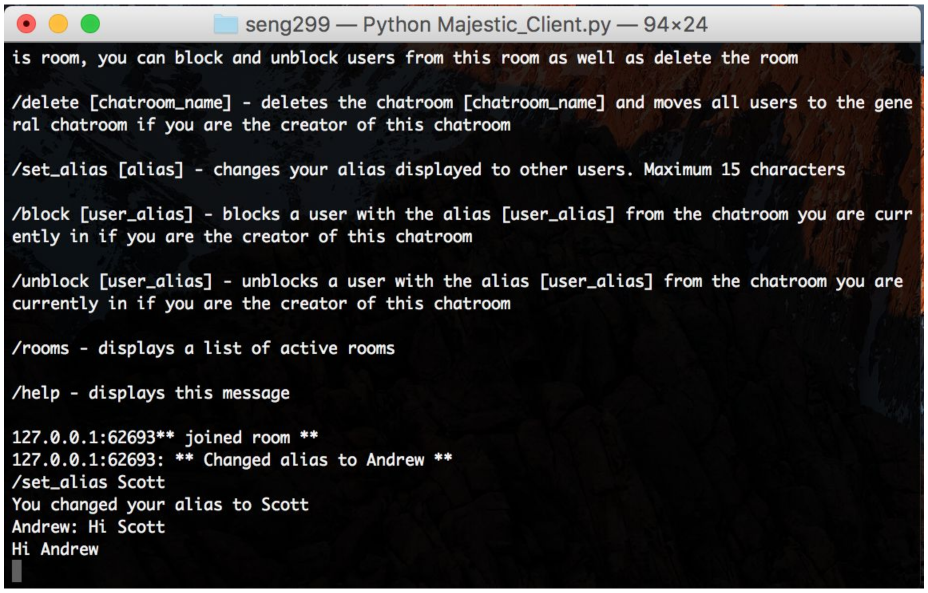
Additionally, we implemented a form of indirection in the RequestHandler class. The function of this class is to abstract the handling of requests coming in to the servers from clients away from main and act as an intermediary from main and textHandler. This lowers coupling and increases cohesion for the design as a whole.

Finally, we also used a controller taking the form of our textHandler class. The function of this class is to interpret each system event handed to it by requestHandler and then perform the appropriate action in order to modify a clientInfo or Room class, or to send a message to the appropriate room. In doing so, this class interprets messages and then delegates the work to other classes in order to carry out the underlying function of the message from a particular client.

On the client side, no real design patterns were implemented. We elected to keep things simple and clean. The client only has one class with few attributes and methods as most of the work is done by the server.

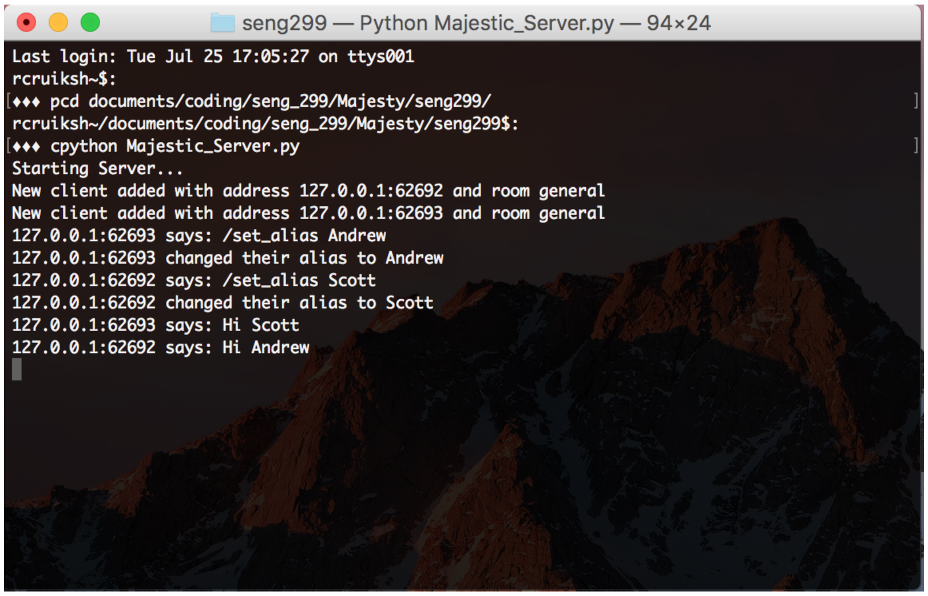
3.2 User Interfaces

This section includes screenshots (Figure 4: and Figure 5:) of the two main user interface screens; the server and the client. Both are simply run from the command line.



```
is room, you can block and unblock users from this room as well as delete the room

/delete [chatroom_name] - deletes the chatroom [chatroom_name] and moves all users to the gene
ral chatroom if you are the creator of this chatroom

/set_alias [alias] - changes your alias displayed to other users. Maximum 15 characters

/block [user_alias] - blocks a user with the alias [user_alias] from the chatroom you are curr
ently in if you are the creator of this chatroom

/unblock [user_alias] - unblocks a user with the alias [user_alias] from the chatroom you are
currently in if you are the creator of this chatroom

/rooms - displays a list of active rooms

/help - displays this message

127.0.0.1:62693** joined room **
127.0.0.1:62693: ** Changed alias to Andrew **
/set_alias Scott
You changed your alias to Scott
Andrew: Hi Scott
Hi Andrew
```

Figure 4: Screenshot of Client Interface



```
Last login: Tue Jul 25 17:05:27 on ttys001
rcruiksh~$:
[♦♦♦ pcd documents/coding/seng_299/Majesty/seng299/                                          ]
rcruiksh~/documents/coding/seng_299/Majesty/seng299$:
[♦♦♦ cpython Majestic_Server.py                                                              ]
Starting Server...
New client added with address 127.0.0.1:62692 and room general
New client added with address 127.0.0.1:62693 and room general
127.0.0.1:62693 says: /set_alias Andrew
127.0.0.1:62693 changed their alias to Andrew
127.0.0.1:62692 says: /set_alias Scott
127.0.0.1:62692 changed their alias to Scott
127.0.0.1:62693 says: Hi Scott
127.0.0.1:62692 says: Hi Andrew
```

Figure 5: Screenshot of Server Interface

4.0 Requirements

This section covers the system's requirements throughout the design, and provides a final list of requirements for the finished product.

4.1 Milestone 1 Requirements

Functional Requirements:

1 User

1.1 User may enter text into the chat client
1.2 User may set their alias to be displayed by chat client
1.3 User may use several commands to interact and moderate chat rooms
- 1.3.1 User may join/leave chat rooms
- 1.3.2 User may create/delete a chat room, User will be room Admin
- 1.3.3 Admin User may block/ unblock users in their chat room
- 1.3.4 User may enter help command to receive list of available commands

1.4 User shall see text entered by themselves and other users in the chat room

2 Chat client

2.1 Chat Client shall support chat rooms
2.2 Chat client shall display text entered by the user
- 2.2.1 Text will viewable in the chat room currently inhabited by User
- 2.2.2 Text will be displayed as **[Alias]: [Text]**

2.3 Chat client will respond to commands entered by user
- 2.3.1 **/join [chatroom_name]** - commits user to specified chat room, prompts if room does not exist
- 2.3.2 **/leave [chatroom_name]** - removed user from chat room
- 2.3.3 **/create [chatroom_name]** - creates chat room, assigns user as Admin
- 2.3.4 **/delete [chatroom_name]** - deletes chat room, removing all users
- 2.3.5 **/set_alias [alias]** - assigns user display alias
- 2.3.6 **/block_user [user_alias]** - blocks user with alias from interacting with chat room
- 2.3.7 **/unblock_user [user_alias]** - allow blocked user to interact with chat room
- 2.3.8 **/help** - lists available commands

Performance requirements:
Input shall be received by the chat server and relayed to all members of the chat room within 7 seconds of being received, 90% of the time.

Design constraints:
All of the code for this product must be implemented in Python, the server should be written using either Python requests, urlib2, or socket as designated by the project outline. Pre-made servers such as simpleHTTPServer cannot be used, the server must be coded from scratch by developers.

Software System Attributes:

Reliability:

Users should have a computer with an internet connection of at least 5 Mbps to reliably connect to the chat client.

Availability:

The user should receive a response from the chat client within 10 seconds of establishing the connection, assuming a connection speed of 5Mbps.

4.2 Amendments

A number of changes were made to the requirements initially decided upon in milestone 1 which have been outlined below.

- Changed functional requirement 2.2.2 to say that text from other clients will be displayed as [Alias]: [Text]
- Removed the "chatroom_name" argument from functional requirement 2.3.2, now just removes the user from their current room and moves them to the general chatroom
- Clarified functional requirement 2.3.4, users are removed from the room and into general
- Clarified functional requirement 2.3.5, an alias cannot be assigned if it is already in use, can also be no longer than 15 characters
- Clarified functional requirement 2.3.6 and 2.3.7, users are blocked or unblocked from the room that the caller uses the command from, if and only if the caller is the creator of the room
- Added functional requirement 2.3.9, **/rooms -** list all rooms available to join
- Added functional requirement 2.3.10, any message beginning with a '/' which is not a valid command will invoke the **/help** command
- Amended performance requirements:

  PR1 - Input shall be received by the chat server and relayed to all members of the chat room within 1 second of being received, 99% of the time.

  PR2 - The Performance Requirements are guaranteed up to a maximum of 20 users.
- Added identifiers C1, C2, and C3 to constraints
- Added constraint C4: The system is supported primarily on macOS. (It is likely that it functions on all unix-based systems, but so far it has only been tested on macOS. The use of the select module prevents it from functioning on windows.)

4.3 Updated Requirements List

Identifier	Description	Achieved
1.1	User may enter text into the chat client	yes
1.2	User may set their alias to be displayed by chat client	yes
1.3	User may use several commands to interact and moderate chat rooms	yes
1.3.1	User may join/leave chat rooms	yes
1.3.2	User may create/delete a chat room, User will be room Admin	yes
1.3.3	Admin User may block/ unblock users in their chat room	yes
1.3.4	User may enter help command to receive list of available commands	yes
1.4	User shall see text entered by themselves and other users in the chat room	yes
2.1	Chat Client shall support chat rooms	yes
2.2	Chat client shall display text entered by the user	yes
2.2.1	Text will viewable in the chat room currently inhabited by User	yes
2.2.2	Text from other clients will be displayed as **[Alias]: [Text]**	yes
2.3	Chat client will respond to commands entered by user	yes
2.3.1	**/join [chatroom_name]** - commits user to specified chat room, prompts if room does not exist	yes
2.3.2	**/leave** - removes user from current chat room and moves them into the general room	yes
2.3.3	**/create [chatroom_name]** - creates chat room, assigns user as Admin	yes
2.3.4	**/delete [chatroom_name]** - deletes chat room, removing all users, and moving them into the general room	yes

2.3.5	**/set_alias [alias]** - assigns user display alias with maximum length of 15 characters. Users cannot choose an alias that is already in use	yes
2.3.6	**/block_user [user_alias]** - blocks user with alias from interacting with chat room. Only a room creator may block users	yes
2.3.7	**/unblock_user [user_alias]** - allow blocked user to interact with chat room. Only a room creator may unblock users	yes
2.3.8	**/help** - lists available commands	yes
2.3.9	**/rooms** - lists all active rooms	yes
2.3.10	Any message beginning with a '/' which is not a valid command will invoke the **/help** command	yes
PR1	Input shall be received by the chat server and relayed to all members of the chat room within 1 second of being received, 99% of the time.	yes
PR2	The Performance Requirements are guaranteed up to a maximum of 20 users.	yes
C1	All of the code for this product must be implemented in Python	yes
C2	The server must be written using either Python requests, urlib2, or socket	yes
C3	Pre-made servers such as simpleHTTPServer cannot be used, the server must be coded from scratch by developers	yes
C4	The system is supported primarily on macOS	yes
Availability	The user should receive a response from the chat client within 10 seconds of establishing the connection, assuming a connection speed of 5Mbps.	yes

5.0 Recommendations

This section describes recommendations that were given to us after an external review, as well as the resulting changes to our system.

5.1 Given Recommendations

Rank	Recommendation:
1.	Adding Connection Handling Class: In Milestone 2 section 2.1 Logical View, a connection handling class is mentioned but never used/displayed in the Class diagram. Adding this class to the Class Diagram will give better understanding of the system.
2.	Options after /leave or /delete commands: Currently the application does not provide any information on where the user goes after /leave or /delete command. The program could do one of the following options for /leave or /delete 1. Disconnect the user from system/logout (only for /leave) 2. Add the user(s) from the deleted or left chat room to the general room 3. Provide a list of active chat rooms for user(s) to join
3.	Checking Alias: The application allows the user to choose his/her own alias but does not check if that alias already exists. This can create ambiguity.
4.	Adding Command to Logout: There is no existing command for the user for logging out or disconnecting from the system.
5.	Checking active roomlist: It would helpful to have a command to see the active chatroom list and number of members/users in them. Currently there is no way for a user to see other existing rooms.
6.	Number of rooms joined: At present the system does not provide any information on the number of chat rooms a user can join. By tracking the number of rooms joined the application can also put a limit to the number of rooms a user could join.
7.	Fixing Activity Diagram: The diagram lacks the flow of the activity. After input text has been sent, the program should flow back to the initial point where the user can enter another input text. At present the flow ends at 'Send text to clients in chat room' , 'Display help text to user', 'Execute command' or Logout/End program.
8.	Remove Ambiguity from Functional Requirement Milestone 1: 1. Combine 2.2, 2.2.1, 2.2.2 into one requirement that the user can view text in the format [Alias]: [Text] 2. Move 1.3 and 2.3 to Constraints section 3. Change the second 2.3.7 to 2.3.8
9.	Split the textHandler class into smaller chunks: The textHandler class makes up the majority of the server code, and could be split up.

5.2 Actions Taken

1. At the recommendation of the group reviewing our first two milestones, we added a connection handler class to reduce the coupling and increase the cohesion of the overall system.
2. We decided to clarify the behaviour of the /leave and /delete commands as they pertain to different chat rooms. Originally it was not clear, but using either of those commands will bring users back to the ‘main’ chat room, which cannot be deleted or left, except if moving to a different chat room.
3. We added extra functionality within the /set_alias command to check whether an alias had been taken or not. When a user tries to set their alias to one that has been taken, the system will return an error and ask them to choose a different alias.
4. Our system, as presented, still does not support a logout command of any sort. We decided against this added functionality as it was not a requirement in the initial project outline. It is just as easy for a user to terminate the process on their end, and the system already handles user disconnects gracefully.
5. We decided to add a command that allows users to see a list of all available rooms. This makes it easy for them to decide which room to join.
6. The suggestion to track the number of rooms, along with the number of rooms a user has joined, was not implemented because it was ultimately determined to be unimportant. A single user can only be in one room at any given time, so the different rooms they move between is not relevant. For the purpose of functionality, the system only needs to know which room they are currently in, which it already does.
7. As suggested we altered the flow path to continuously loop and made changes to more closely follow the completed system. When data is received by the server it will still be processed to determine if it is a command or a massage. However after being processed by textHandler, either a message is sent to users in the chat room, or a feedback message is sent back to the user based on the command. When the data is received by the client the flow path starts again. The flow path will now only terminate if the client disconnects from the server.
8. We acted on several of the recommended changes to our requirements, and updated the requirements section accordingly. These changes are outlined in section 4.2 of this report.
9. We decided against splitting up the textHandler class as doing so would have greatly increased the coupling of our design

6.0 Design Process

This section describes the design process used during this project, as well as the project timeline.

6.1 Process Used

Throughout our design, we ultimately followed an agile approach. Although most of each design stage was fleshed out in full, there were times when we needed to go back and change some of the work done in previous stages, either to match with what we did in later stages, or because we realized that our decisions would not work.

At each stage in the design process, we found it important to look back on what we had done in previous stages. This allowed us to notice differences between stages, and we could then determine if we should stick with the ideas outlined in the previous stage, or if we should update our documentation with a better idea or design.

Although this approach sometimes resulted in extra work from going back and changing things in earlier stages, it ultimately allowed us to be more adaptable. The added flexibility allowed for a better final implementation, especially since there were certain libraries we used that we did not initially have knowledge in.

6.2 Timeline

Below are a gantt chart and a table (Figure 6 and Figure 7)corresponding to each element. These match our milestone 1 timeline except for any submission deadlines that were extended.

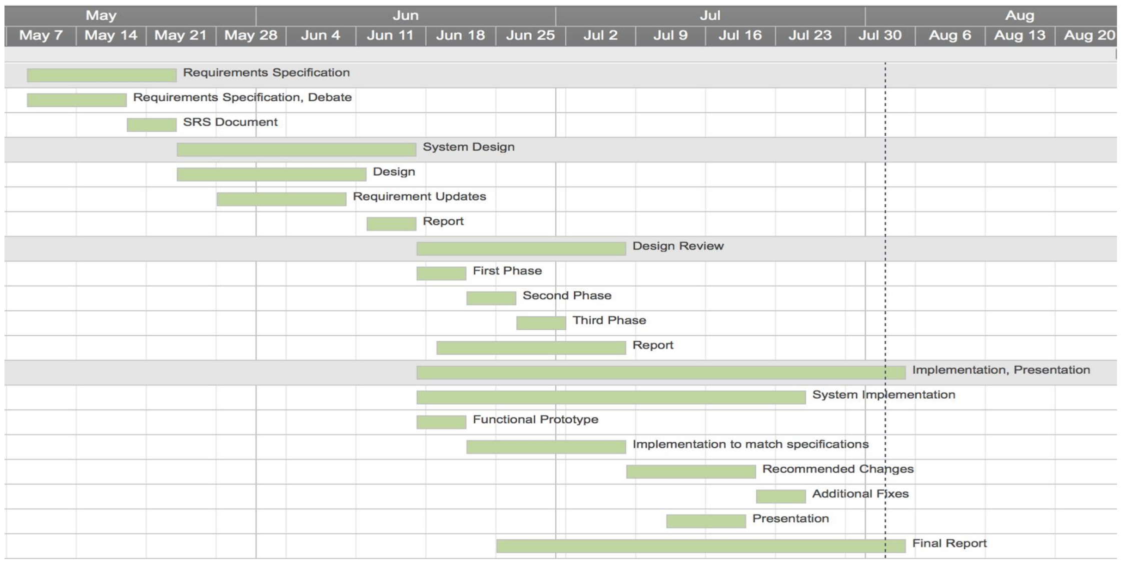
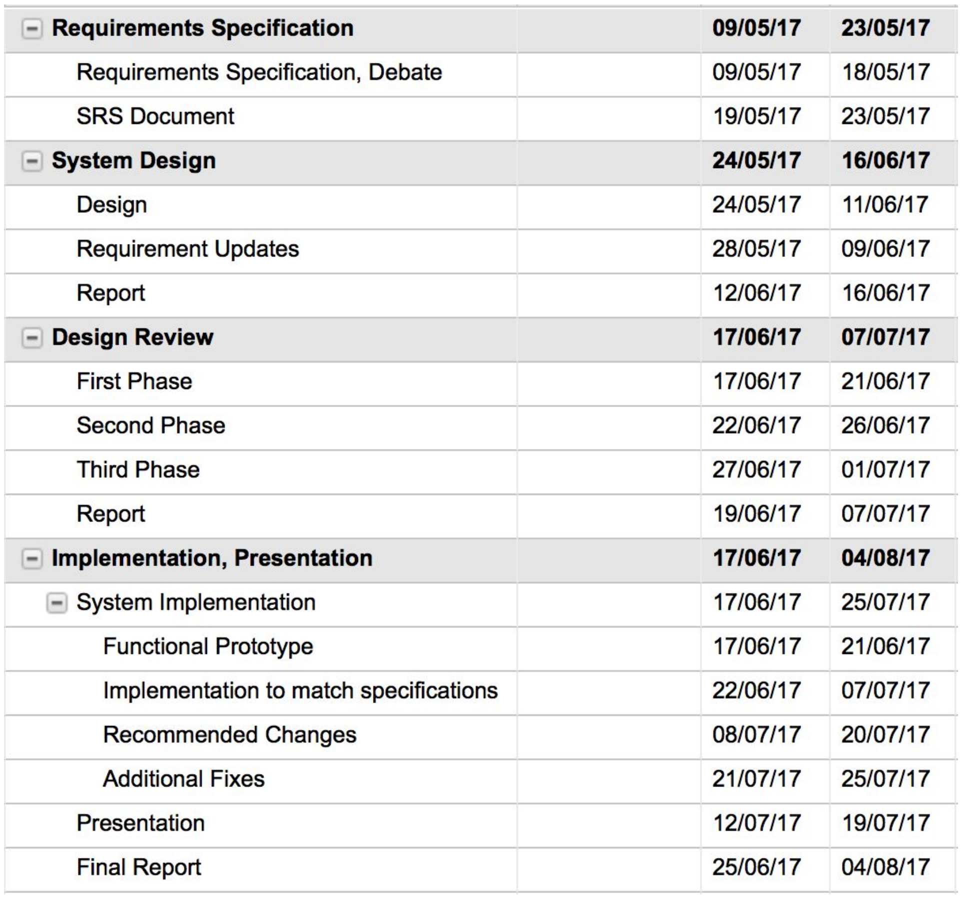


Figure 6:Timeline Gantt Chart



Requirements Specification		**09/05/17**	**23/05/17**
Requirements Specification, Debate		09/05/17	18/05/17
SRS Document		19/05/17	23/05/17
System Design		**24/05/17**	**16/06/17**
Design		24/05/17	11/06/17
Requirement Updates		28/05/17	09/06/17
Report		12/06/17	16/06/17
Design Review		**17/06/17**	**07/07/17**
First Phase		17/06/17	21/06/17
Second Phase		22/06/17	26/06/17
Third Phase		27/06/17	01/07/17
Report		19/06/17	07/07/17
Implementation, Presentation		**17/06/17**	**04/08/17**
System Implementation		17/06/17	25/07/17
Functional Prototype		17/06/17	21/06/17
Implementation to match specifications		22/06/17	07/07/17
Recommended Changes		08/07/17	20/07/17
Additional Fixes		21/07/17	25/07/17
Presentation		12/07/17	19/07/17
Final Report		25/06/17	04/08/17

Figure 7: Timeline Table

7.0 Roles

This section describes each team member's role during each phase of the design and implementation of the system.

Stage	Group Member	Roles
Requirements Specification	Scott	Functional requirements, constraints
	Richard	Functional requirements, project plan
	Logan	Quality attributes, functional requirements
	Andrew	Performance requirements, functional requirements
System Design	Scott	Logical description, data structures, design process used
	Richard	Data activity diagram and descriptions, timeline
	Logan	Process sequence diagram, process description
	Andrew	Logical class diagram, updated requirements
Design Review	Scott	Purpose, Architecture, Data Flow, Completeness and Quality Reviews
	Richard	Design Patterns, Notation, Document Structure, Architecture, Editing
	Logan	Requirements and Quality Attribute Reviews, Checklist
	Andrew	Feasibility, Review Processes, Recommendations, Conclusion
Implementation & Presentation	Scott	Client implementation, testing, demo, requirements, actions taken
	Richard	Low-level socket interactions, server implementation, timeline, lessons learned, recommendation implementations
	Logan	UML diagrams, overview and changes, high level design, diagram descriptions
	Andrew	Server and command implementation, design patterns, recommendations, encountered problems

8.0 Encountered Problems

One major problem encountered by our team was our lack of familiarity with socket programming, and low-level networking as a whole. This meant that it was difficult for us to implement proper communication between the server and connected clients. This lack of understanding also caused some differences between our initial design ideas and the final implementation, based on what we learned as we were working on the programs themselves. However, we worked together and made extensive use of available documentation in order to learn how to properly handle socket connections.

Another major problem in our design process was our lack of clarity in our initial requirements. Because our requirements did not contain as much information as they should have, it caused some issues with the design of the system, and the implementation itself. We had to make several changes and updates, especially in the design stage, to fix this issue. However, we were able to solve the issue by setting aside some time to reexamine our requirements in more detail during our design stage, which made the implementation easier.

The third problem we experienced occurred during our design and implementation phases. As we worked on our design, and started implementing it, we realized that the way we had structured some elements of the system resulted in high coupling and low cohesion. In order to fix this, we needed to revisit our designs and split some modules into smaller, more cohesive ones. For example requestHandler was divided into a smaller version of itself, and a connectionHandler. We also rearranged the connections of other modules such as connecting Room and ClientInfo to textHandler. After some time working on this, we were able to reduce the coupling and increase the cohesion of the overall design.