Software Architecture Technical Report

for

Silencio CHAT CLIENT AND CENTRAL SERVER

Prepared by

Team Member	Responsibilities
Omnielle	Organization and setup. Section 1, 2, 3.1, 4, 6 + refinement
Others	Sections 5 + implementation
Others	Section 3.3 + implementation
Others	Sections 3.2 + implementation

July 2017

Table of Contents

1. Introduction

1.1 Statement of purpose

The purpose of this document is to present a final design view and implementation of the Silencio chat application. This document will incorporate all defined requirements, detailed design, critical reviews and recommendations from previous milestones. This will be accomplished through analysis of the overall design, including UML, screenshots and code snippets as well as outlining which requirements and/or recommendations have been implemented or not.

1.3 Source Code

https://github.com/Silencio0 also attached to submission.

2. Overall Design

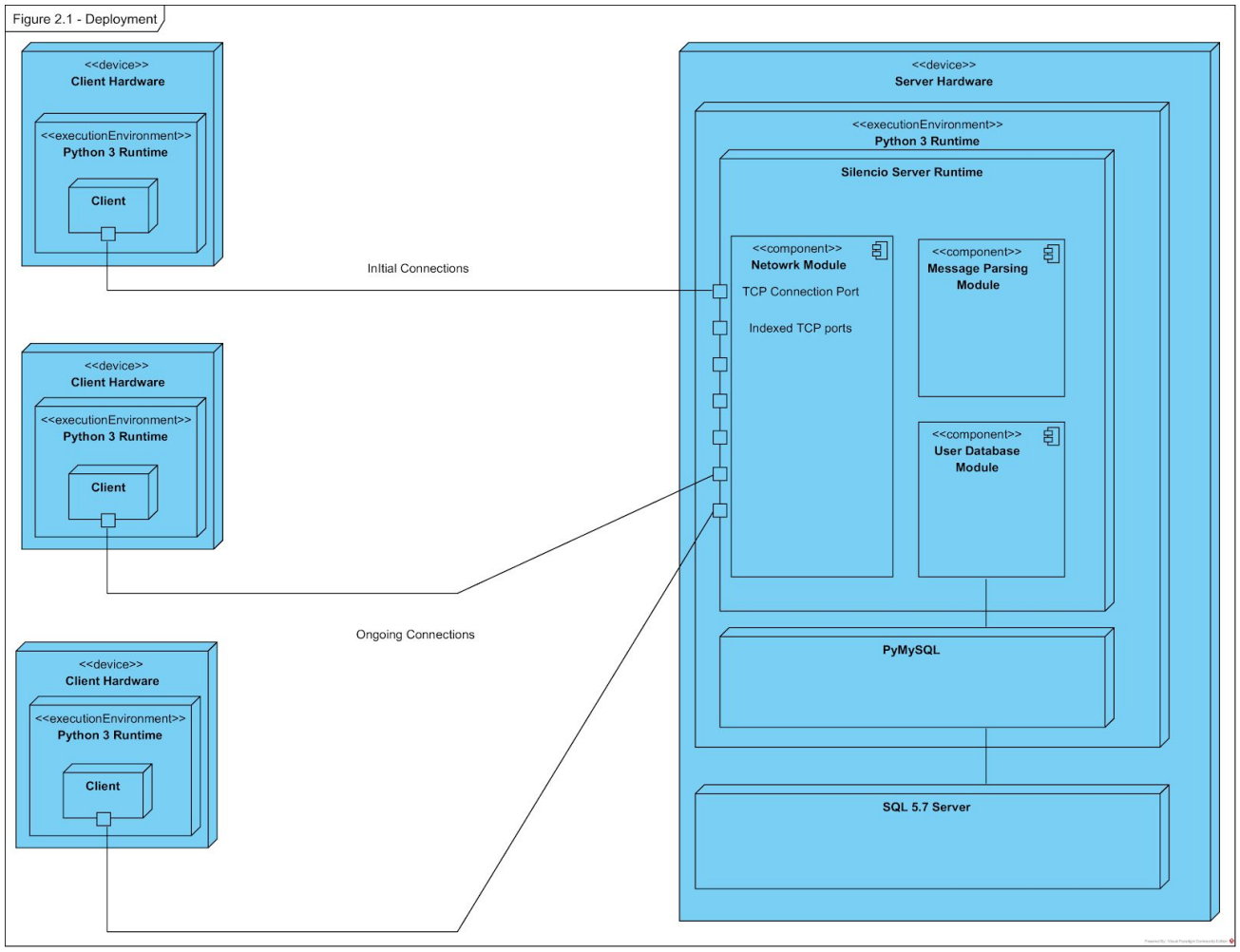
The design process was mainly focused on the functionality of the Silencio system and the available skill sets. Earlier on, we figured out the user end interface and following that came up with a deeper structure of the project. UML and use cases clarified all the components needed for the system.

Client has two main modules:

- Interface: Receives user input and forward it as a message
- Network: Communicate messages to and from the server

Server has three main modules:

- Network: Routes communication and manages user connections
- Message Parser: Processes and dispatches incoming messages and commands
- Database: Provides storage of user and chatroom data settings.



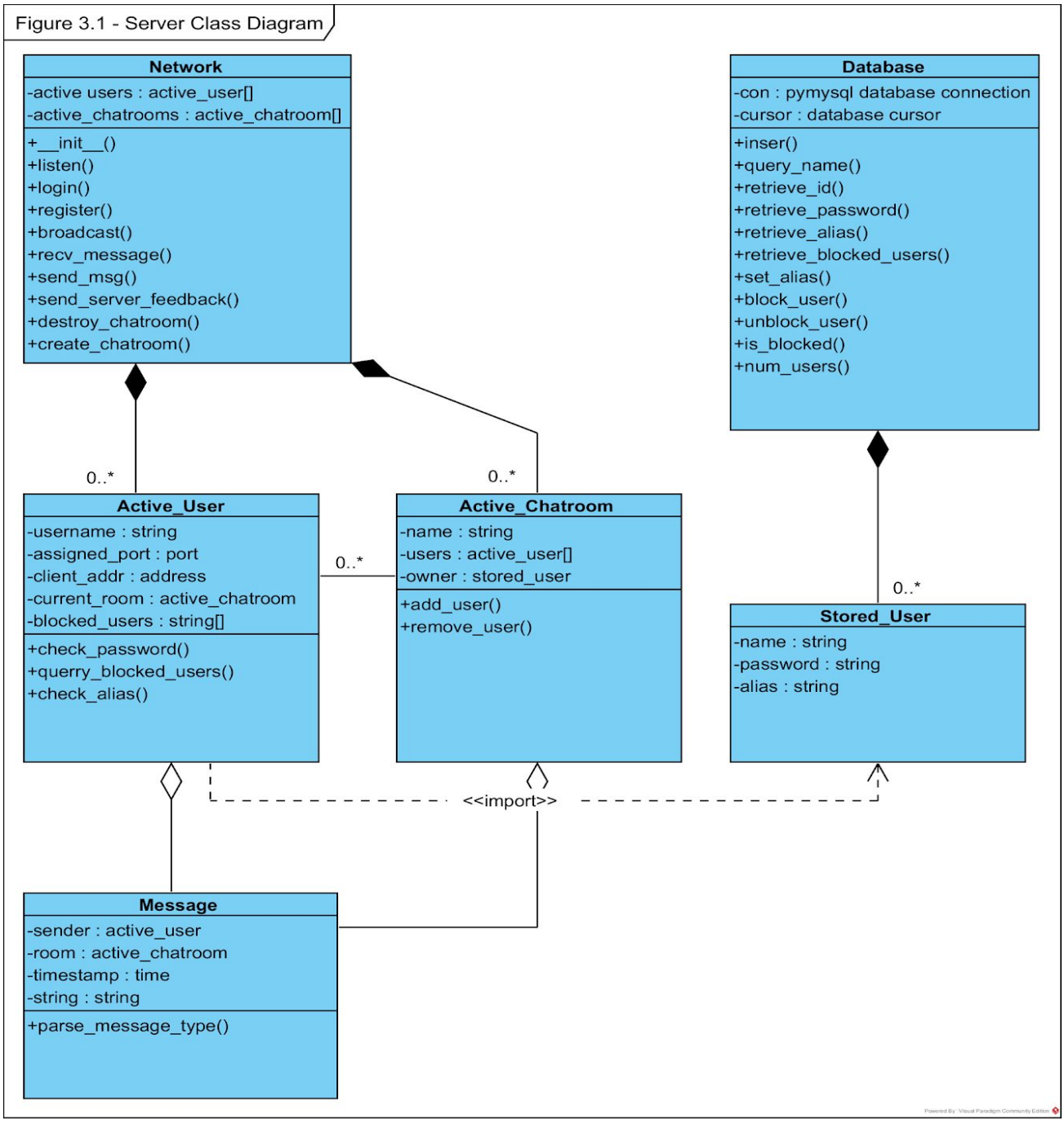
3. System Implementation

Implementation is applied to client and server components in the project. For the server, the network class is the one that contains most of the functions to handle all users and the very basic networking communications. The database class on the other hand connects the main server class with MySQL in which the default configuration is the local host. Implementation of the client is very simple as it contains only two classes that are processed on the server side. The interface class to prompt for user input and sending that as a message, and the network class for communicating messages to and from the server.

3.1 Server Implementation

The server contains the largest most complex class structure within Silencio. The class diagram can below in figure 3.1.

Figure 3.1 - Server Class Diagram



The main control function of the Silencio server starts in the network class. This class keeps track of all active users and active chat rooms on the server. The network class also

contains all the functions to manage the aggregated active users, active chat rooms, and basic connections. The listen() function holds the basic control loop of listening for input and handling it. The login() and register() functions handle the connection of users. The broadcast(), recv_message(), send_msg(), and send_server_feedback() functions handle the sending of messages to entire chat rooms and corresponding single users. Finally, the remaining chatroom functions handle the creation and deletion of active chat rooms. With these functions, the network class acts as a controller within silencio.

Active user and active chatroom classes are what the server uses to keep hold of the data needed for processing active connections. In the case of active chatrooms, it essentially functions as a list of active users to send common messages to. All active users must be in an active chat room to send messages, and all are placed into a default room when they log on. The active chatroom class also contains the function to manage who is in the chatroom. The active user class contains pointers to the associated connection as well as needed user info. Database requests are also found in the active user class for retrieval of user information.

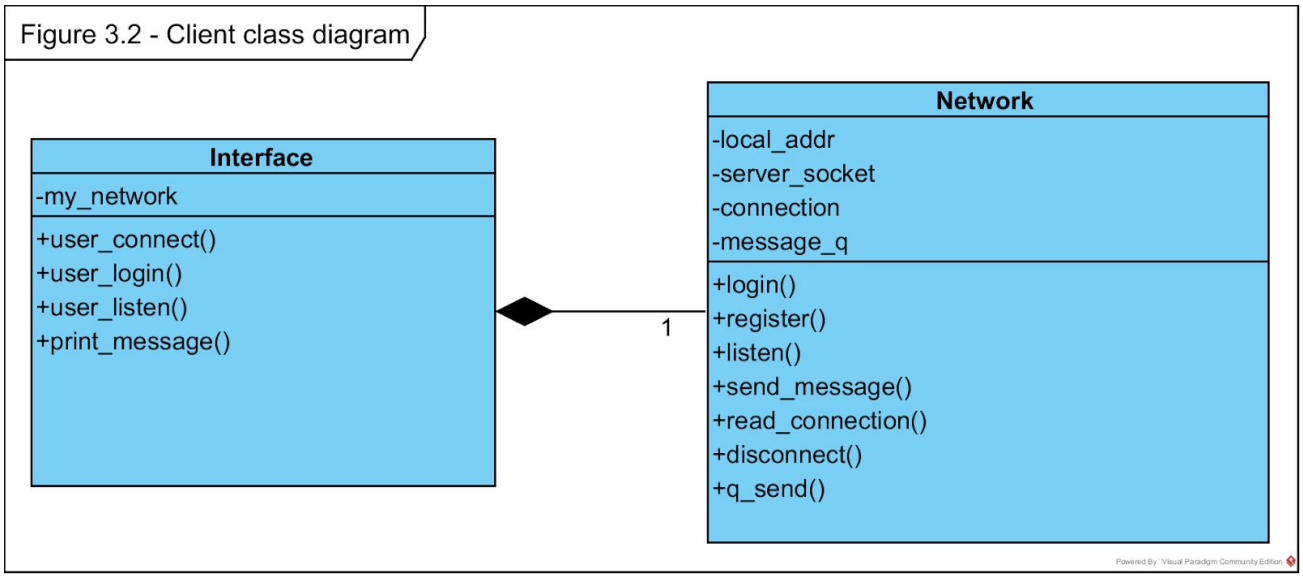
The message class itself is also a simple class. It contains pointers to the user who sent the message, the chat room it was sent in, and also contains the string of the message itself. The only function within a message is to parse the message type, which uses simple regular expressions to find any /commands in the message. Returns from the parse_message_type() command are handled by the network class.

The database class acts as the main class for interacting with the MySQL database, with the stored user class acting as a level of separation between itself and the data requesting users. The database class contains pointers to a database connection, which in our case is configured to be a local connection, as well as a pointer to a database cursor for interacting with the database. All other functions in the database module take advantage of PyMySQL to query and insert data to and from the MySQL database holding user information.

3.2 Client Implementation

Compared to the server, the client side of Silencio is much simpler. The client side of Silencio can be seen in figure 3.2. It is comprised only of an interface and a network class. The interface class functions as a way to constantly poll for both user input and call network functions when needed. The interface class itself contains a pointer to the created network and the functions to send and receive messages from this network.

Figure 3.2 - Client class diagram



3.3 Code snippets

Network listen select loop:

The main control loop sits within the network.listen() function. It is comprised of a select loop on the connected ports with handling for the various states inside. As you can see, the most handling is for received messages in lines 83 to 258 which are excluded. This handling includes handling for incoming chat commands.

```
63      def listen (self):
64          """function that listens to all connections for incoming traffic. Also listens to initial connection port. """
65          readable, writeable, errored = select.select(self.connection_list, [],[])
66
67          #for all readable ports in the list with connections waiting.
68          for s in readable:
69
70              #if initial port has a connection waiting, connect them.
71              if s is initial_sock:...
80
81              #else we read message from the socket and handle accordingly.
82              else:
83                  ...
258
259         #For all writeable ports, send messages if waiting.
260         for s in writeable:...
263
264         #For all errored ports, remove them if they have an associated user.
265         for s in errored:...
```

Broadcast method:

The broadcast method for Silencio simply iterates through user connections within a chatroom and transmits the message.

```
337     def broadcast(self, in_message, in_room_name):
338         """ Broadcast function for bradcasting a message to a chatroom. Takes a message class and a room name string. Returns true if succeeds, false if room is not found."""
339 
340         #lookup chatroom
341         room_found = False
342         for room in self.active_chatroom_list:
343             if room.name is in_room_name:
344                 room_found = True
345                 break
346 
347         #if chatroom does not exist, return false
348         if room_found is False:
349             sys.stderr.write('Failed to broadcast message to non-existant room ' + in_room_name + '.\n')
350             return False
351 
352         #send message to all users in the room
353         else:
354             sys.stderr.write('Broadcasting message to chat room named ' + in_room_name + '.\n')
355             for user in room.users:
356                 if user is not in_message.sender:
357                     self.send_msg(user, in_message)
358 
359         #record to chat log
360 
361         return True
```

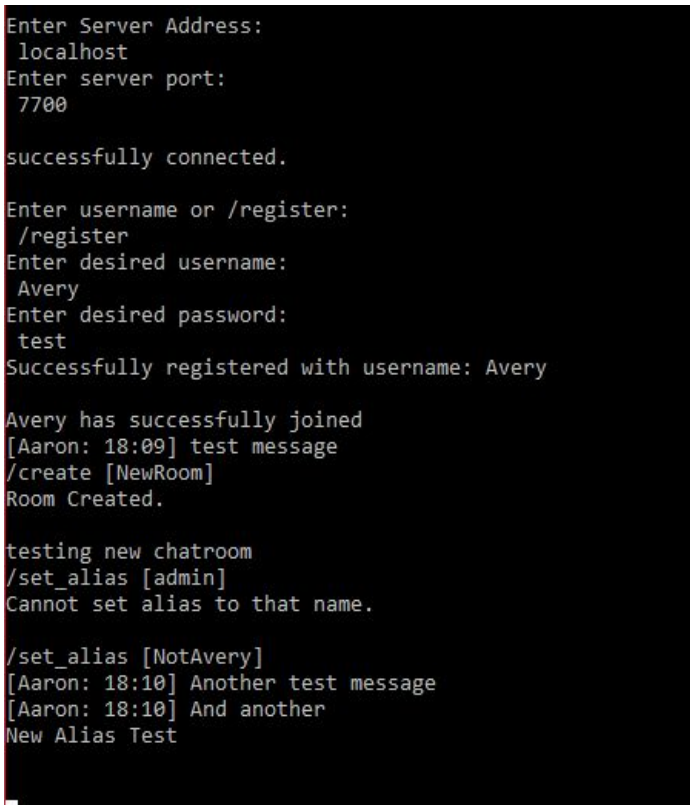
Database queries:

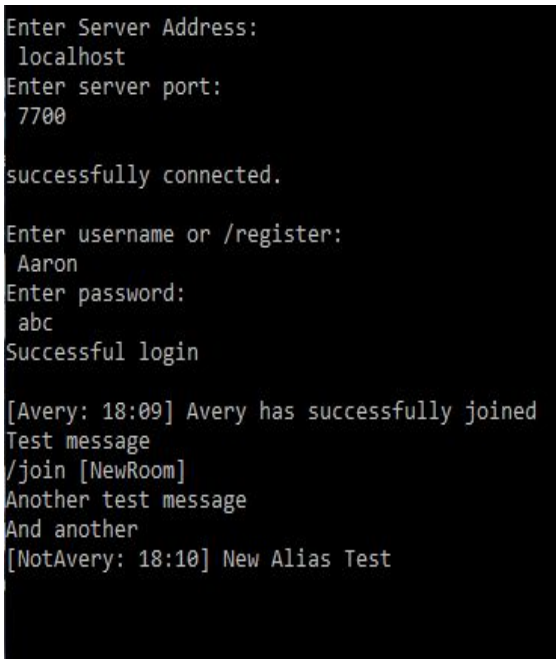
Database queries were all done with PyMySQL. A snippet of this code is shown here.

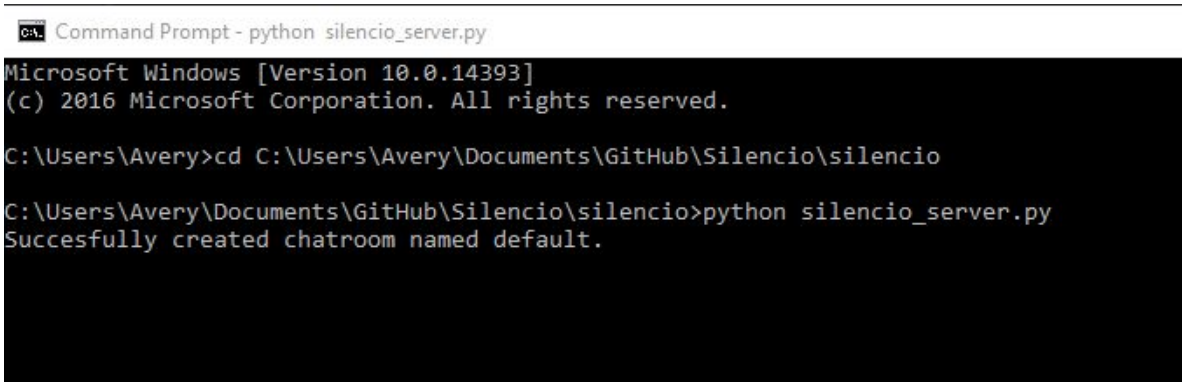
```
71  #Returns alias of a certain user.
72      def retrieve_alias(self, username):
73          self.cursor.execute("""SELECT alias from users WHERE name=(%s)""",(username))
74          temp = self.cursor.fetchone()
75          if temp is not None:
76              return temp[0]
77          else:
78              return None
```

3.4 Screenshots

The first of these presented screenshots is of two clients during their login and exchange on the server. In this, we can see the creation of a chatroom, and setting of an alias. We can also see the Server console window during the creation of the default room for these two to join.







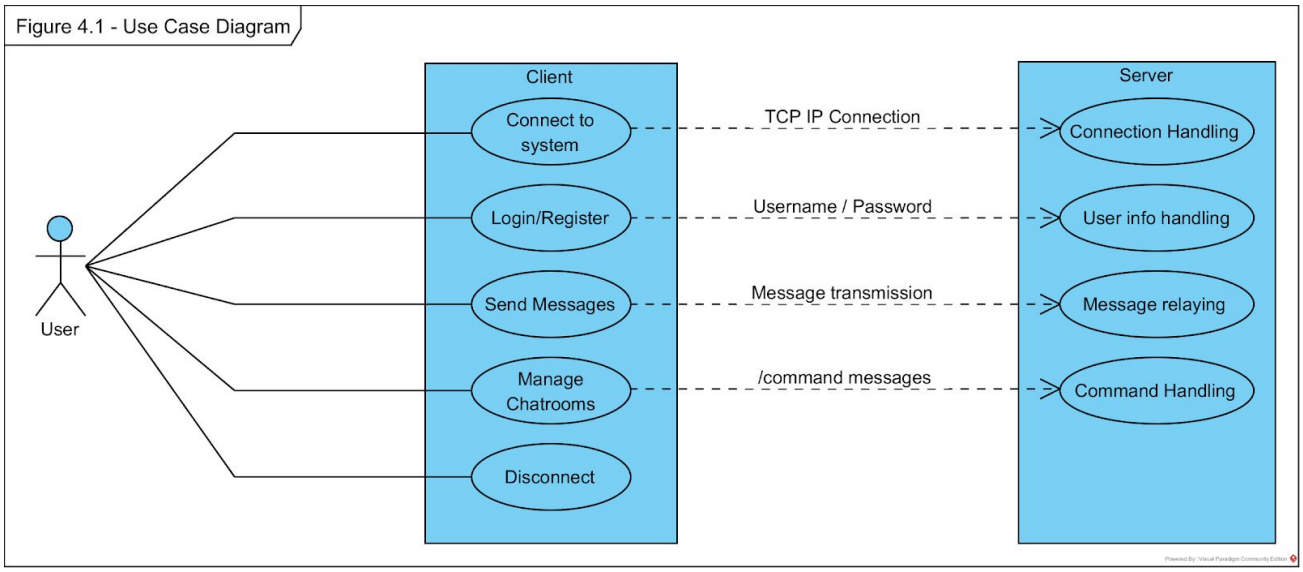
4. Design Choices

For our design decisions on Silencio, the biggest deciding factor was assignment requirements and personal skill of the team. As only one member had experience with

concurrency and none had experience with GUIs, it was decided to go with chat commands. From there, we built up requirements that we felt built up silencio to the basics of functionality. Design decisions were also based in the timing of the assignment milestones. Sadly, with some milestones, the decisions came before some design methods were truly understood.

4.1 Use Cases

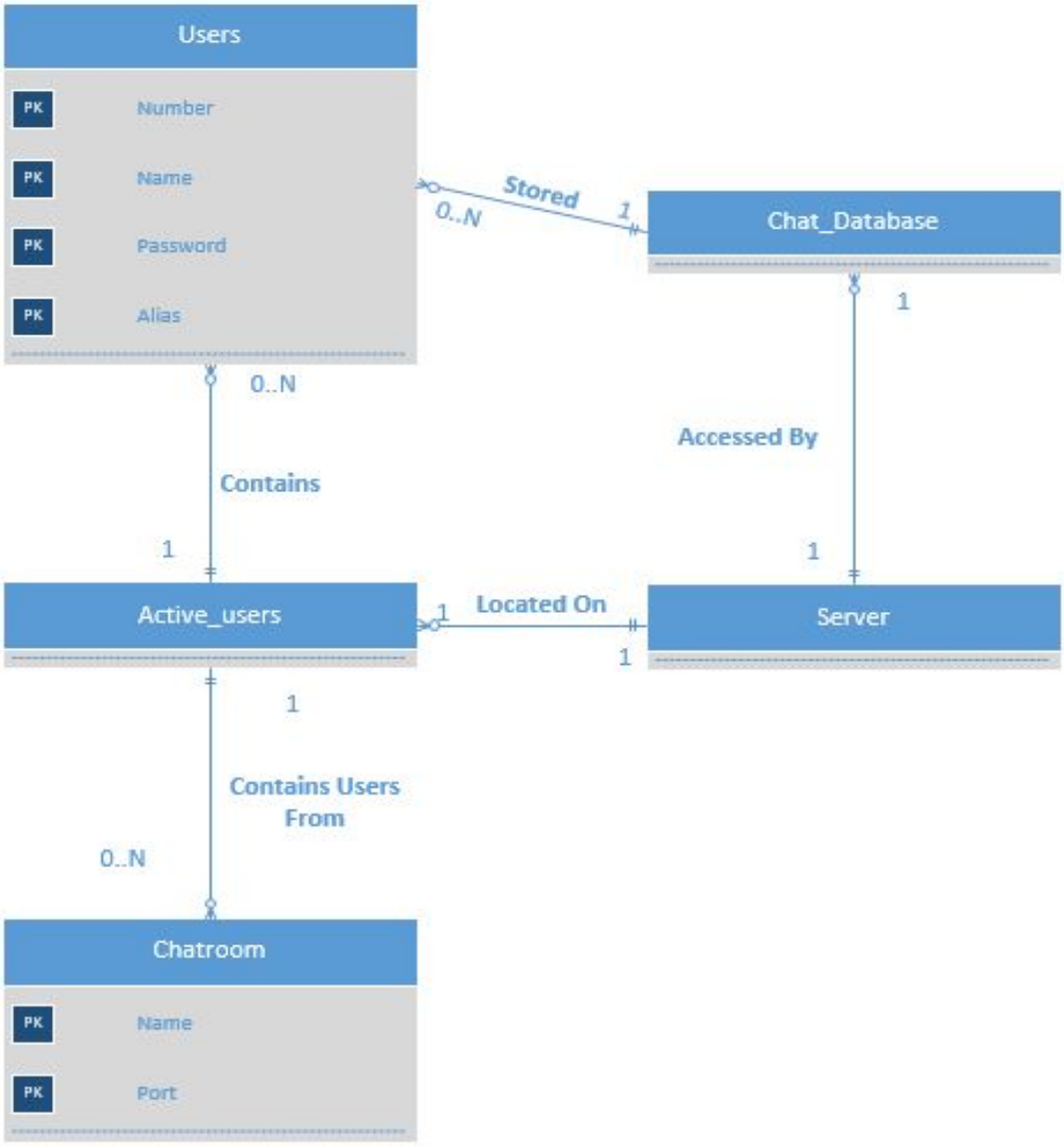
Use cases created the base of the design for silencio. These use cases created our requirements that eventually shaped our design. Some of these use cases can be seen below in Figure 4.1.



The ability to connect to the server and to login came first. A user had to be able to do this to do any more. This shaped the requirements for usernames and passwords, namely requirements REQ 3-1 to REQ 3-5. The ability to send and receive messages were also a basic use case which formed requirements REQ 1-1 to REQ 1-4. From there, we had the chatroom management use cases, which formed REQ 2-1, REQ 2-3, REQ 4-1 and REQ 4-2. Alias based use cases lead to requirements REQ 2-2, which was later abandoned. The alias functionality lacked several requirements that were later met for it. If these were added, they would be about ability for a user to set an arbitrary with commands.

4.2 Databases

Based on our requirements, a database of some kind seemed needed. Requirements REQ 3-1 to REQ 3-5 all required a single unique username and password system which would need to be stored on the server. Losing these usernames and passwords between server sessions would make any sort of testing difficult, so it had to be saved in a file system somehow. A database was designed with relations as shown in the following diagram:



The database is simple and contains user data settings such as username, alias and password for each user. The server stores and retrieves data from this database. Active_users are the ones available in the chat room and who occupy the active memory. The server is able to fetch users from this active memory. The server also fetches blocked users from the database when sending to a user, so it can comply with requirement REQ 3-5.

4.3 Other design methods

Most of the final design was sadly devoid of too many formal design patterns. However, some more basic design structures were implemented.

The network module ended up functioning as a controller for the server, as it controlled the basic i/o loop and handled the different types of commands. In hindsight, a more dedicated controller above this would have been a better choice, but lack of knowledge about these design patterns before class diagrams were built lead to the original class structure.

Original class diagrams had a level of indirection between active users, active chatrooms, and the network, as well as between active and stored chatrooms and users. Although the indirection classes between active users, chatrooms, and the network were deemed unnecessary and removed, the stored user class can still be found as a level of indirection between active and stored users.

The database class itself can be found to function as an information expert, managing all information for users. This design pattern of course came in on it's own with the requirement to interface with a database.

4.4 Meeting Requirements

Final product meets most of the initial requirements identified in milestone 1. There are 20 requirments out of which 17 have been met and 3 have not been met. Details are as follow:

Req#	**Met/Not Met**	**Failure Factors & Time realized**
REQ 1-1	Yes, the server is able to respond to client sent messages.	-
REQ 1-2	No, this was phased out with the change to standard TCP sockets.	Updated with network model update and switch to standard TCP socket use.
REQ 1-3	Yes, the server is able to broadcast all messages to clients that are in the proper chat rooms and not blocked.	-
REQ 1-4	Yes, server does all computation and commands input in client.	-
REQ 2-1	Yes, every username is unique.	-

REQ 2-2	No, no discerning between identical aliases in one active chatroom.	Updated on July 18th as an unneeded feature. May cause some problems in rare cases, but not ones that will break functionality.
REQ 2-3	Yes, only logged in users are placed into chat rooms where messages are sent and received.	-
REQ 3-1	Yes, messages are related to one username.	-
REQ 3-2	Yes, user settings are related to only one user.	-
REQ 3-3	Yes each username cannot be changed.	-
REQ 3-4	Yes, each username is associated with one alias. Each alias is simply the username until changed.	-
REQ 3-5	No blocked users not kept explicitly in server	This was updated on July 18, milestone 4 phase to keep blocked users in the database which is technically in the server
REQ 4-1	Yes the server only allows the owner of a chatroom to destroy it.	-
REQ 4-2	Yes, the server limits all messages to a client based on their chat room.	-
REQ 5-1	Yes, the client gets to see the typed message that they sent.	-
REQ 5-2	Yes, the client displays messages received from server.	-
REQ 5-3	Yes, client side prompts user for login	-
REQ 5-4	Yes, help commands exist on client side class.	-
REQ 6-1	Yes, server handles unlimited number of clients until it crashes.	-
REQ 6-2	Yes, it depends on the server traffic. Normal traffic takes about 2 seconds.	Proper queueing of the messages could provide faster responses, but was cut due to the late discovery of this method.

4.5 Recommendations

We have received some critical and constructive positive feedback from other groups and as such design recommendations, document and general recommendations were pointed out in the previous milestone. Here is a list of these recommendations and the changes we have done to take on them.

Recommendations	**New Changes**
Create class diagram for client side to meet REQ 5.x	From milestone 3 did you make changes according to them why or why not? explain in details use table
Link requirements to design aspects	More clearly defined requirements linked with assets in this final report.
Reduce number of objects in Class model by making it more abstract	Reduced number of classes by removing some unneeded list classes.
Take away dedicated port setup to simplify design	Removed manual assignment of ports and replaced with the build in tcp assignment found in python socket.
Provide textual description for the class model and fix conventions.	Provided more in depth description in this final report.
Elaborate on relationships in the deployment model	Provided more in depth description in this final report.
Define verification code in the class model.	Removed verification code from the networking model.
Use hashed password to improve security.	Disregarded as it is out of security scope for our implementation.
Clarify that each user is related to one client.	Provided more in depth description in this final report.

5. Project Plan

The software development methodology used in this project is the Waterfall because of the nature of sequential milestones, due dates and time constraints. Responsibilities have been specified where possible and each member was expected to perform their tasks. Most of the team was involved in the development of the process in order to have maximum understanding of the functionality of the application and the verification process.

5.1 Project Difficulties

Project difficulties rose up very early in the project. We were able to tackle some of them yet others were left unresolved and materials had to be delivered in a satisfactory form. The huge constraint that we had and led to the following difficulties was the time constraint. As everyone is loaded with other courses, demands and commitments, we had to sacrifice some of our project's quality for those. However, we have put together our best effort considering our circumstances and we are proud of it.

No	**Issue**	**Resolution**
1	Meeting and getting together.	Online correspondence via FB & emails
2	Consistency in documents in earlier phases	Get a clear understanding of the whole project
3	Little time build and test	Working overnight
4	Socket programming & Networking	Follow existing tutorials
5	Lack of understanding of design patterns	Attempts to review designs more and more
6	Cramming milestones until the end	Organize, set up document early & assign roles
7	Lack of OS experience	Use available skills

5.2 Project Timeline

Task Description	Plan Start	Plan End	Assigned to
Milestone 1	**May 9**	**May 23**	
Requirement Gathering	May 9	May 18	Collaborative
Project Plan	May 18	May 21	Collaborative
Milestone 2	**May 23**	**June 16**	
Deployment diagrams	May 24	May 31	Others
Class diagrams	May 24	May 31	Others
Networking model	June 1	June 7	Others
Interface model	June 12	June 14	Omnielle
Database model	June 7	June 14	Others
Milestone 3	**June 16**	**July 5**	
Requirements Critique	June 17	June 24	Omnielle
Quality Attributes	June 17	June 24	Collaborative
Recommendations	June 20	July 2	Others
Milestone 4	**July 5**	**Aug 4**	
Presentation & Report	July 6	July 20	Collaborative
Detailed Tasks Assignment	July 6	July 20	*Refer to section 5.3*
Server Implementation	July 20	July 27	Others
Client Implementation	July 18	July 27	Others
Database Implementation	July 18	July 27	Others

- Project plan is different from that in milestone 1. We had to adapt ourselves to the changes as we proceeded forward.
- Role assignments have changed throughout the semester as we got a clearer picture of the project and individuals' skills. Roles assignment was an attempt to match tasks to each one's familiar area and time constraints.
- Assigned due dates and time estimations have changed to fit our busy schedules.