Documentation for Sup? Project

or-drop-table team
Prof. Ronald Czik
Boston University
Fall-2015

Contents:

1. Introduction: -

1.1. Motivations for the project.

With the development of technology these days, and the prices for personal gadgets are dropped down so almost everyone has a personal computer, there is a significant demand for applications that allow people to contact one another. We or-drop-table team decided to design an application that allows people to do so. Sup? is a desktop program that be used to send/receive messages.

1.2. or-drop-table-team: -

or-drop-table team is consisting of six members. the team members and the their roles are as the following: -

1. Steve Jarvis (**Project Leader**)
   Responsible for organizing and assigning tasks, managing the software repository and continuous integration.
2. Ismail Zoubi (**Quality Assurance, Documenter**)
3. Colin Horowitz (**Software Architect**)
   Designs and documents the overall structure of our software and the high level interactions between components.
4. Le Yiu (**Developer**)
5. Zhen Chui (**Developer, Documenter**)
6. Daniel Alvarez (**Developer**)

During the development process, we found some disparities in programming backgrounds among the team members. This pushed us to reform assigning tasks for each one of us. The result is that everyone participates in the project and does what he/she can do.

2. About Sup?.

2.1. Specifications: -

2.1.1. 1-on-1 conversations via text through a variety of devices.

- Chatting the core functionality of the program, and will be the first priority.
- Conversing with individuals means the program also has to find those individuals online, meaning a contacts list feature is essential.
- Initially the devices will be limited to computers, likely Windows and Mac OSes, but enabling other operating systems and mobile devices to chat using this program is a long term goal (likely longer term than just a semester).

2.1.2. Native client GUI

- Essential to the success of any program, and especially chat programs, a functional and intuitive GUI needs to be implemented.
- The GUI should be usable by users of all ages and technical abilities.
- Longer term, formatting of text as a user preference will be implemented. Initially the user will be limited to a default font.

2.1.3. Distributed, anywhere in the world

- This folds back on point a, where the software should function on a variety of devices. It also means the program has to be able to communicate across the internet, and not just on LAN or other local networks.

2.1.4. Secure conversations

- Security is a major goal of any communication software. After the basic communication framework is implemented, we apply both password login security method and encryption on channel method to ensure private conversations.

2.2. Requirements

2.2.1. Non-functional requirements

a. System
- Send messages between two users.
- Server must be available to remote clients.
- Messages delivered in less than 1 second, ignoring abnormal network latency.
- Communications between users are secured.

b. Server
- Server must support at least 10 simultaneous clients.
- Server must be available 90% of the time.
- Server must provide a list of online contacts.

c. Client
- Usable by users not comfortable in a command shell.
- Able to chat with multiple other (remote) clients from a single local process.

2.2.2. Functional requirements

a. Server startup
- *Associated non-functional reqs*: Server avail. to remote clients, avail 90% of the time
- *Actors*: operator
- *Inputs*: compiled server application
- *Outputs*: server ready to receive messages

- *Normal Operation:* Operator issues command to start server application. Server application starts and begins listening for client connections on a well-known port.
- *Exceptions*: Port could be in use and server will fail to bind, operator will need to investigate or change port number. Changing the well-known port number would need to be announced to all clients somehow.

b. Server accept new connection
   - *Associated non-functional reqs*: Server avail. to remote clients, server support at least 10 clients
   - *Actors*: client
   - *Inputs*: login message
   - *Outputs*: record of active user, status message
   - *Normal Operation:* Client sends login message to server. Server receives the new connection and tracks it in a list of active client connections. Server associates the logged on username with the connection.
   - *Exceptions*: Username could already be taken, in which case server sends back error and drops the connection.

c. Server forwards communication between users
   - *Associated non-functional reqs*: Server support at least 10 clients, messages delivered in <1 second
   - *Actors*: client A, client B
   - *Inputs*: chat message from user A
   - *Outputs*: chat message to user B
   - *Normal Operation:* Client sends chat message to server, addressed to another logged on client. Server looks up the receiving client and forwards the chat message to that user.
   - *Exceptions*: Receive chat message from user who is not logged on or to user who is not logged on. In either case, drop the message.

d. Server recycles usernames
   - *Associated non-functional reqs*: Server support at least 10 clients
   - *Actors*: client
   - *Inputs*: client connection closed
   - *Outputs*: associated connection removed from map and username available again
   - *Normal Operation:* Client sends logoff message to user or unexpectedly goes offline. Server notices and removes the client from collection of online users.

- *Exceptions*: Receive logoff message for user who is not currently recorded as online. Ignore the request and close the associated connection.

e. Server provides list of online contacts
   - *Associated non-functional reqs*: Server provides list of online contacts
   - *Actors*: client
   - *Inputs*: client request for contacts
   - *Outputs*: message to client containing list of online contacts
   - *Normal Operation:* Client sends contacts query to the server
   - Server interprets the message and replies with a message containing a list of all the online contacts that can be chatted with.

f. Client startup
   - *Associated non-functional reqs*: Send messages between users
   - *Actors*: user
   - *Inputs*: user start application, desired username
   - *Outputs*: client application ready to sent messages
   - *Normal Operation:* Client prompts user for username and uses it to log in to the server. After login client downloads list of online contacts from the server.
   - *Exceptions*: Username is taken, client should ask for another name.

g. Client commands handler
   - *Associated non-functional reqs*: Client can chat with multiple remote clients
   - *Actors*: user
   - *Inputs*: user command
   - *Outputs*: appropriate action for command dictated by the user
   - *Normal Operation:* User enters a command for interpretation by the client application. The client interprets this command and performs the expected action. Supported commands should include switching conversations, logging off. Commands probably need a particular escape character to be recognized as such.
   - *Exceptions*: User enters a command that is not understood. The client should display an appropriate error and be ready for new input.

h. Client send message
   - *Associated non-functional reqs*: Client can chat with multiple remote clients, send messages to remote clients
   - *Actors*: user, server
   - *Inputs*: user-entered chat message

- *Outputs*: message to server for dispatching
- *Normal Operation:* Client application is in a state to receive a chat message for a particular user. User enters a chat message. The client application constructs the appropriately formatted message and sends it to the server.
- *Exceptions*: Remote user/contact has gone offline. Client should notify the user.

i. Client receive message
   - *Associated non-functional reqs*: Client can chat with multiple remote clients, send messages to remote clients
   - *Actors*: user, server
   - *Inputs*: chat message from server
   - *Outputs*: message displayed to client
   - *Normal Operation:* Client receives a chat message from the server. It's parsed to learn the sender and displayed to the user.
   - *Exceptions*: Message is of invalid format. That'd be an internal error, so possibly display an error message to use. Drop the message.

2.3. Design Patterns

2.3.1. Model-View-Controller (MVC) pattern

We use ***Model-View-Controller*** pattern to isolate messaging logic from user interface. The model represents the information of the application and rules to manipulate operations. The view corresponds to interface, which is GUI in this application, as we show in the demo. The controller deals with the details involving communication between model and view, like user login, text message in GUI window etc..

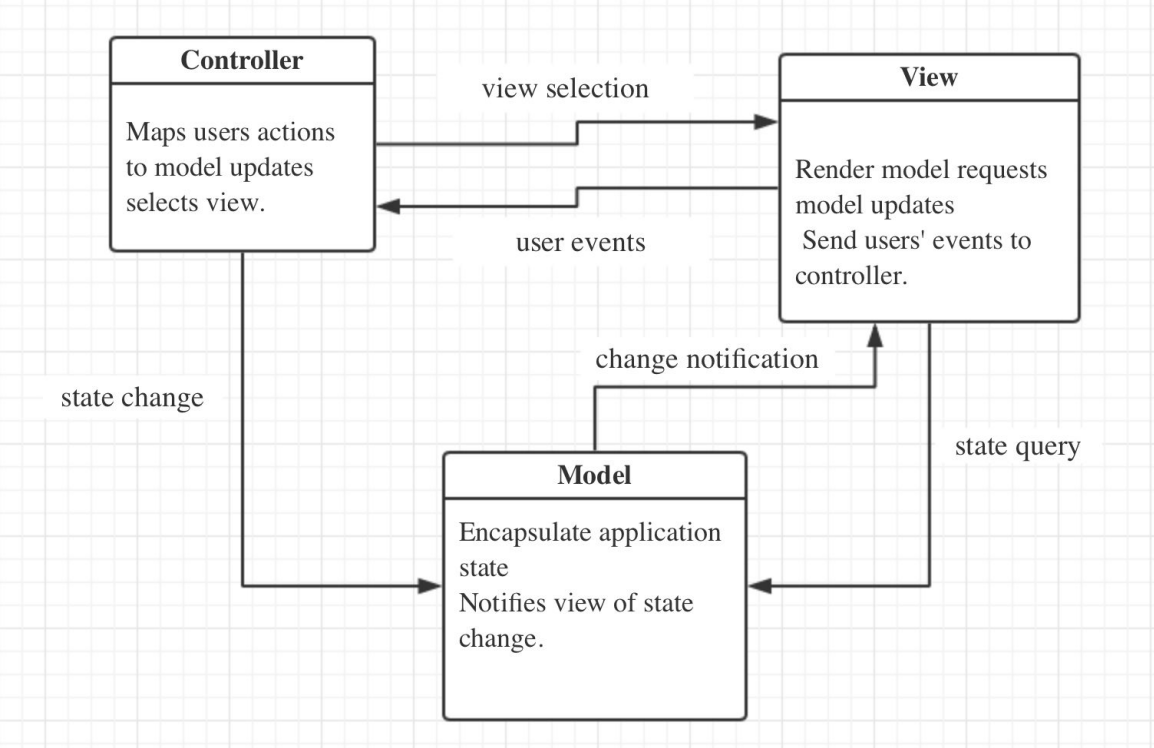


Figure 1. MVC pattern

2.4. Key Algorithm

Within the Sup? application there were lots of algorithms implemented. For example when a username was typed in the login window, we need to compare it with the contact lists stored in the server.

And one of the most important algorithms that be implemented in is the one implemented to obtain ***anti-spam*** feature. This requires ability to select those hateful speech and unwanted conversation, or those users sending out message too quickly to be human,

and then block them from the list to fulfil the security requirements. So here we could use prefix tree(trie) to implement sequence matching and delete them from the list.

Another one is ***encryption***, we are using Java's built-in security libraries to encrypt all communications between the clients and server with SSL, which will be talked more on in section 2.7.

2.5. Quality Assurance Strategy

2.5.1. Test Design

a. Tests will be designed as the code is being designed in order to ensure the software meets the specifications.
b. Tests will cover 3 levels in a pseudo-V model:
    - Unit testing for individual pieces of the code, as each piece is being designed
    - Component testing for adding components to the software, as each component is being added to the software
    - Acceptance test to ensure entire program will meet the customer's demands for each version
        - These tests are defined in the Appendix, section 7.4.

2.5.2. Other metrics

a. Code review
    - Once a change has been made, at least one other person of the team will look at the code developed to make sure it is both doing what it is supposed to, and easy to understand for facilitated maintenance.

2.5.3. Measure those metrics

a. Continuous integration
    - The testing will be applied as the programmers add code to the GitHub repository.

2.6. Sup? Process Model

We, or-drop-table team decided to use incremental model to develop Sup? Moreover, the specification, and development of each iteration's functionalities is Agile.

Because we have only one semester to develop Sup?, we divided Sup? into three iterations, and specify the requirements of each iteration. However, by the time we move on, other functionalities emerge and that's why we chose agile.

2.7. Security insurance

Security is one of the major tasks of this communication . After the basic communication framework is implemented, we apply both password login security method and encryption on channel method to ensure private conversations.

2.7.1. Login with password

Figure 2

A login window with password required on GUI design is implemented. Once a password is typed in correctly, and the typing would not be displayed as plain text to ensure that users' password wouldn't be seen by one else. And if a wrong password is typed in, the login tempt would be prevented so that the basic security need is fulfilled.

2.7.2. Encryption

After the login in with password strategy applied, Sup secure the conversation with public key encryption and SSL(secure sockets layer).

SSL is a protocol that operates directly on top of TCP(transmission control protocol). When using SSL correctly, all an attacker can see on the cable is which IP and port you are connected to, roughly how much data you are sending, and what encryption and compression is used. He can also terminate the connection, but both sides will know that the connection has been interrupted by a third party.

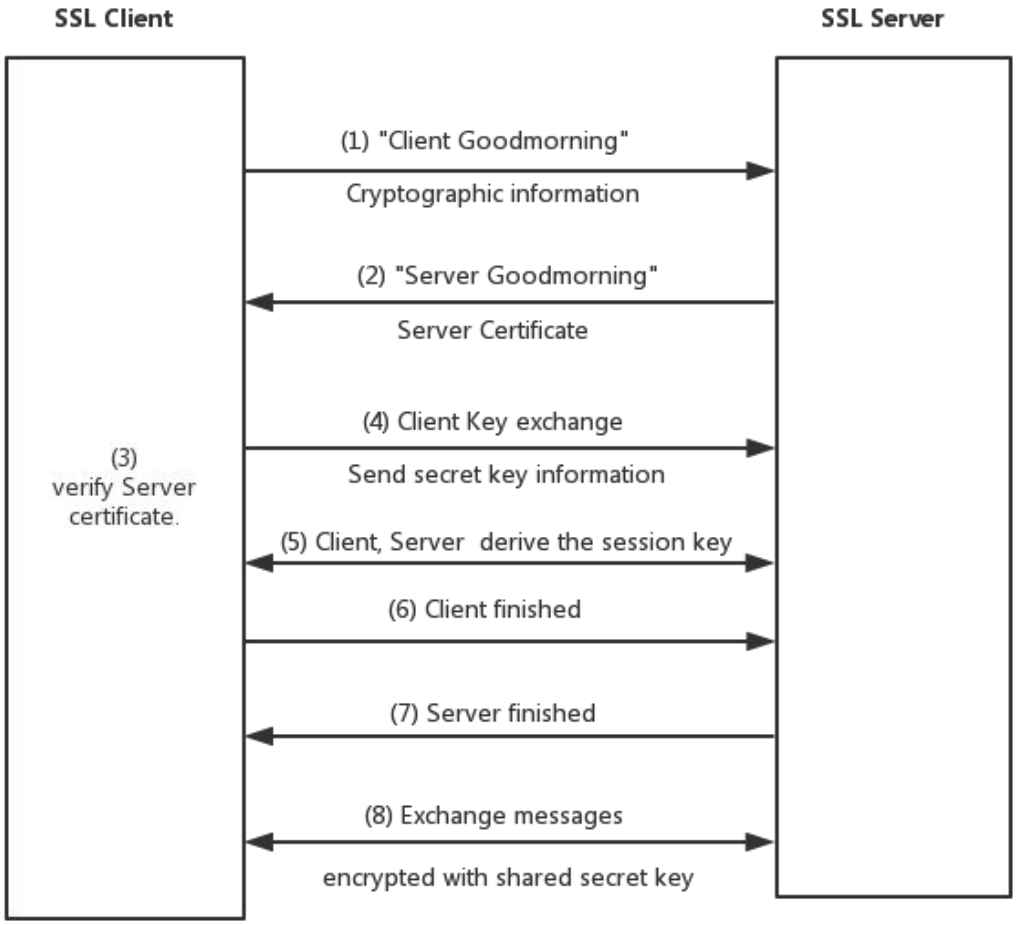
We use Java's built-in security libraries to encrypt all communications between the clients and server with SSL. And here is the certificate we need to build showed in Figure 3 for example:

```
sjarvis@verona ~/dev/worksp...sup/server/keys $ keytool -genkey -alias supcert -keyalg RS
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  or-drop-tables
What is the name of your organizational unit?
  [Unknown]:  cs673
What is the name of your organization?
  [Unknown]:  BU
What is the name of your City or Locality?
  [Unknown]:  Boston
What is the name of your State or Province?
  [Unknown]:  MA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=or-drop-tables, OU=cs673, O=BU, L=Boston, ST=MA, C=US correct?
  [no]:  yes

Enter key password for <supcert>
    (RETURN if same as keystore password):
```

Figure 3. Certificate

Here is a diagram that illustrates the process exchange messages in the program by using SSL:



-

Figure 4

3. Sup? Architecture

3.1. Sup? Modeling

the UML diagrams below from to show the classes in our design. Due to that there are Client side and Server side, we separate the whole classes into different sections.

3.1.1. High-level design:

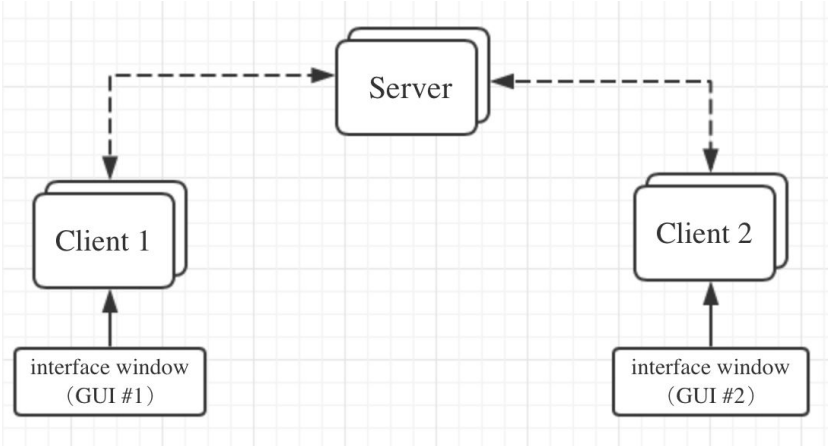


Figure 5. High level design

3.1.2. Class design

a. Object Server:

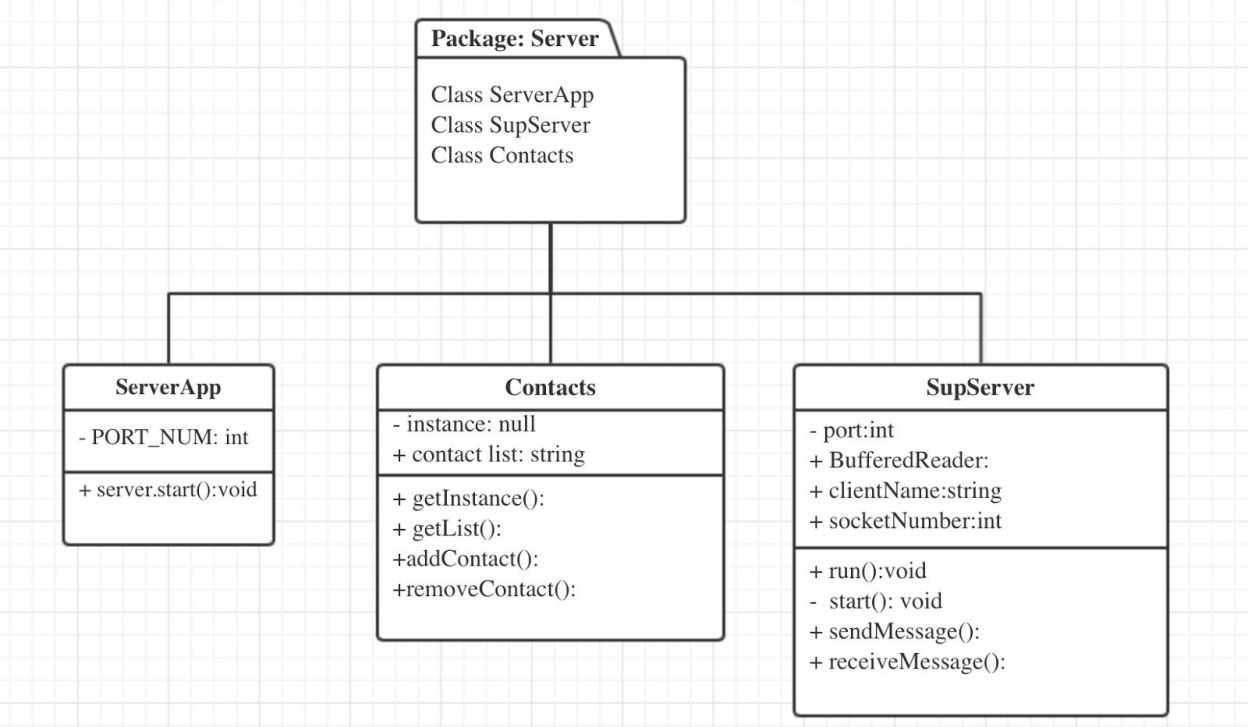


Figure 6. Object Server classes

ServerApp is the main class for the server application. It serves to instantiate and start the server's business logic.
Contacts is a class that save the contact lists, which is safe to edit by any thread of execution.
SupServer is the main business logic for the server. This class starts and continuously listens for new connections from clients. Each connection is tracked and chat message are forwarded as appropriate during normal execution. In this class, ***run*** method is the logic that will be (continually) executed during this thread's lifetime. And ***start*** method start the server to listen at the expected port, and ready to accept new client connections.

And there is also a subclass named "***SupClientHandler***":

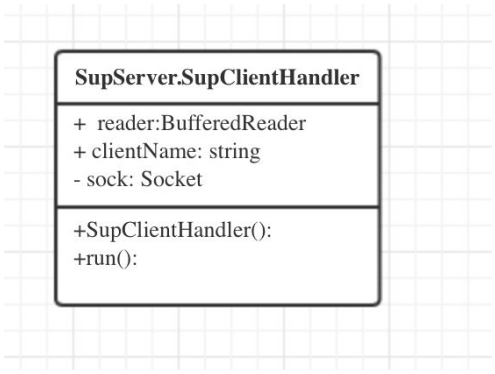


Figure 7

A new ***SupClientHandler*** is created and operates in its own thread, when the server gets a new connection request from a client. it handles communications from the connected client. Commands and chat messages can be received and handled in this class.

b. object Client:

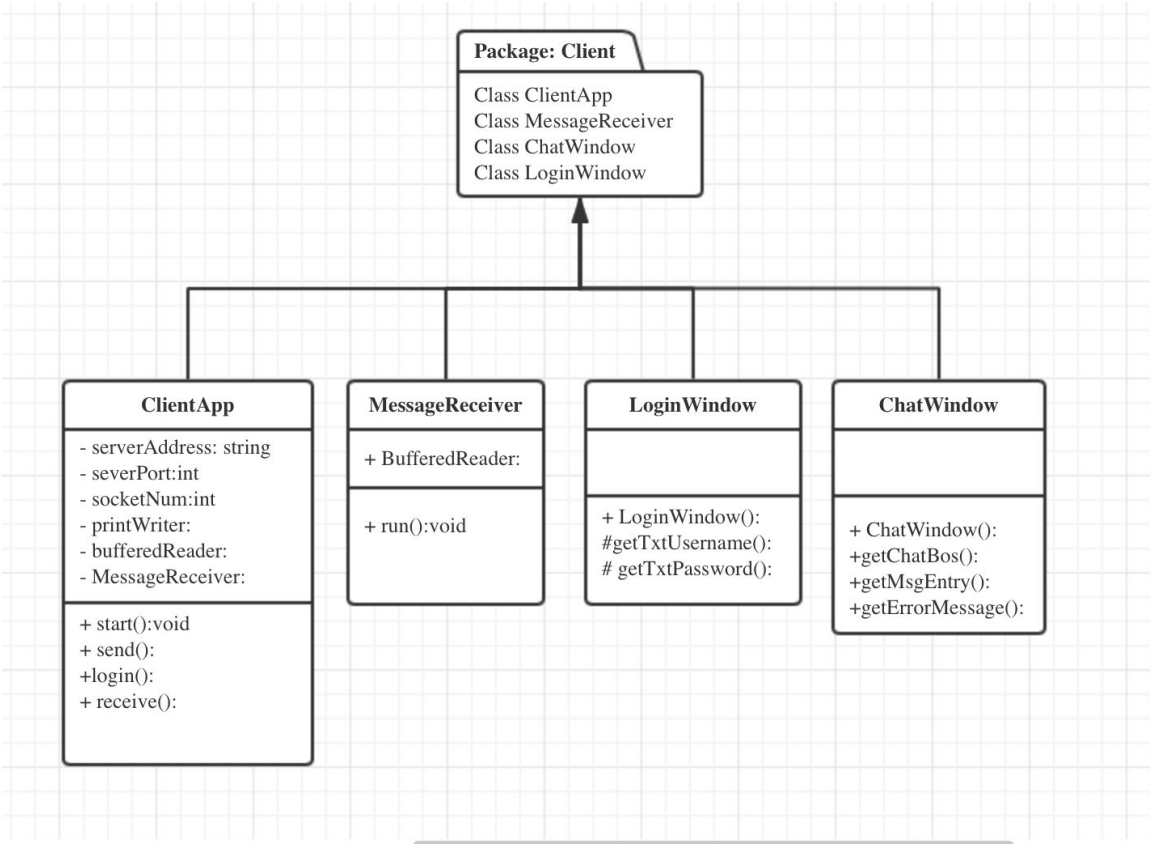


Figure 8. Object Client classes

ClientApp is the main class for the client application. In ***start*** method, it starts the client logic executing and creates the connections needed and begins accept input from the user.
MessageReceiver provides logic to continuously receive message from the server and provides notification to the user. In ***run*** method, while the client is running, read the message from the server and display for the user, and handle status messages.

LoginWindow and ***ChatWindow*** both are interface classes that take charge of communication between users and the server. We use JFrame API to build our GUI window, and in ***LoginWindow*** class, ***getTxtUsername*** method and ***getTxtPassword*** method get the strings from user input, while in ***ChatWindow*** class, ***getChatbox*** method and ***getMsgEntry*** method use JTextField to set up an area for text input, and getErrorMessage uses JLabel to label a error message.

3.1.3. Sequence Diagrams for some use cases.

Some use case of the program are: -

A. a client registers into the server.

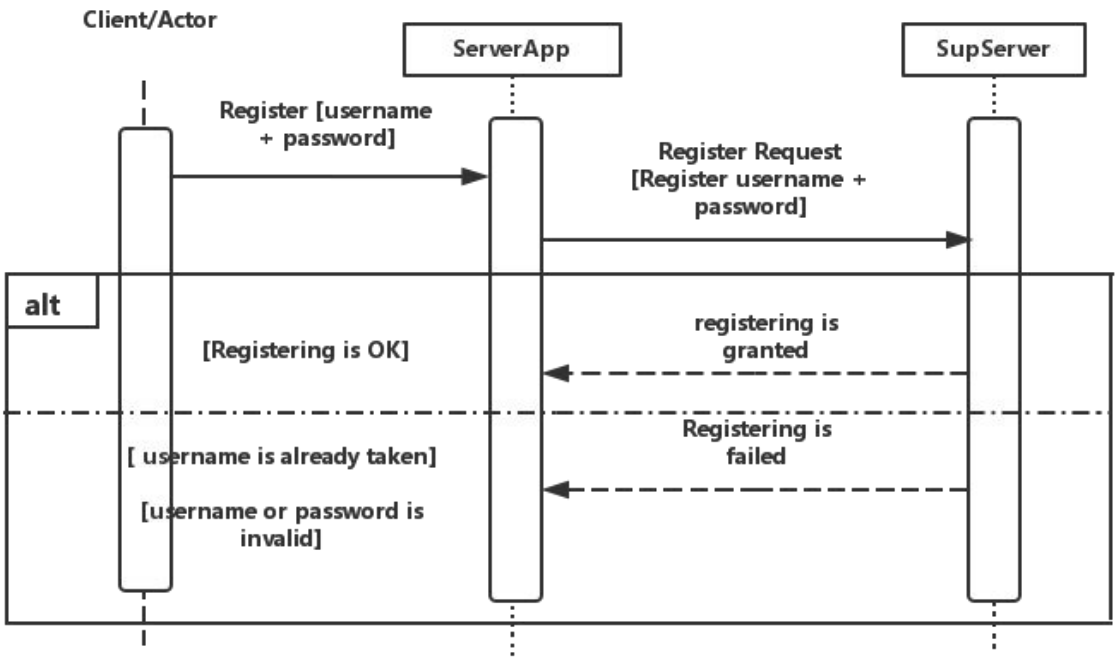


Figure 9

B. a client logs in into the server.

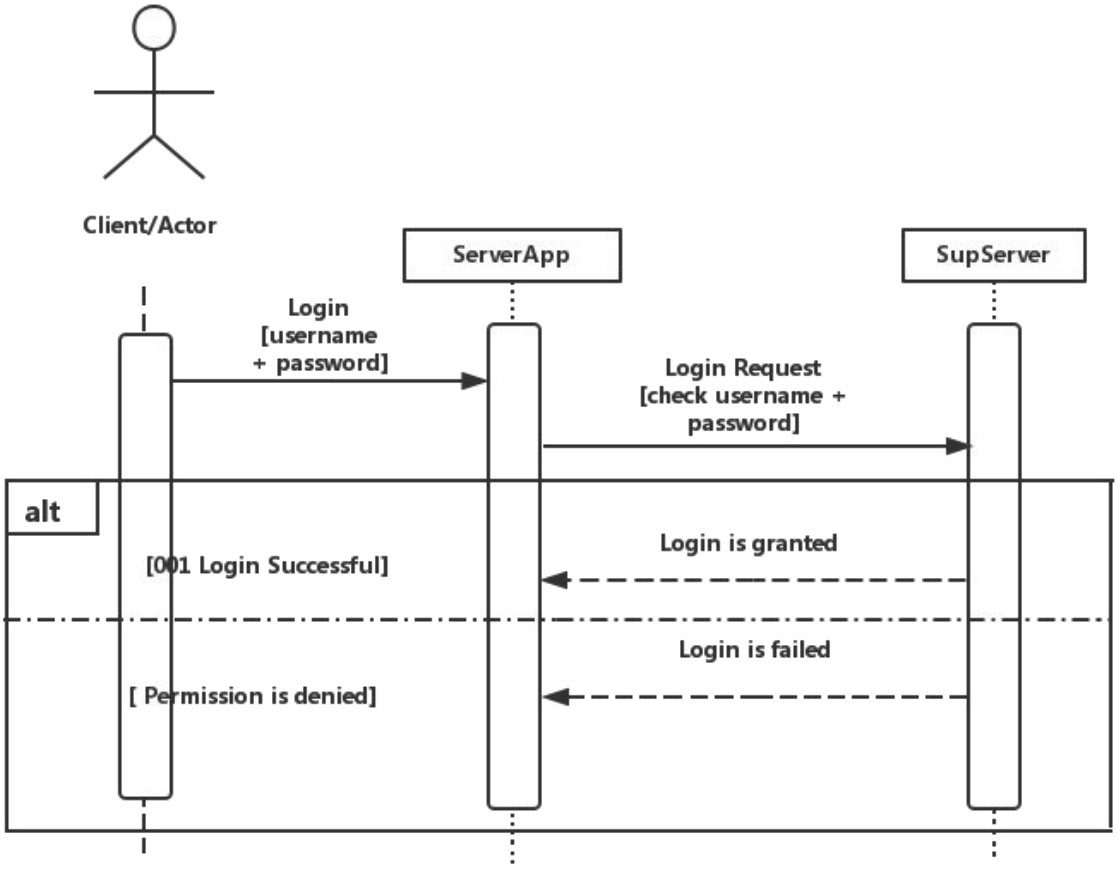


Figure 10

Responding by one generic message (permission is denied) in the case of the failure of logging in is to secure the application. for example, if we reveal the cause of the failure was in the password, this helps the malicious users to know that the username is correct and the password is wrong. Therefore, we echo a generic message.

C. a client checks other client's status.

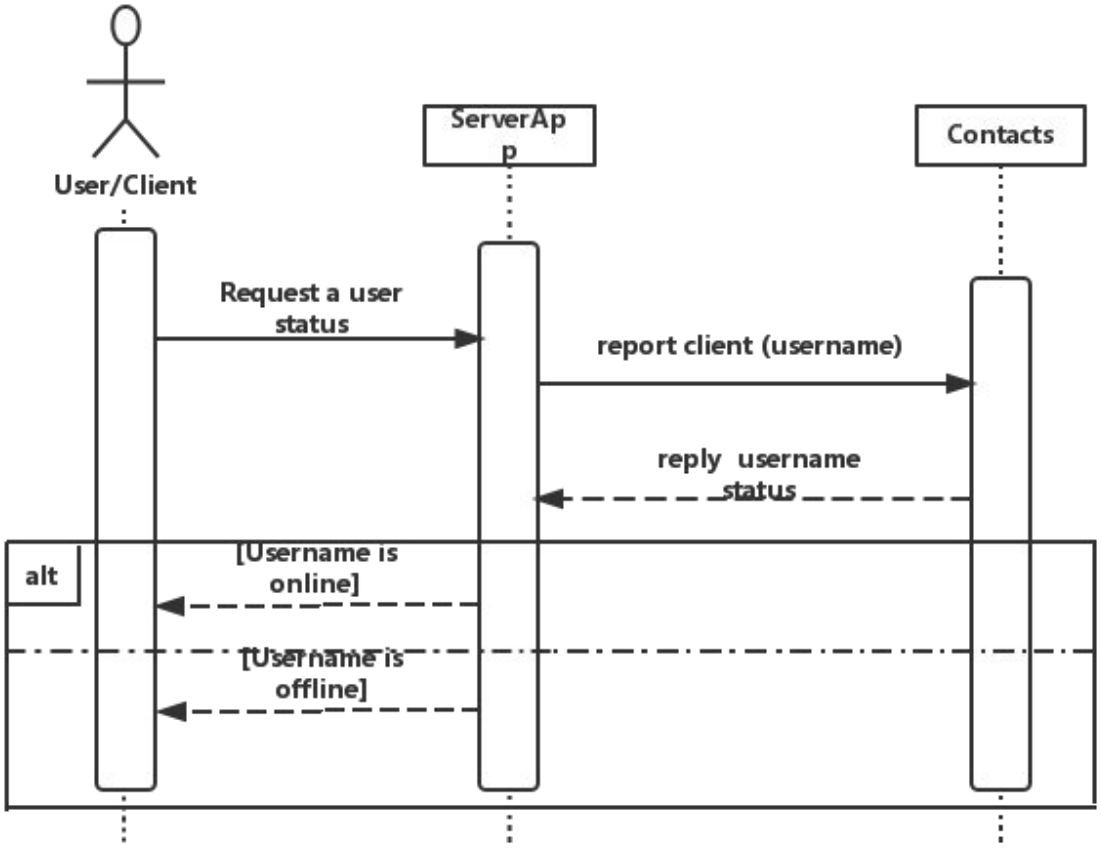


Figure 11

D. a client sends a message to other client.

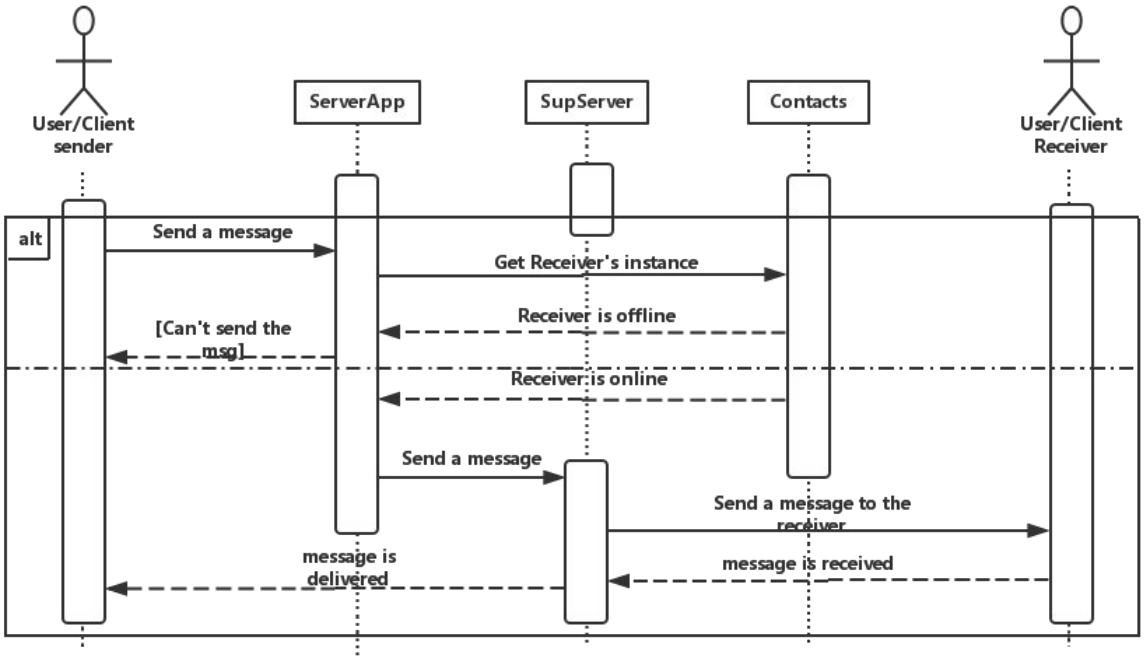


Figure 12

4. Process Management

We use Agile approach in our project, which is an iterative and incremental process. Since there are at most 10 weeks to produce the whole Sup? application, we divided process roughly to 3 iterations, and in first one we succeeded to build basic functional P2P chat room without GUI. Then the second and third will be adding more features to the basic frame and ensure the security concerns. And the main purpose of these 3 iterations is to edit according to the change customers' requirements, which is also the main feature of agile method.

4.1. Source control

We manage our source code on ***GitHub***. It uses repository pattern to manipulate independent components. Changes made by one component can be propagated to all components, and all data can be managed consistently.

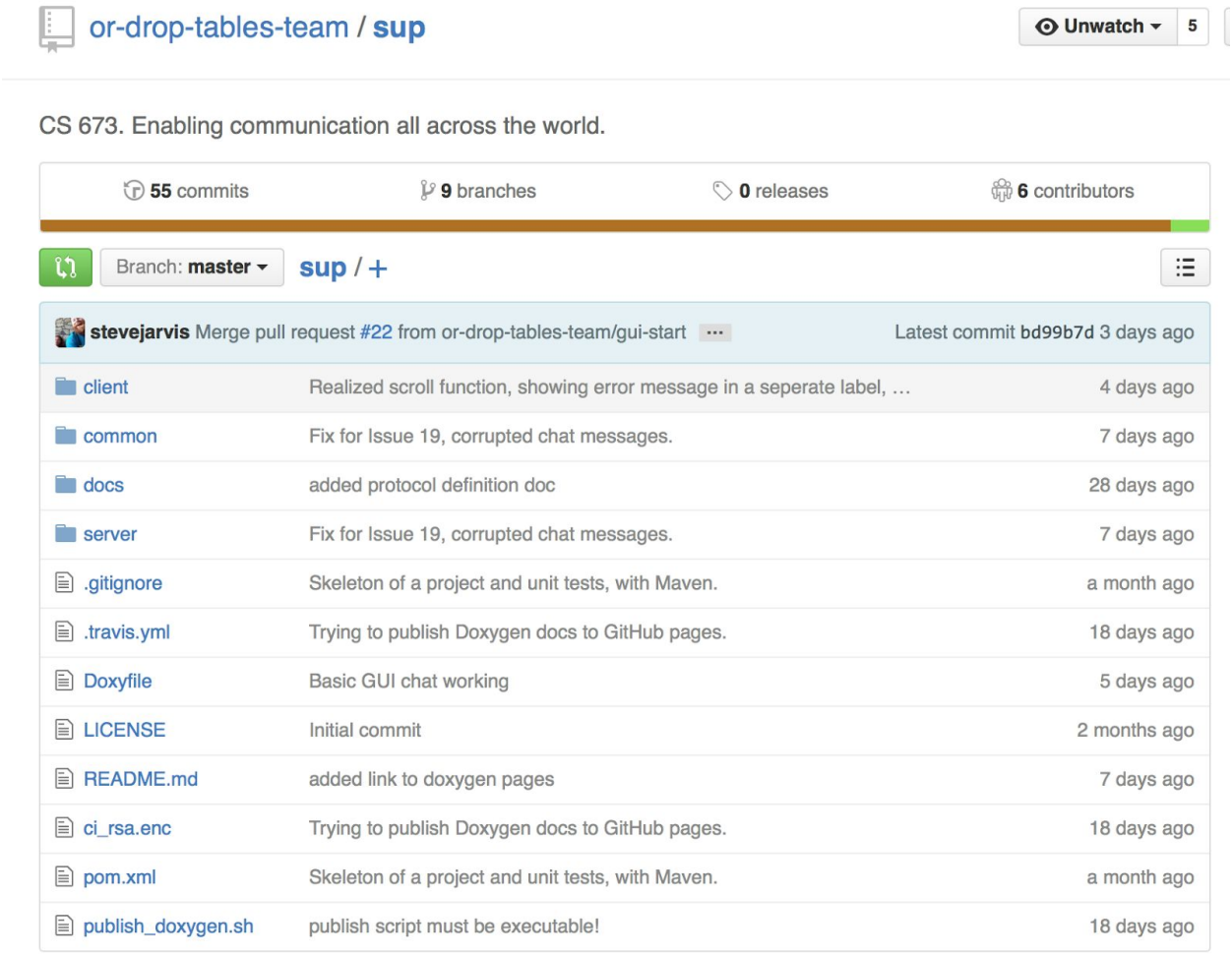


Figure 13. Source control on GitHub

4.2. Task management

Task management was done on ***Trello***. We separate the whole progress into 4 lists, *Backlog, Active(current goal), In progress(right now),* and *Finished*. First we put all tasks in *Backlog* list, then with the moving on progress, the task will finally reach *Finished* list, which is the right-most part, so that we can track the progress of every single task.

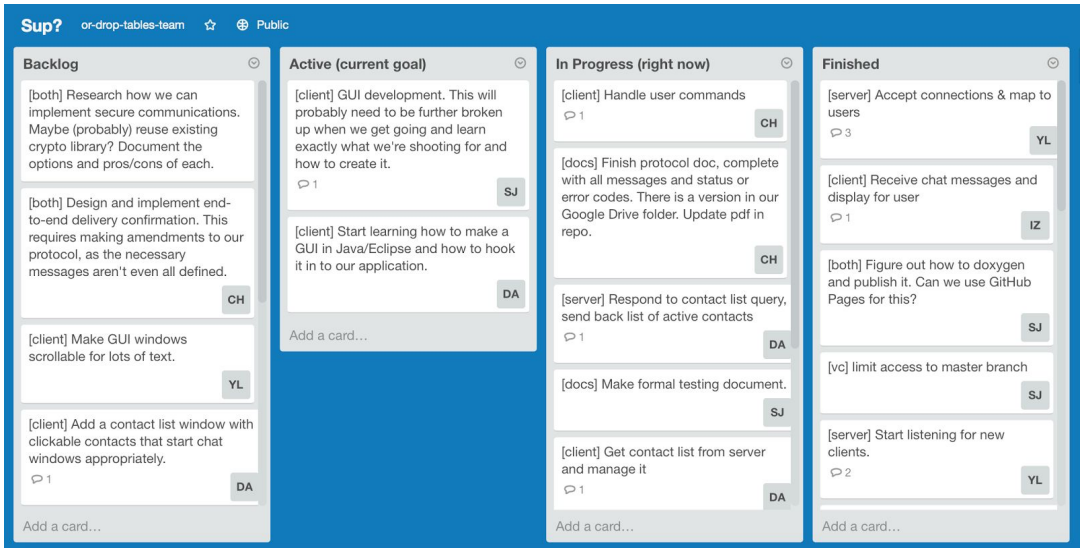


Figure 14. Trello for task management

4.3. Communication tool

Also a cross-platform integrated communication tool named ‘slack’ has been using to ensure a daily-based connection between group members.

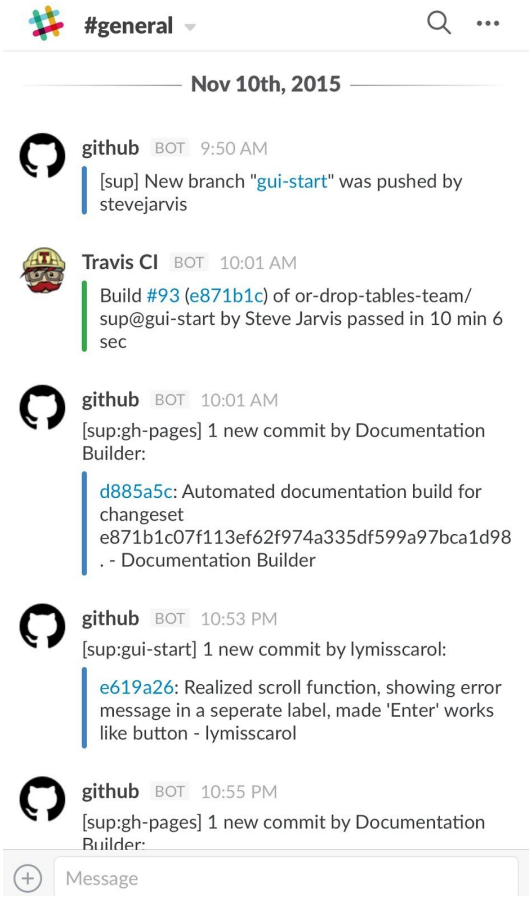


Figure 15. Communication on Slack

5. Reference
6. Glossary
7. Appendix

7.1. Status Code

Status codes are 3 digit numbers encompassing a variety of necessary messages. Statuses beginning with a 0 are good messages. The vast majority of status messages will use status:

000: Everything okay

Statuses beginning with a 1 indicate client problems. Statuses beginning with a 2 indicate a server side problem.

The second digit indicates the scope of the problem. The last digit is used to differentiate the status codes.

- x00: Issues relating to protocol
- x10: Issues relating to login
- x20: Issues relating to contact queries
- x30: Issues relating to the chat messages
- x40: Issues relating to connection status

The major current status codes include:

- 000: Everything Okay
- 101: Protocol incorrect [This is server message to client]
- 201: Protocol incorrect [This is client message to server]
- 011: Login Successful
- 111: Username taken (Login failed)
- 112: Username invalid (Login failed)
- 211: Server Full (Login failed)
- 212: Permission Denied (Login failed)
- 031: To-user received message
- 131: User offline (Message not delivered)
- 132: User not a contact (Message not delivered) [Sent from server back to client if:]
- 232: From-user not a contact (Message not received) [Sent from client to server to reject message]

7.2. General Protocol

All messages are sent in ASCII text. Variable (e.g. actual chat messages) or unprintable ASCII (e.g. EOT) are enclosed in brackets, <>, to denote appropriate substitutions. All messages are terminated with an end of transmission character, denoted “<EOT>”, ASCII value 0x04, and space-delimited.

Both the server and the client will respond to other messages with a status messages, detailing the success or failure of the original message. These statuses take the form of codes, with the code definitions being in a shared library in the common code. The format of the messages is as follows:

status <status code> <message><EOT>

A. **Login Protocols:**

After successful establishment of a TCP connection, the client can send a login message. Before logging in, a user must have successfully registered a username and password with the server.

register <username> <password><EOT>

The registration message shall be responded to with either success or a reason for failure, which the client can take action on.

The client can log in by sending a login message to the server of the form:

login <username> <password><EOT>

If the login is successful, the server shall respond with the universal success message. If it fails, there are status codes defining why the login failed, but will be intentionally vague and a bit generic in order to not aid malicious login attempts.

B. **Contacts Query Protocols:**

Clients can query a list of all online contacts. The message format is:

get contacts<EOT>

The server shall respond with a status message, and if successful, a subsequent message including a list of space-separated contact names. The format for the contact list message is:

return contacts <username1> <username2> <...><EOT>

The client can also query the status of a particular contact (or several). The message to do so has the form:

get contacts <username1> <username2> <...><EOT>

with the same return message as with getting all contacts. If more statuses were to be implemented than simply online or offline, a return of the form <username1>,<status1> <username2>,<status2>... would be used.

C. Sending and Receiving Messages Protocols:

Once logged on, a client can send chat messages to other clients. These messages are routed to the destination user by the server application. A chat message from one client to another shall have the format:

send <to username> <message><EOT>

The server will then forward the message to the destination contact, and the forwarded message shall have the format:

recv <from username> <message><EOT>

D. Logging Off Protocols:

When the user wishes to exit the client application, the client shall notify the server of the log off. This message shall have the format:

quit <username><EOT>

The user should also wait to receive the status message from the server before ending the process.

7.3. Database Schema

The authentication data is stored on the server in an SQLite3 database. This database has a single table, “USERS”, with username and password information used to authenticate clients.

USERS

NAME (text)
PASSHASH (text)

The passhash is a string representation of the hexadecimal SHA256 hash of the user's password.

7.4. Release Test Plan

To be confident in the stability and capabilities of Sup we need to subject it to a regular, comprehensive, and consistent set of tests to verify its behavior. While this includes the regular execution and maintenance of unit tests, this document will focus on system and user tests, which will require at least some level of manual intervention.

System Tests

System tests verify the correct operation of the system as a whole. They encompass such things as correct message formats, proper error codes, and stability and uptime of both the client and server.

Here are a series of system tests:

1 - Legal Username

Reason: The username used to log in must be safe for parsing. It should not contain any special characters that may cause the parsing of later messages to fail. Accepted usernames shall consist of only alphanumeric characters.

Steps:

1. Attempt to log in with various usernames containing the following characters:
   a. <space>
   b. ,
   c. ~
   d. !
   e. @
   f. #
   g. $
   h. %
   i. ^
   j. &

k. *
l. (
m.)
n. \
o. ‘

Expected Results: All attempts to log in are denied. Appropriate error is returned from server.

2 - Unique Username

Reason: A username must map to at most one user.
Steps:
1. Log in on one client with any legal username.
2. Attempt to log on with the same username on a second client.

Expected Results: The first log on is successful, the second fails. Appropriate error message is returned by the server.

3 - Long Chat Message

Reason: Users can send chat messages of arbitrary length. We want to be sure long messages are delivered successfully.
Steps:
1. Log on to two different clients.
2. Send the United Nation’s Declaration of Human Rights from the first client to the second. Ensure the text is delivered in its entirety.
a. The Declaration can be found here: http://www.un.org/en/documents/udhr/

Expected Results: Text is delivered in tact at receiver.

4 - Empty Chat Message

Reason: To ensure stability and expected behavior.
Steps:
1. Log on to two clients
2. Send an empty message from one to the other

Expected Results: Recipient receives an empty message.

5 - Server Uptime

Reason: Our chat server should have the highest possible uptime and availability.
Steps:
1. Start the server
2. Start a client and successfully log on to the server
3. Repeat step 2 once a day for 5 consecutive days (without restarting the server)

Expected Results: Server is available and responsive at each attempt (login successful).

6 - Server Scale

Reason: We ultimately want to be able to accommodate a large number of users.
Steps:
1. Start 10 instances of the client application
2. Log all instances into the same instance of the server
3. Send a few messages between client applications

Expected Results: All clients can log in. Messages are delivered without issue.

7 - Server Recognize Dropped Connection

Reason: Clients may disappear unexpectedly. Server must handle this and recycle resources.
Steps:
1. Log on to server with client
2. Kill the client application, with no notification to the server
3. Log on again with the same username used in step 1

Expected Results: Server recognizes dropped connection and recycles resources, including the username. Second login is successful.

User Tests

User tests are performed from the perspective of a user. They are meant to verify expected behavior and appearance of the user-facing components. These tests include such aspects as correct GUI elements being available, UI is updated appropriate for various actions, and the application properly starts up as intended.

1 - Choose Contact to Open Chat

Reason: It's expected behavior, and there must be a way to choose the contact.
Steps:

1. Open the app and log in
2. See the list of contacts
3. Choose a contact

Expected Results: Contact list is downloaded and displayed. When a contact is selected you are able to chat with that user.

2 - Long Chat Message

Reason: Ensure an excessively long chat message is displayed reasonably and in a readable manner to the user.
Steps:

1. Log on to two different clients.
2. Send the United Nation's Declaration of Human Rights from the first client to the second. Ensure the text is delivered in its entirety.
   a. The Declaration can be found here: http://www.un.org/en/documents/udhr/

Expected Results: Text is delivered and readable at both the sender and the receiver. Also, the text input at the sender allows for editing the entire body of text.

3 - Usual methods of input work correctly

Reason: Some expected functionality of native applications doesn't come "for free". For example, Java's Swing framework will not associate <Enter> with a submission of the form (and how could it, it wouldn't know what widget to invoke).

We should test that <Enter> submits an entered message or prompt as users are accustomed to.

Steps:

1. Start a client. Enter a username and press enter.
2. Enter a chat message and submit with enter.

Expected Results: The username and chat message are each submitted to the server with the press of <Enter>, the same as if the buttons had been clicked with the mouse.