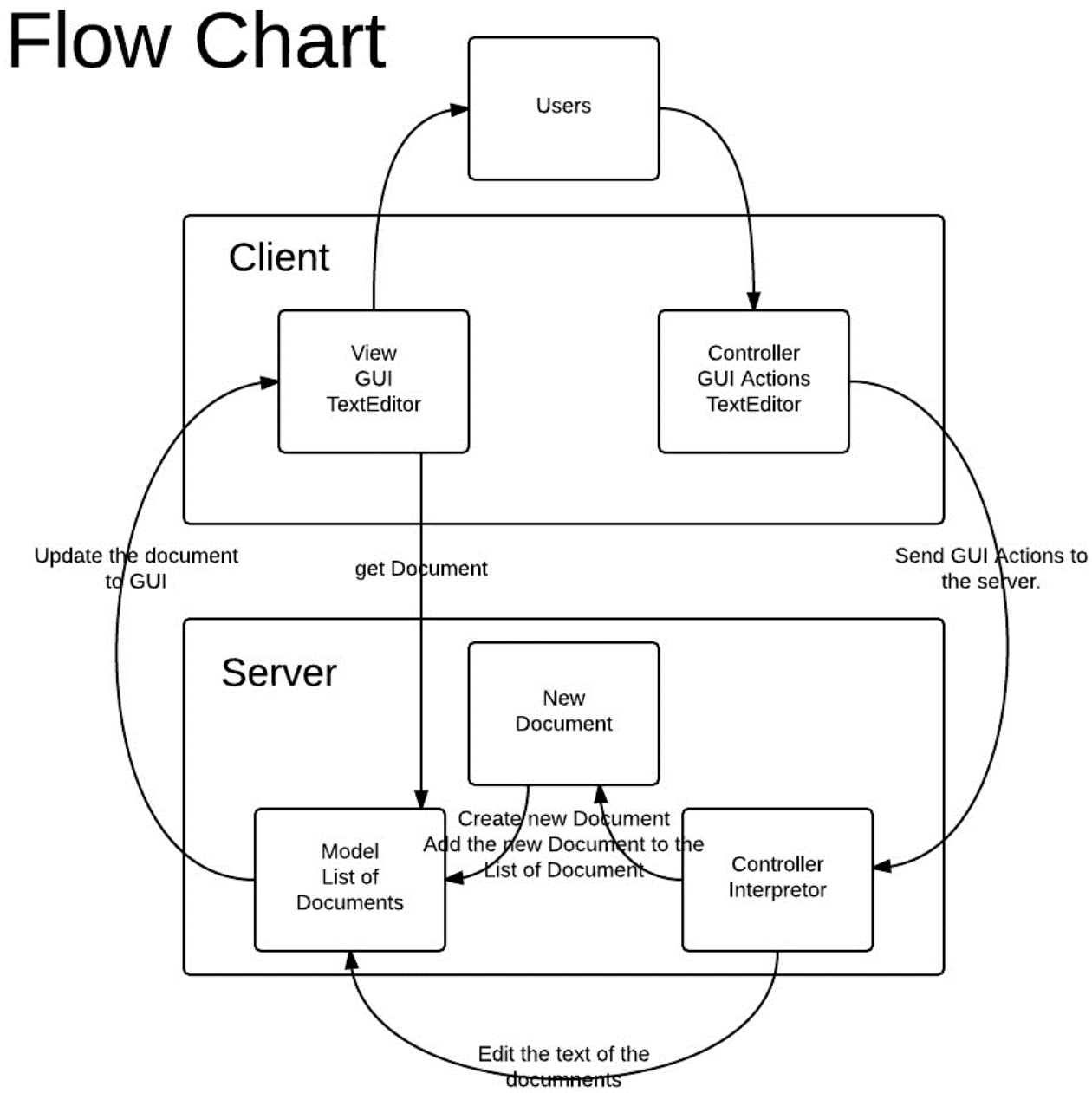
Project 2 Design Decisions

ericemer, salazarm, mpan1218

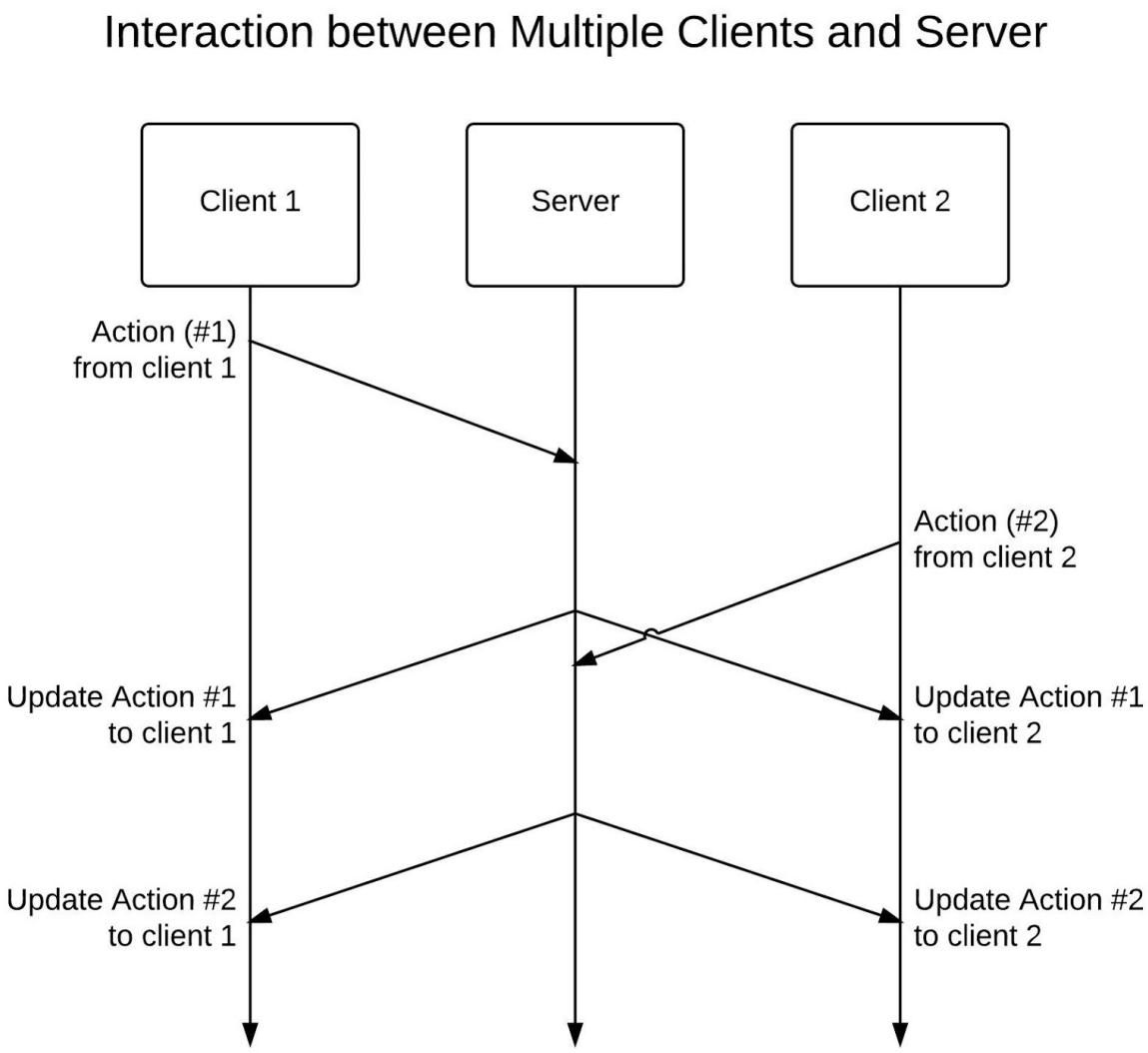
Client => View - Controller Pattern
Server => Model - Controller Pattern
Client - Server => Model - View - Controller Pattern



In this design, there is a central server that stores all the documents for the users and is

connected by the clients. The server will function as a Model - Controller Pattern. The Controller in the serve will edit the design according to the action made by the server on the Client.

The client basically is two different User interfaces which function both as the view and the controller. The first interface is the DocList where the users are allowed to open an existing document or create a new document. The second interface is the TextEditor where the users can edit the file collaboratively.



When a Client sends an action to the Server, the central server will handle the action and make an update to all the Clients are the same time. When multiple actions are received from different Clients, the Server will handle action by action and update to Client whenever it finishes handling an action.

Client:

Fields:

ID - ID of document being edited

Methods:

We will have Swing listeners for different events such as highlight-paste, deleting entire sections, etc. They will appropriately handle how many INSERT and DELETE commands we need to send. There will be a main SEND command which sends text to the server via the output stream.

Design Choice:

1. All Changes are going to be handled by the central server directly. Even the person making the change will not see the update until the server has handled the edit and returns the updated version of the document (We believe the lag will be minimal and unnoticeable enough for this to work).. We withheld the design of storing document in all clients because for the initial design, we want to keep things simple and easy to implement. However, our choice of design might result in latency of updates if lots of edits are made in a short time.

Server:

Fields:

ArrayList<Document> Documents - List of documents on the server.
Map<ID, Queue> queueMap - Maps each document to a queue.

Methods:

String get(ID) - returns Documents.get(ID).toString() and calls Documents.addActiveUser(Socket).
void handleConnection(COMMAND) - Gets the ID from the COMMAND and adds it to the appropriate document's queue.

class Document:

Fields:

Queue<COMMAND> queue - A Queue of Commands to be handled.
ArrayList<char> doc - An ArrayList of characters representing the document.
Map<Socket, PrintWriter> activeUsers - A Map of users editing this document and their output streams.

Methods:

void addActiveUser(Socket) - Opens an output stream and adds the socket and the stream to the activeUsers map.
void insert(INDEX, CHAR) - Inserts CHAR at position INDEX of the doc ArrayList.
void remove(INDEX, CHAR) - Removes CHAR at position INDEX of the doc ArrayList.
void toString() - returns a string representation of the document.
void handleCommands() - This is always active in a background thread. It executes commands in the Queue and calls updateActiveUsers() after every command.
void updateActiveUsers() - Goes through all open sockets in the activeUsers Map and sends the result of document.toString() through the output stream.

GRAMMAR:

COMMAND ::= NEW | INSERT | DELETE | GET

NEW ::= “NEW” NAME
NAME ::= [.]+
DELETE::= ID “DELETE” INDEX
INSERT ::= ID “INSERT” INDEX CHAR
GET ::= “GET” ID
ID ::= [0-9]+
INDEX = [0-9]+
CHAR = [\\s\\S]
DOCUMENT ::= CHAR*

Assumptions:
1. All user edits will be something that can be broken down into inserts and deletes.
2. Server queue will be able to handle requests in a timely manner without debilitating visible lag.
3. A user who makes edits and then rapidly leaves the document will still have his or her requests posthumously handled by the server.
4. A single server/queue will be capable of handling an unlimited number of users.
5. Race condition: Offline editing would cause problems of two people editing the same thing and trying to merge the two documents.
6. Client only expects to receive string representations of data.

Testing Strategy:

Server:
We will test our server by creating a unit test. The unit test will make an instance of a server. Then it will connect multiple clients to the server, and send messages to the server through the clients. It will test that the server sends the proper message to the client. The clients should all see the same message.

Client Methods (Swing Listeners):
Our client consists of a few parts, so we will have to test our client in a few different ways. We will have to test that the methods work. Recall that the purpose of these listeners is to break down edits into commands of INSERT and DELETE, which we can send to the server’s CommandQueue. We will run tests that change the JTextArea in specific ways. These will be INSERTs of characters, DELETEs of characters, highlight-DELETE, and paste. We will then check to see that the updateLog created by the change matches the updateLog we expect the client to send to the server. We will also obviously execute manual testing, to verify that the user’s experience is adequate.

GUI (Document List):
Since this a GUI interface that is going to directly be used by the users, the testing strategy is testing each individual JComponent manually according to the specification. Therefore, the

main thing to test is to make sure that the responses to actions are correct. In this sense, we are essentially testing the GUI as a user. More detail information about the testing strategy and what to look for is listed in the Java file “GUITest” under client.