Final Design:

General Notes

Locking: we decided to implement locking to help minimize conflicts between users. When a user is typing, all the characters in the word that the user is typing are locked. This means that another user cannot insert another character in that word until it is unlocked. An edit is unlocked when the user enters whitespace or moves the cursor. When another user tries to type a letter in the locked edit, that letter is denied and not entered. However, at this point the edit is unlocked so that the second user could enter characters in the edit is he or she would really like to do so. Locking helps minimize conflicts between users but still allows important insertions to be made.

Addressing conflicts between users or because of fast typing: when one user types very quickly or two users type in the same space at the same time, both of these inserts are sent to the server. They are put in the queue based on how quickly they arrive over the network. For example, say that they were typing at index 27. User1 types a and user 2 types b. If user1's message gets there first, it will go on the queue first, followed by user2's message. "a" will be popped off the queue and inserted at index 27. Then, "b" will be popped off the queue and inserted at index 27, shifting "a" to index 28. In this way, we resolve conflicts by letting the network arbitrarily affect the speed of the messages over the network, and we let the inserts be in whichever order, since no one user should have preference over another.
This conflict resolution works the same way for two letters pressed at more or less the same time by one user. The user sends two messages, both with the same insertion index, but ultimately one insert is at that index and the other insert is pushed to the next index. Neither one of the two characters will be lost even if the user thinks that the inserted them at the same location.

Cut, copy, paste: these work but only when you select them from the dropdown menu. The keyboard shortcuts don't work since we didn't add key bindings (there was no time to test out key bindings on all operating systems).

Multiple active documents: since each message to and from the server includes the name of the document that the user sending that message is working on, different users may be working on different documents at once without having any errors. We have tested this and it works. The server is never in a 'single-document state' of sorts; that is, any number of the documents it stores are always accessible, as opposed to the server locking everyone out of all the documents that are not the active one.

Sending messages over the sockets: we implemented sending messages from the server to the GUI by separating each item by pipes. For example, a save message from the server would read clientName|docName|save . Splitting by pipes instead of by spaces means that we preserve spaces whenever we send over the contents of any ServerDocument. However, it means that we can't send pipes in the ServerDocument. Ultimately, we decided that we

would separate the items with unicode characters, but we ran out of time to implement this correctly. We attempted to implement it quickly but something about the unicode characters being sent over sockets was not working. We scratched this attempt and decided to write about it here instead. It is certainly a design limitation but it is one that we understand how to overcome.

Handling ASCII text: As stated above, pipes (which are an ASCII character) are not supported in our program. Otherwise, all ASCII characters work. It handles line breaks appropriately.

Persisting: The server doesn't close when a client closes, and, since the server owns the documents, the documents persist over multiple sessions.

Explanation of classes

We have the following classes: DocGUI, Server, ServerDocument, EditController, and Edit. Using these five classes, we take changes from the user, update the document as appropriate, and flood messages to each user. The classes interact as follows.

Edit

The most basic datatype in the program is an Edit. An edit has a one character value (but in the form of a String not a char) and an owner. The owner is originally the client who added the edit to the document, but the owner is later changed to "document". When the document owns the edit, it means that any client can place another edit near it. When two edits in a row belong to the same client which is not document, no other client can insert an edit in between them. In this way, the edits enforce locking of words that are currently being edited by one user. This simplifies our conflict resolution because it minimizes the number of situations where the cursors will be typing right next to each other.

ServerDocument

The ServerDocument is the data type that stores the contents of a document. It has an ArrayList of type Edit. It has methods for adding and removing edits from its ArrayList, taking into account the way that we have defined locking. It also has toString-like methods for generation document content in a form that is convenient for appending to the end of messages from the server to the DocGUI.

There will only be one global ServerDocument per document, as opposed to there being local copies. There will be multiple ServerDocuments, but they will represent completely different docs. All clients will see and edit this global ServerDocument. There will be no local ServerDocument for each client (but there will be a local GUI), as this would mean that multiple versions of the ServerDocument will have to be merged, and edits may be overridden. Using a global ServerDocument will be less problematic, erase the issue of overridden edits, and allow users to see what is being edited(especially with the way we have defined an edit to be).

DocGUI

Any time that a key is pressed or the cursor is moved in the GUI, the GUI sends a message to the server. The GUI is always listening for various messages from the server since the server does not respond directly in response to the GUI sending it a message. The DocGUI uses JFrames and JDialog. It has the following attributes:

0. Welcome window

Welcome Message

1. Name (6 letters max)
2. IP Address
3. Port

Connects the client to the correct server.

1. New/Open screen

First screen that client sees when connected to the server

Three possible options:

1. New: New Screen pops up
2. Open: Open Screen pops up
3. Close: client disconnects from server

2. Open screen

1. Client chooses from existing inventory of ServerDocument. Server will open the appropriate document in Document Screen
2. Cancel: client goes back to screen 1.

3. New Screen

1. Client types in ServerDocument name, and clicks CREATE. creates a new ServerDocument in the Server under the name, and opens document in Document Screen.
2. cancel: client goes back to screen 1.

4. Document Screen

Options:

1. Close → should return to screen 1, does not disconnect client
2. Save

Another important part of the GUI is handling carets. Here is the caret design:

<u>insert update:</u>

1) if the cursor's current position is before or at the index of the update, the cursor is reset to its current position

2) if the cursor's current position is after the index of the update, the cursor is reset to its current position plus one

<u>remove update:</u>

1) if the cursor's current position is before the begin index of the update, the cursor is reset to its current position
2) if the cursor's current position is inclusively between the begin and end indexes of the update, the cursor is reset to the begin index
3) if the cursor's current position is after the end index, the cursor is reset to its current position minus (end - begin)
new protocol:
clientName docName insert

How does the GUI send messages to Server?
When a message needs to be sent to the server, a new instance of ServerMessage (String message) is created.
This message is sent via a background thread through the output stream that was opened when the GUI was created.

How does the GUI read messages from the Server?
Messages from server are read via an open input stream that was created when the GUI was created. This input stream listens to the socket and parses the messages accordingly.

Listeners Implemented
1. Window Listener:
    Used so that we can close windows without exiting the GUI
2. Key Listener:
    Used so that textfields would have a finer implementation, where buttons will be enabled/disabled depending on user input
3. Action Listener:
    For general button clicking
4. Caret Listener:
    To track the caret at any point

Thread Safety/ Concurrency in DocGUI
Client is connected to the server in his own socket, and thus anything that one client does to the GUI would not influence anything on another client's GUI. The GUI is a separate entity, and is opened by the client.

The only times the GUI can be not thread safe is anytime it communicates with the server. This means that any message sent (ServerMessage) must be synchronized and any message received that isn't partial to only the client's GUI must also be synchronized.

The ServerMessage is synchronized such that messages can only be sent one at a time, and that any threads created to send messages can only be run one at the time. This ensures that multiple messages will not be sent at the exact same time.
Dead locking does not occur as the ServerMessage only cares about printing the message to the output stream. As a result, there are no cycles and thus threads would be waiting on each other in a cyclic fashion.

There is a synchronized block in the parsing of the incoming messages from the server. Update, insert, remove and space entered messages are the only messages that needs to be synchronized as these messages are received when there is the possibility of multiple users and multiple edits.
This ensures that only one of these messages can be carried out at one time and as a result, the updated/insertion/remove should be concurrent. this also makes sure that our caret location is update consistently, which would help us ensure that the right things are being entered at the right place

Server
The server runs similarly to the server from problem set 3. It has a serve() method which has a Thread that waits for new Sockets to connect to it. Then, it creates a new Thread for each client that connects. This client Thread, called Handler (which is simply a class that implements Runnable), calls handleConnection and finally closes the socket.
HandleConnection reads in input from the GUI. It then puts this input on a queue by giving it to the EditController. Then, it pops things off of the queue through the EditController until the queue is not empty. This guarantees that there is as little lag in the queue as possible. It floods the message that the EditController returns after popping something off the queue to all clients. Then, it floods an update message to all users that are currently connected.
The Server owns a Map of documents and document titles. This Map is where the ServerDocuments are all stored. As long as the Server is never closed, these ServerDocs will persist, regardless of which or how many clients are connected. Clients will connect to the server. The client will have a local GUI, a local cursor, but will edit the global ServerDocument. This is to prevent edits being overridden. Clients can disconnect and connect to the server and can still access the edited ServerDocument.

EditController
The EditController deals with the queue. It interfaces between the inputted messages that are popped off of the queue and the ServerDocument; based on the ServerDocument's success with an action, such as an insert or remove, the EditController constructs an appropriate message based on the protocol to send back to the clients via the Server.
The order in which two objects are added to the queue is the order in which the EditController deals with them; their arbitrary order is how we deal with edit conflicts; whichever one happened to get across the network first is the one that takes priority.

Protocol

Only the DocGUI and the Server send messages to each other over sockets. The following is the protocol that the messages follow.

One GUI sends message to server → server deals with message → server sends message to all GUIs

From GUI → Server

Tokens separated by a unicode star character with hex value 2605

Messages	token[0]	token[1]	token[2]	token[3]	token[4]	token[5]
new	clientName	docName	new			
open	clientName	docName	open			
save	clientName	docName	save			
insert	clientName	docName	insert	keyChar	index	
remove	clientName	docName	remove	indexBeg	indexEnd	
enter	clientName	docName	spaceEntered	enter	index	
spacebar	clientName	docName	spaceEntered	space	index	
tab	clientName	docName	spaceEntered	tab	index	
checkNames	clientName	docName	checkNames			
getDocNames	clientName	docName	getDocNames			

From Server → GUI

Tokens separated by a white space

Messages	token[0]	token[1]	token[2]	token[3]	token[4]	token[5]
new success	clientName	docName	new	success		
new fail	clientName	docName	new	fail		
open success	clientName	docName	open	lines	content →	

open fail	clientName	docName	open	fail		
save success	clientName	docName	save			
save fail	clientName	docName	save			
insert success	clientName	docName	insert	beginIndex		
insert fail	clientName	docName	insert	fail		
remove success	clientName	docName	remove	beginIndex	endIndex	
remove fail	clientName	docName	remove	fail		
spaceEntered success	clientName	docName	spaceEntered	beginIndex		
spaceEntered fail	clientName	docName	spaceEntered	fail		
getDocNames	clientName	docName	getDocNames	names →		
checkNames	clientName	docName	checkNames	names →		
update	clientName	docName	update	lines	contents →	

Messages from server to the user:

```
message ::= newSuccess|newFail|openSuccess|openFail|saveSucces|
saveFail|insertSuccess|insertFail|removeSuccess|removeFail|
spaceEnteredSucess|spaceEnteredFail|checkNames|getDocNames|update
newSucess ::= clientName docName “new success”
newFail ::= clientName docName “new fail”
openSuccess ::= clientName docName “open” content
openFail ::= clientName docName “open fail”
saveSuccess ::= clientName docName “save”
saveFail ::= clientName docName “save”
insertSuccess ::= clientName docName “insert” indexBeg
insertFail ::= clientName docName “insert fail”
removeSuccess ::= clientName docName “remove” indexBeg indexEnd
```

```
removeFail ::= clientName docName “remove fail”
spaceEnteredSuccess ::= clientName docName “spaceEntered” indexBeg
spaceEnteredFail ::= clientName docName “spaceEntered fail”
checkNames ::= clientName docName “checkNames” documentNames
getDocNames ::= clientName docName “getDocNames” documentNames
update::= clientName docName “update” lines content
documentNames ::= [docName]*
indexBeg ::= index
indexEnd ::= index
clientName ::= [a-zA-Z]{1-6}
docName ::= [a-zA-Z0-9]+
keyChar ::= [^\\s]
lines ::= [0-9]+
content::= [^\\p{ASCII}*$]?
index ::= [0-9]+
```

Messages from user to the server:

```
message ::= new|open|save|insert|remove|enter|tab|spacebar|checkNames|
getDocNames
new ::= clientName docName “new”
open ::= clientName docName “open”
save ::= clientName docName “save”
insert ::= clientName docName “insert” keyChar index
remove ::= clientName docName “remove” indexBeg indexEnd
enter ::= clientName docName “spaceEntered” “enter” index
spacebar ::= clientName docName “spaceEntered” “space” index
tab ::= clientName docName “spaceEntered” “tab” index
checkNames ::= clientName docName “checkNames”
getDocNames ::= clientName docName “getDocNames”
indexBeg ::= index
indexEnd ::= index
clientName ::= [a-zA-Z]{1-6}
docName ::= [a-zA-Z0-9]+
keyChar ::= [^\\s]
index ::= [0-9]+
”
```