Project 2 Final Design

Last Updated: Wednesday, December 12

By: Hanwen Xu, Tran Nguyen, Youyang Gu

For features, usage, and overall testing strategies, please consult the README in the main directory.

System Architecture:

We will be utilizing a Operational Transformation (OT) technology (reference) to maintain concurrency documents between users. We will adopt a replicated architecture for the storage of shared documents. The replicated documents will be copied at each local site, and then propagated to remote sites. Remote editing operations arriving at the local site are handled by a program and then applied to the local document. In order to make sure all the documents are convergent, we need to code application consistency in the handling of remote operations. The lock-free, nonblocking property of this system makes local response time not sensitive to network latencies. The replicated architecture has the added benefit of parallelizing processing, reducing latency and increasing usability in a group collaboration document. Processing bottlenecks are now unlikely since each client is responsible for its local graphics, as opposed to a central model which must update all of the clients. However, the replicated system will require O(n^2) number of connections between clients. Therefore, we shall use a hybrid form, with a centralized notification server delivering messages to all the clients to process commands. This will result in a O(n) number of connections. The clients are still responsible for processing the individual operations. This is why we will use OT in our 6.005 project.

PROTOCOL:

Basic operations and Grammar:

The implemented grammar is much more complex. Over the network, we are now communicating by transferring Operation class objects, and our state machine representations. This is done by implementing serializable for both our Operation class, and the state machine representation. We also communicate by sending entire strings from the server to a new client, and other data types.

We will use the operation class as serialized objects that contains all information relevant to the operation that was performed (e.g. value, offset, siteID). These operations are passed between our server/client model as the way to ensure concurrent editing.

There are 2 basic operations, insert and delete. Each insert and delete command will contain the index where the operation is taking place, and the ascii characters representation (table) being added. If it is a deletion, the second number is the number of characters to delete. For example, suppose we have a shared text document containing only the string “hello world”. This is being concurrently modified by two users. Suppose we have two concurrent operations

1. O1 = Insert 0 This is
2. O2 = Delete 2 1

After O1, the document becomes “This is hello world”. If the O2 is not parsed correctly, then the document will be “Ths is hello world”. After the O2 is parsed and handled correctly, the final document should read “This is helo world” on both clients and the server.

Message passing:
The clients and servers protocol allows them to pass Operation type messages as specified in the grammar. We serialized the Operation object class for transmission. In addition, the server also passes other types of objects to the client, such as the updated list of users and documents.

Datatype

We use the built-in swing JTextField to store the document string in the GUI, as well as other JFrame components to complement our editor GUI. All changes are made locally by the client or server. Each remote operation spawns a new thread in the client or server before being merged into the document using the OT algorithm. The only time concurrency issues may occur is between when an operation is created (i.e. a user typed a character) and when the change gets reflected in the JTextField. To solve that, we simply lock the TextField to prevent any malinformed transformation from occurring during this tiny fraction of a second.

Edit

An edit is viewed as an insert or delete operation generated by the client, which contains all relevant information about the state of the edit (order, key, position, etc). More information about the operation class is discussed in OT section.

The Server/Client Model

We use a server/client model to abstract the design of this collaborative editor. We have both a text-based protocol (Host.java) and a GUI protocol (LoginPage.java) to make all the connections and communication between the clients and server. Here, the server is defined as the central entity that can receive and send message to all the clients. The client will represent a “user” that will be editing a single collaborative document using a GUI. The client will use a standard Berkeley socket, TCP connection to communicate with the server, and does not know information about the other clients.

From an abstract perspective, the clients keep local copies of the document that is being edited. Changes to the local copy on the client creates an operation (with all the appropriate states) that is sent to the connected server. The server is notified and this operation is sent to all other clients in its original form. Each client (as well as the server) will then use the operational transformation algorithm to update their own local copies. The operational transformation algorithm guarantees that all clients using it will converge to the same document, which is what we are seeing in our implementation.

The server allocates one thread for each connection, and keeps track of the following info for each client:

- the socket for the connection
- the input and output streams
- the siteID (a unique ID to the client that starts at 1 and increments by 1) (the default siteID of the server is 0)
- the document the client is editing (a HashMap of name to ServerGui model)

Each time a new client connects, the server will send the initial string and ContextVector of the document that the client is editing. The client then creates a new instance of the ClientGui model with this information. Future operations generated by this client will have the document ID embedded in Operation.key.

Concurrent edits on the same document are handled via the OT algorithm. Editing multiple documents are done according to the following:

- the server keeps track of a HashMap of document names to ServerGui
- the client relays to the server what document it wants to edit
- the server sends to the client a copy of the ContextVector for that document. That way, future operations created on the client gets relayed properly to the correct document
- when the server receives an operation from a client, it checks for the Key of the Operation, which specifies which document the operation is for. It updates that particular document using the OT algorithm.
- the server sends the operation to all clients, who then use the same key to determine whether or not the operation applies to their own document.

Note: The server can switch its view to see different documents, although we made a design decision to make all documents only editable from the client side.

<u>Identifying the clients:</u>

We will set a maximum limit of concurrent users to a document, MAX_USERS. We keep an ArrayList of all users, as well as an ArrayList of all the corresponding sockets. Each new client is assigned an ID that starts from 1 and increases by 1 - it is simply the index of the ArrayList of users. This is the internal ID used by the server - the external ID can be set by the user at the creation of the client. When a client disconnects, we flag the client as “closed” in the ArrayList of sockets, so that we don’t send operations to that client from now on. We use the ArrayList of usernames to update the display on the right side that shows all connected clients. A similar implementation is used to display the list of documents.

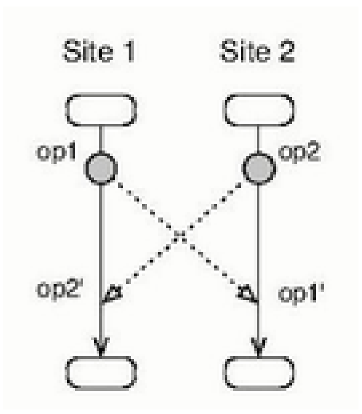
When a client disconnects, the server appropriately closes all sockets and streams while updating the list of active users. All other clients are not affected. However, a server cannot close - closing the server will disconnect all clients because all socket connections has been broken.

<u>Rejected Identification Model:</u>

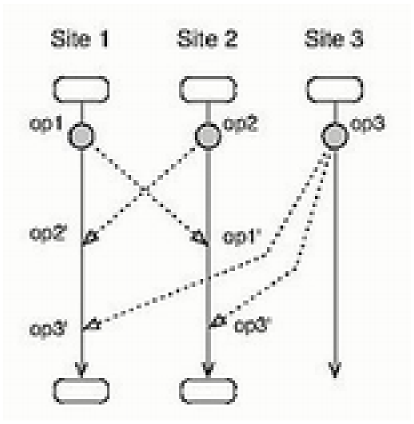
We will set a maximum limit of concurrent users to a document, MAX_USERS. We will keep an updated list of active and disconnected users, whose IDs range from 1 to MAX_USERS. Each time a new client connects, we will assign it an ID from the list of disconnected users and move it to the list of active users. The entire identifier will include: *timestamp-host IP-client ID.* When it disconnects, we put the ID back into the list of disconnected users. This operation will be thread-safe, ensuring that two users will not have the same ID.

Operation Transformation Functions:
As discussed in the reference, we will need to include two categories of transformation functions. Inclusion Transformations IT(Oa, Ob) transforms Oa against Ob such that the impact of Ob is included. Exclusion Transformations ET(Oa, Ob) transforms Oa against Ob such that Ob is excluded. These functions must be written to satisfy the following two convergence properties:

CP1/TP1: For two concurrent operations, the final result must be equivalent on all clients.



http://en.wikipedia.org/wiki/File:OTtp1.jpg
CP2/TP2: For three concurrent operations, the transformed operations should not be dependent on the order executed. All three copies must converge to the same document.



http://en.wikipedia.org/wiki/File:OTtp2.jpg

OT Control (Integration) Algorithm

Introduction and background
We will be utilizing an algorithm described in the graduate thesis of Ali Asghar Zafer of Virginia Polytechnic Institute (reference) in section 4.7.2.

We are striving for the following goals: we want to reduce latency, so that local edits are extremely fast, and indistinguishable from single person editor. Also, the latency must be governed by the network latency. In addition, multiple users must be allowed to simultaneously and freely edit any part of the document with no locking at any time. And, with the hybrid-replicated architecture, we want the users to be allowed to work from remote locations, connected by network with some unknown latency.

In a high level overview of the algorithm, whenever a user joins a session, the client establishes a connection with the server. The server sends a copy of the document to the client. The client will then display the document. The user will then make edits, which are applied directly to the local copy, and they are buffered and sent to the server to distribute to other clients. The server receives updates from all the clients, processes, updates its own copy, and sends the updates to the clients so that they can update their own copy. We now must outline the algorithm that makes sures all contentions and conflicts are resolved and the copies at clients and servers are convergent.

Divergence might occur due to the random latency time between operation transmission and receiving. This might lead to a state of the document being different at different clients if we do not have transformation algorithm.

The first thing we need to preserve is causality. If O2 is executed locally after execution of O1, then the execution order must be preserved at all clients and the server. Let us define causal ordering relation on operations in terms of their location of generation and execution:
Causal ordering relation “->”: With two operations Oa and Ob, generated by clients i and j, Oa -> Ob if and only if
1. i=j, and the generation of Oa happened before the generation of Ob
2. i is not equal to j, and the execution of Oa at site j happened before the generation of Ob.
3. there exists an operation Ox, such that Oa -> Ox and Ox -> Ob (transitive)

Dependent operations: Given any two operations Oa and Ob, Ob is said to be dependent on Oa if and only if Oa -> Ob

Independent operations: Given any two operations Oa and Ob, they are independent if there exists no causal ordering relation between the two. This can be expressed as Oa || Ob.

We can solve basic inconsistency by enforcing a total ordering on messages sent to the clients. This can be done using the central server which can time-stamp each message before sending to clients. However, this subjects the program to round trip latency before a local change can be applied to the local document. We will also encounter intention violation if we only enforce total ordering.

Intention violation is another property that we will try to enforce. Because certain operations are independent of each other, and are generated without knowledge of other operations, the actual effect of an operation at the execution time might be different from the intended operation. We shall illustrate this with an example.

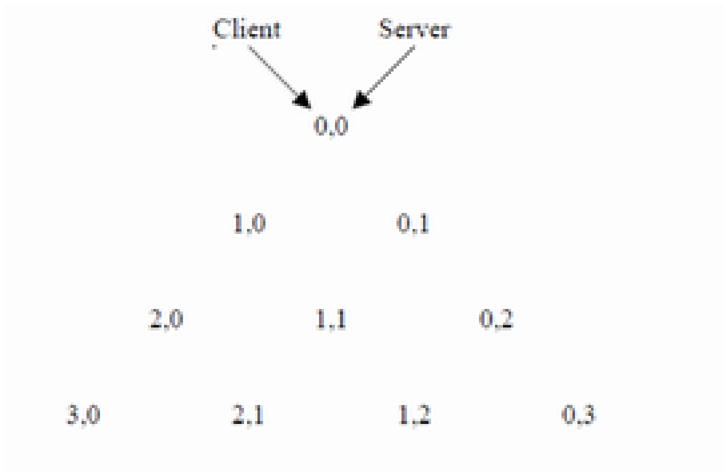
Let us suppose there are two clients, C1 and C2. The document at each client is currently “Hello World”. Let C1 give operation O1 which is “Insert 6 ‘Brave New’ “, or insert “Brave New” at offset 6. Let C2 give operation O2 which is “Delete 7 5”, or delete 5 characters, offset 7. The intention was to produce “Hello Brave New”, but the state at C1 will be “Hello New World”, because it executes O2 second. This violates the intention of O2 since it wanted to delete “World”. Now we have three properties we need to solve for: divergence, causality, and intention.

Consistency Model:
This consistency model is to enforce properties desired of a collaborative editing system. The major properties we desire are Causality, Convergence, and Intention. Causality ensures execution order of operations are in a cause-effect relation. Convergence requires that all documents are the same at all local sites and the central server at the stable state where all operations have been executed. Intention is the execution of what was desired. We shall refer to this model as the CCI model.

Transformation of a message
Transformation will basically be changing the offset of the message at which it is applied to the document, so the new offset is consistent with the execution of all the other operations, fulfilling our CCI model. We can use the central server to impose global ordering to take care of convergence and causality. However, we also need to transform independent operations with respect to each other to handle Intention. This will be handled by modifying the offset with respect to each other. This will only be necessary if they originate from the same state of the document.

Algorithm Description:
We will now discuss the synchronization algorithm in detail. We shall first discuss the state machines and data structures.

State Machines:
The State space contains a vector of numbers, where each index represents the client number, and the value at that index represents the number of operations that client has processed. The states of each client should try to converge where the number of operations processed is equal on all clients. Both the server and the clients have states in the this format. However, on the server side, there will be an individual state for each client-server pair. The server will then have equal number of states to client connections.



Reference: [1]

As an example, a client with state (2, 2) means that it has generated and processed 2 operations on its own, and received and processed 2 operations from the server. A server with the state (1, 3) means that it has received and processed 1 message from the client, and it has generated and processed 3 messages on its own. Note that the server will have a state for each client connection. If the messages are received in the correct order, then the state of the client, and the respective state on the server should follow the same movement in the state space.

User Interface design:

Our UI design currently support 4 different screens. The first screen (1) is for the user to select whether they then to be a server or a client. There has to be a server opening the connection already, so if the user clicks on the server, it will direct the user to second screen (2), it will automatically record the localhost as the server and port is 4444. When the client connects, he can specify name, IP and Port. Default value is Anonymous, localhost , and 4444 respectively.

If the client finish screen 2, it will have to option to select the document to edit or create a new one on screen (3). When the client makes the selection, it will be direct to screen (4) and the client and server will both share this main editing screen. The server can't edit anything on this screen, but it can switch between screens to view different documents concurrently editing right now. When the new user or new document created, it will be shown on the right pane.

If the client closes the window, its connection will be terminate leaving others unaffected. Current supported features for concurrent editing is copy,cut, paste, redo, and undo. BOLD, ITALIC and UNDERLINE are not supported concurrently among users. During this procedure if the user doesn't specify things correctly, there will an errorDialog shown, but it doesn't nothing to the progress.

The user can have to option to start from command line or from GUI. This is implemented using MVC. There are instructions to guide the user through every step. For more info, refer to the documentation in ServerGui.java and ClientGui.java, as well as our updated sketch on the last page.

Abstract Designs we rejected:

Our original design focused around using some form of String Buffer datatype to represent the document. The details of the original design are below:

We shall name our buffer ConcurrentBuffer
Properties: Threadsafe, mutable buffer which allows insertion and deletion of characters to the string buffer.

```
Functions:   Insert(int pos, String s):  inserts the string into this position of the buffer
             Delete(int pos, int len):  deletes string of length len from pos.
             length():  returns length
             toString():  returns string of the buffer
```

As discussed in class, we shall use a gap buffer datatype, which is optimized for document editing.

```
GapBuffer implements ConcurrentBuffer
private char[] a;
private int gapStart;
private int gapLength;
// Rep invariant:
// a != null
// 0 <= gapStart <= a.length
// 0 <= gapLength <= a.length - gapStart
```

We need the invariant that makes sure the gap does not exceed the bounds of the character array.

However, after realizing that all of the JComponent in Swing has an inherent model, we decide to use this inherent model for most of our design such the DefaultStyledDocument for JTextPane, DefaultListModel for JList. All of these are fully supported by swing and are thread-safe, so it's extremely convenient to utilize this.

The thread-safety argument holds because each client only accesses its own copy of the buffer, so multiple threads will not be accessing this datatype at the same time. All changes will be done by the OT algorithm in the client itself, which is sequential.

- We thought about having a central entity keep track of the document and sending it to the clients rather than each client keeping a local copy. There is a problem with this design: When the server sends the updated document back to the client, how does the client integrate this document to its own local copy if the client has made additional changes in between the time it

took for the server to process the edit. If we write an algorithm to integrate it, what would be the purpose of having a central entity in the first place?
- At this moment we want to be able to first support inserts and deletes of one character. We believe that simplicity and basic functionality should take precedence over complexity. Once we have a working model of a collaborative editor that does inserts and deletes, we will expand our operations to include more complex functionalities, such as copy & pasting.

Old edit

Each client and server pair will contain a queue of all the changes that need to be made. Any single, simultaneous change made by a client is added to the queue according to the timestamp this request was sent. The client will go through the queue to make all the changes. From the point of the view of the client, an edit is any addition to its own queue. For example, each time a new character is typed, the change is added to the queue. This change is also added to the queue of the server, which adds it to the queue of all other clients. Hence, an edit made on one client will become edits in all the other clients. Insertion, deletion, copying and pasting are all examples of an edit.

Some rejected states:

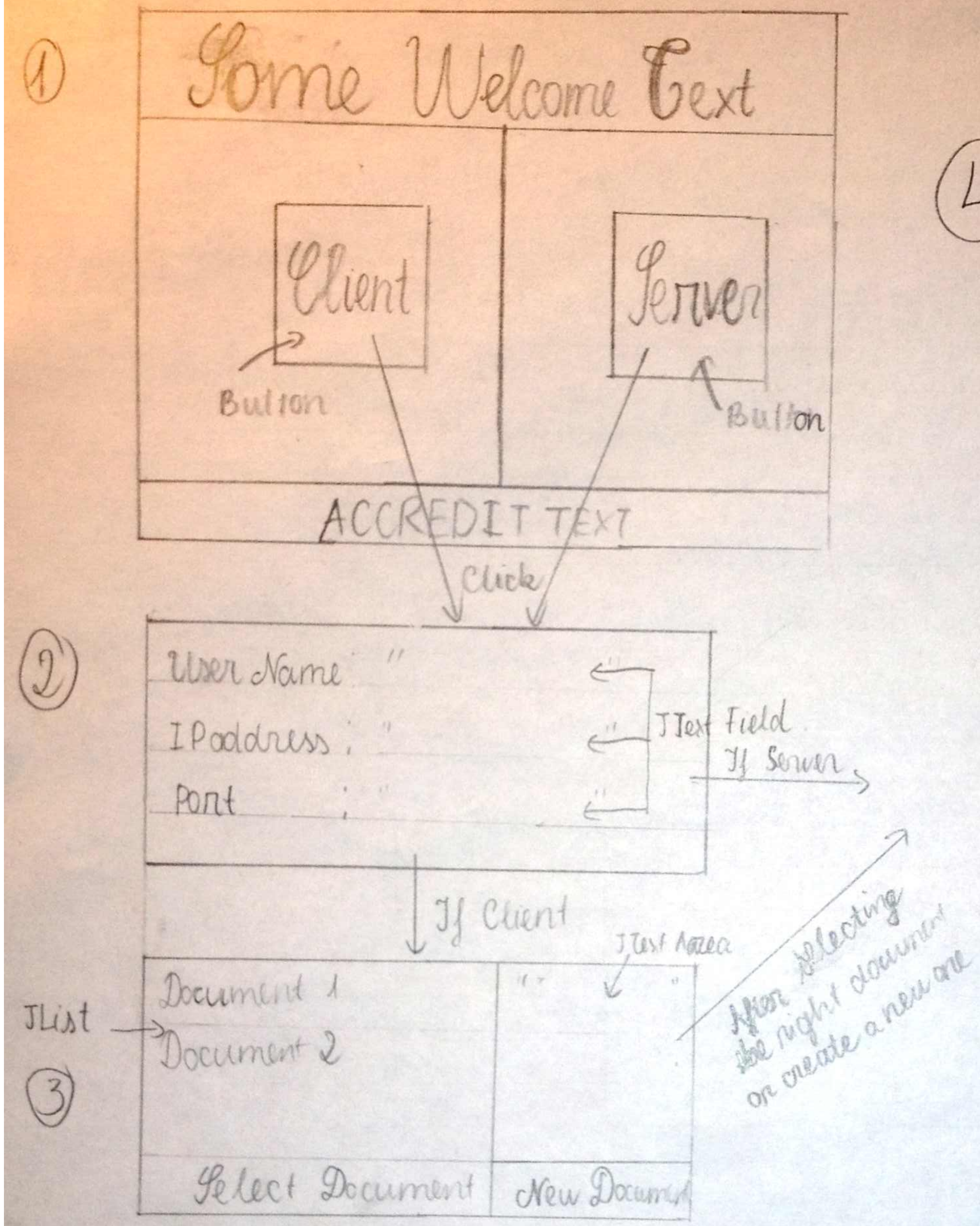
We initially thought of having just two states: editing and observation for each client. While it is editing, it will transmit the operations to the server for the server to distribute to other clients. While it is observing, it will just passively receive operations from the server. We realized that this state will not function properly because it will not satisfy the CCI model we discussed previously. This is because concurrent edits will cause the documents to not converge at the stable state of observing. Also, we did not have state for the server.

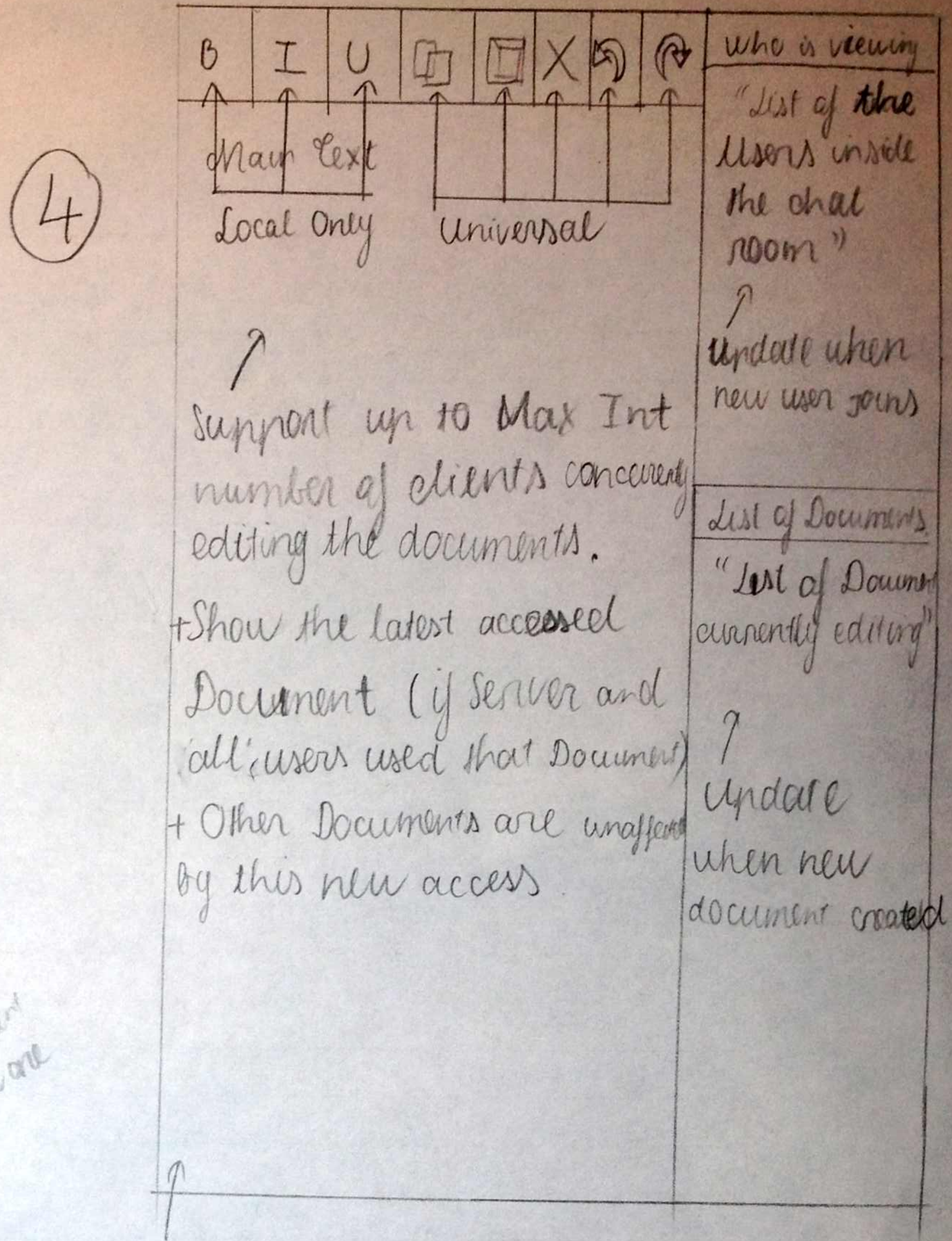
Client side algorithm:

The purpose of this algorithm is to maintain the CCI model on each client side. This is done by taking into account the state space of the server and the current client, and transforming messages in the buffer. The state is updated for the client. This is done utilizing a recursive algorithm that keeps track of all operations and the operation's state by using a HistoryBuffer. This HistoryBuffer allows each client to keep track of what operations were done, and in what state space. Therefore, when given a new operation remotely, each client can use the HistoryBuffer to call back the operations that were done concurrently, and apply the necessary changes so that the document is concurrent.

Server side algorithm:

Similarly for the server, we need to transform messages in the buffer to maintain the CCI model. We do this by comparing server state space with the client state space to determine how we modify the operation. The operation is then executed, and the server's state space is updated.




4
B
I
U
X
Main Text
Local Only
Universal
who is viewing
"List of the Users inside the chat room"
Update when new user joins
List of Documents
"List of Documents currently editing"
Update when new document created

Support up to Max Int number of clients concurrently editing the documents.

+Show the latest ~~accessed~~ accessed Document (if Server and 'all' users used that Document)

+ Other Documents are unaffected by this new access.

If the user select the old document, this screen will show the lastest documents access by the latest user, leaving other document untouch. If the user creates the new one, the screen of server and that client will be new