Project 2 Design

Hanwen Xu, Tran Nguyen, Youyang Gu

System Architecture:
We will be utilizing a Operational Transformation (OT) technology (reference). 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 grammar will be
Operation ::== Insert | Delete
Insert ::== "Insert" DIGIT+ CHAR+
Delete ::== "Delete" DIGIT+ DIGIT+

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 only Operation type messages as specified in the grammar.

Overview

We will have a JTextField in Swing to display our document, as well as other JFrame components to complement our editor GUI. We will follow a server/client model to keep track of all the edits. The document will be stored on a central server, with each client making its own local copy as changes are reflected in the central document.

An 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.

The Server/Client Model

We will use a server/client model to abstract the design of this collaborative editor. We will use a text-based protocol, namely **telnet**, to make all the connections and communication between the clients and server. Here, the server is defined as the central entity that can receive from and send message to all the clients. The client will represent a "user" that will be running an instance of the collaborative document using a GUI. The client will use a 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. All the local copies are also sent to the server, which *does not integrate them*. Instead, each time a local copy from a client is changed (i.e. an edit is made), the server is notified and this new copy (along with the states) is sent to all other clients. Each client will then use the operational transformation algorithm to update their own local copies. If no edit is made during this time by the clients, then all the clients should have the same copy of the document. The server will act as a special client in that it will also keep a local copy of the document. This copy will not be edited by the server, and will be updated using the same algorithm as the clients. One thread will be allocated for each client.

The server will act as a special client that never disconnects, ensuring that its local copy of the document will be saved even after all the "real" clients have been disconnected. Each time a new client connects, the server will create a new copy dedicated to that client, and send this copy (with the appropriate states) to the client to store as a local copy.

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.

Datatype

For the document string in the GUI, we will use some form of String Buffer datatype to represent the document. 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.

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.

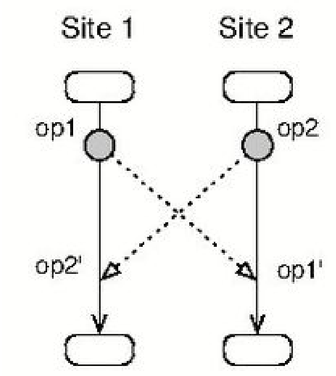
Abstract Designs we rejected:

- 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.

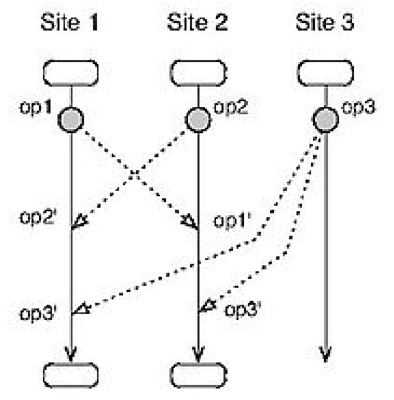
Operation Transformation Functions:

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 client copies.



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.

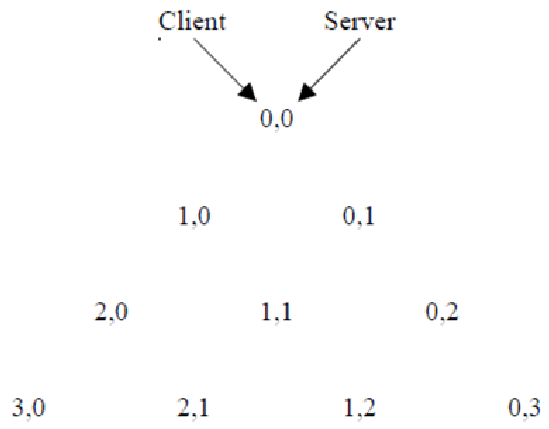
Let us suppose there are two clients, C1 and C2. The document at each client is currently “Hello World”. Let C1 give operation O1 which is “Insert 6 ‘Brave New’ “, or insert “Brave New” at offset 6. Let C2 give operation O2 which is “Delete 7 5”, or delete 5 characters, offset 7. The intention was to 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 two numbers, first number represents the number of operations done on the client, and the second state represents the number of operations processed on the server. The states of each client should try to converge where the number of operations processed is equal on client and server. 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.

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.

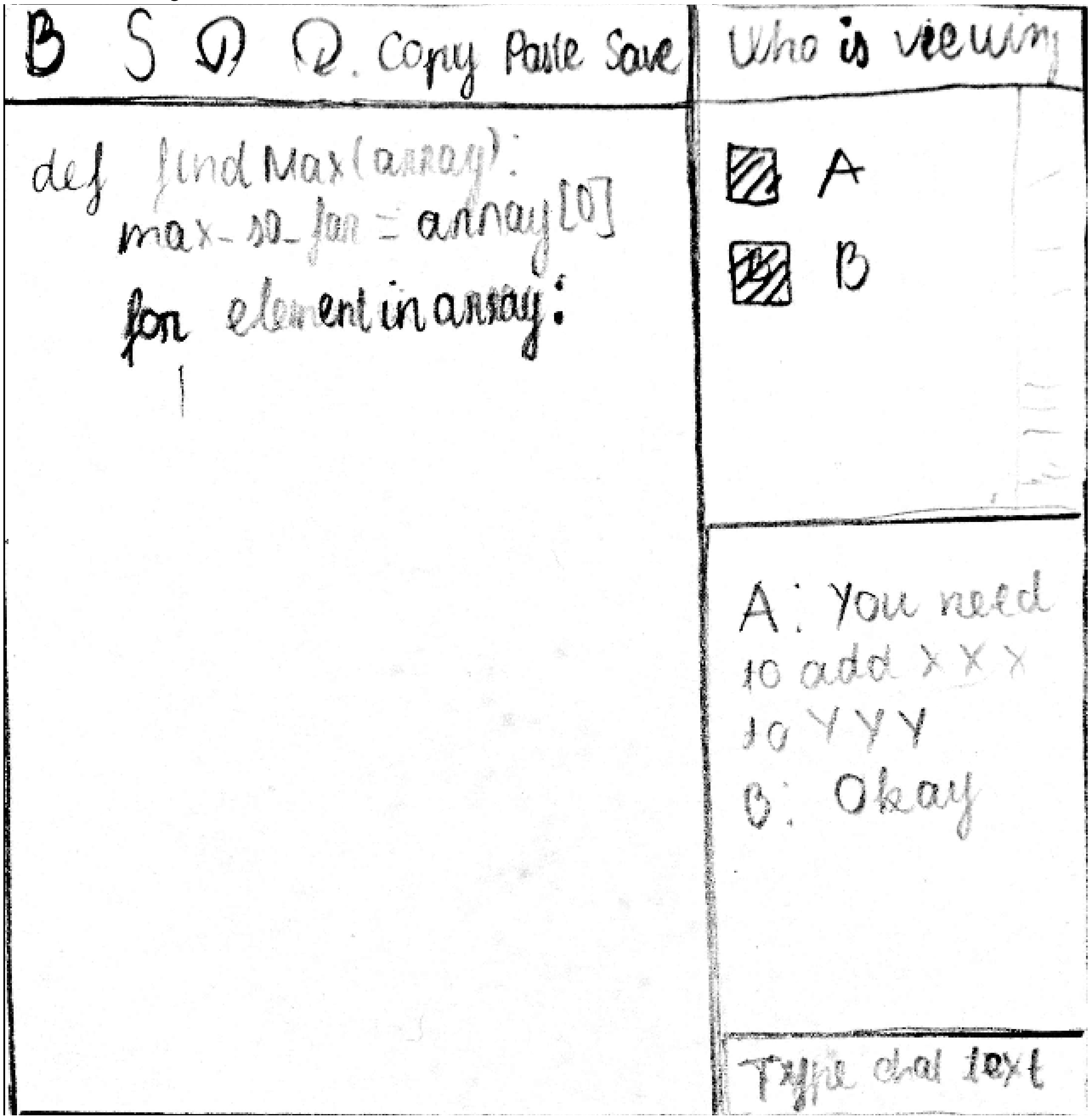
Client side algorithm:
The purpose of this algorithm is to maintain the CCI model on each client side. This is done by taking into account the state space of the server and the current client, and transforming messages in the buffer. The state is updated for the client.

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.

Telepointers:

We also would like to implement telecarets to show the cursors of the other concurrent users. We would also like to see when another user has selected a section of the text. It will be a feature that can be added after the basic document has been implemented.

Sketch UI Design:



```
def findMax(array):
    max_so_far = array[0]
    for element in array:
```

A: You need to add XXX to YYY

B: Okay

NOTES:
We will have a central, remote server that also acts as a special client by keeping track of an updated copy of the document. For current plans, we think we will have a separate machine be the server.