CPSC 416 Project Proposal

Jordan Coblin, Chris Li, Sean Tyler, John Wiebe

Introduction

The ability to collaboratively edit a piece of text in real time with the preservation of intention from each collaborator presents an interesting challenge in the realm of distributed systems. One main challenge lies in the fact that the speed at which edits are propagated to a shared a document is limited by network latency. Due to the user expectation that edits happen in 'real time', changes to a local copy of a document must be synchronized to the changes of another remote copy of the same document which presents the main problem of figuring out exactly how to apply updates in such a way that all collaborators will be editing a local copy of a document without having any conflicts that may produce divergent versions of said document.

Our goal is to design a simple, collaborative text editing web application. Users will be able to edit a shared document and see changes as they occur in real time. The underlying infrastructure will be written in Go and will serve as the main logic that will handle the actual editing of the text document while Javascript will be used to facilitate the front-facing user interaction with the document. The primary problem that we will attempt to address in this project is that of data consistency, with a smaller focus on data unavailability.

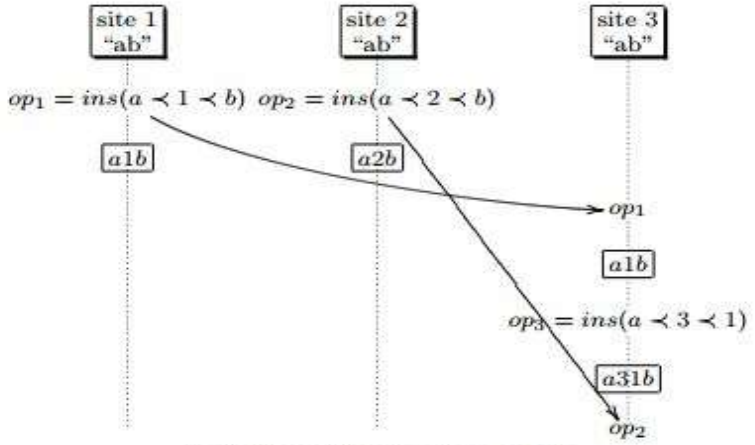
Background

In order to manage collaborative editing on distributed systems, the WOOT (WithOut Operational Transformation) framework was developed by Oster et al (2006). Meant to be a simpler alternative to the popular Operational Transform approach, WOOT ensures consistency between replicas using three properties:

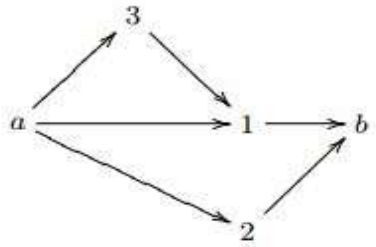
1. Convergence: Replicas of a document will agree on its state once the same set of operations has been performed on each one.
2. Intention Preservation: Performing an operation at a remote replica has the effect that the user intended and any concurrent operations do not interfere.
3. Precondition Preservation: An operation can be performed only if its preconditions are met.

To ensure the convergence of replicas, WOOT provides a solution to the problem of producing a total ordering of characters in a document given only a partial ordering. To illustrate this problem, consider the case when multiple sites (replicas) try to insert different characters into the same string in a document. Because the order in which the inserts will be received at each site differs, a partial ordering, and not a total one, is generated. For example, in figure 2, site 1 attempts to insert "1" between "a" and "b", site 2 attempts to insert "2" between "a" and "b" and site 3 attempts to insert "3" between "a" and "1". The partial ordering generated from these operations results in the following possible orderings (linear extensions): "a312b", "a321b", and "a231b". Obviously, we need some consistent way of determining a total ordering from the generated partial ordering otherwise the states will diverge. WOOT solves this problem through the use of a monotonic linear extension function, where a total order is decided upon

using unique character identifiers. Additionally, assuming that id(“1”) < id(“2”) < id(“3”), when site 3 receives the call insert(a≼2≼b), “3” and “1” are already in between “a” and “b", hence two possible orderings exist: “a231b” and “a312b”. WOOT resolves this by making the placement of characters independent of the order that they are received.



(a) Generating partial orders



(b) Ordering relationships

Figure 2: Partial orderings

In order to ensure convergence of replicas irrespective of the order of operations, the pairs of operations (ins,del), (del,del), and (ins, ins) must commute.To accomplish this, WOOT outlines multiple algorithms. These algorithms fit together in three stages. The generation stage is where an operation is first integrated at its local site (replica), and broadcast to all other sites to be integrated. The reception stage is where a site verifies the preconditions of an operation (for example, the existence of b if del(b) is called). Lastly, the integration stage is where each of the sites integrate the operations, and where the main implementation challenges lie (due to the previously mentioned challenge of partial ordering).

Data structures required, as well as our general approach to implementing a collaborative text editor based on the WOOT framework, will be explained in the next section.

Approach

The system we are designing has its primary challenge in efficiently maintaining consistency between replicas of a text document. To tackle this challenge, we plan to base our design on the WOOT framework as described in the previous section. We will now outline the essential components of our system.

Data Structures

Each character in a document will be represented by a five-tuple that we will call a W-character. W-characters contain the following elements:

- ID: A tuple containing a clientID and the client's logical clock (which is incremented after each operation)
- Visible: A boolean that determines whether a character should be displayed in the document.
- Content: The character itself.
- prevChar: Points to the W-character that comes before.
- nextChar: Points to the W-character that comes after.

A few things to note: Since the visible element indicates whether to display a character, W-characters don't actually need to be deleted. Also, the prevChar and nextChar elements suggest a linked list structure for strings of W-characters, which leads into our next data structure.

W-strings are linked lists that maintain an ordering of W-characters and are essentially what will be used to hold the state of a document. They simply contain a sequence of W-characters, with special characters to mark the beginning and end of the string. The two main operations to be performed on a W-string are:

- insert(a≼x≼b): insert x between a and b.
- delete(x): set x to invisible.

The precondition preservation property ensures that W-characters with IDs "a" and "b" exist, and that they are properly ordered for insert, and that x exists for delete.Also, since each ID is characterized by a unique clientID and a logical clock, these IDs are themselves unique. Hence, there is no ambiguity when calling an operation on a given W-character (e.g. deleting 'c' from the string 'acbc' will have no ambiguity). Other functions will need to be defined for W-strings for a full implementation of this project.

System Design

The basic design of our system will consist of assigning each user a local replica of a shared text document. These replicas will be required to converge given any sets of operations from the various users, and the document will be displayed to users via a simple web application. Each replica in the system will be assigned a unique ID, which will act as the first element in the ID tuple of any given W-character. Each replica also has a logical clock, a string of W-characters, and a list of operations pending to be integrated.

As mentioned in the background section, when a user makes an edit to the text document, this edit will be processed in three stages:

Stage 1: Generation
func generateInsert(pos, x) {} // *integrate the insert locally and broadcast the insert to other replicas*
func generateDelete(pos) {} // *integrate the delete locally (set visibility to false) and broadcast to other replicas*

Stage 2: Reception
func receiveOperation(op) {} // *add the operation to replicas pool of operations*
func isAllowed(op) {} // *check that preconditions are satisfied upon receiving an operation to integrate from another replica*

Stage 3: Integration
func integrateInsert(x, a, b) {} // *integrate the remote insert locally using strategies from the WOOT framework*
func integrateDelete(x) {} // *integrate the remote delete locally*

The general idea is that when a user types a character x into the document, generateInsert(pos, x) will create a W-character for x and then integrate the insert at the current cursor position in the document. Then, the insert will be broadcast to all other replicas, and upon receival, these other replicas will add the insert to their list of pending operations. Each replica will have a loop which constantly checks the list of pending operations and integrates them. This integration poses the problem of character ordering as mentioned in the background section. Using the WOOT framework, a total ordering of W-characters can be determined at each site given a partial ordering using the unique IDs of the W-characters.

We will be using a gossip protocol to broadcast operations between replicas. The gossip protocol works in the following way: Each replica will periodically send data to another randomly selected replica (which will then send its data back in response). These replicas will then merge the received data with their own to reflect any changes. Since each replica is periodically doing this, there is a doubling effect that allows the information to spread quickly and robustly throughout a distributed system. We will be using UDP to send gossip messages, which will have a TTL to limit propagation.

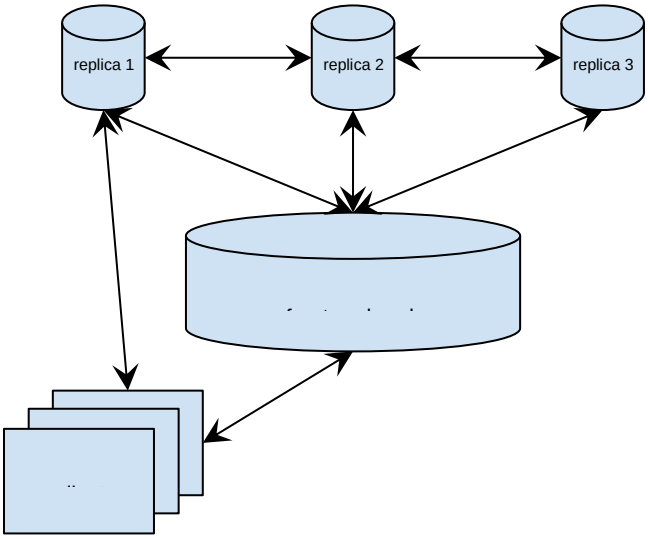
System Interaction

There will be several different interactions that will occur in the system between the client, the front-end node used for serving the client, and the replica nodes used for storing and delegating data from client to client. The front-end node will be continuously listening for new replica nodes, of which are manually initialized and connected to the front-end node. Note that these replicas are never re-replicated and if they become unavailable at any time, they will remain so as the job of the front-end node will only narrow down to general bookkeeping and routing.

Clients will connect to the front-end node and will either choose to create or edit a document, in turn causing the front-end node to effectively route the client to an existing replica node. Clients will then

communicate with this node directly via a WebSocket protocol defined on both sides (clients will use the natively provided Javascript WebSocket API, while nodes will be communicating to the clients using the Gorilla Go library). Actions to a document (creation or edits) are applied instantly to a local copy of the document and are then sent from the client to a replica node, in turn causing it to randomly select another replica node from a list of active replica nodes (provided by the front-end node) to send this action to.

At the same time, the sending node will broadcast this action to all the clients that are connected it and are editing the same document. Clients receive this action and will apply this change into its own local copy, completing the update loop. Replica nodes that receive a broadcasted change from another node will apply the change to its stored version of the document (using the same algorithms described previously) and, again, propagate these changes to connected clients. Notice that there could be a client connected to a different node that could be editing the same document, but because of the gossip protocol implemented between the replica nodes coupled with the consistent structure of the data type forming the document, we are guaranteed an eventual consistency amongst the local copies of each client. Changes will *eventually* propagate to the client, while maintaining user expectations as the document is continuously edited and the document itself will *eventually* reach the same state between the editing clients.



Testing

A simple terminal client can be used in place of the web application to initially test the interaction between the front-end node and the replicas. As the project moves forward, testing becomes more trivial as we just need to make sure changes are received properly and the perceived document is the same across all collaborating clients.

Timeline

March 7th	- Basic design of data structures and operations - Basic web app started - start by allowing editing for single user
March 14th	- Gossip protocol for broadcasting operations - Implement each of the stages for an operation
March 21st	- Begin integration with web app - allow multiple users to edit the same document
March 28th	- Replica failure
April 4th	- Testing - Begin final report
April 11th	- Project deadline

SWOT

Strengths:

- Majority of the group has strong experience with software engineering and design principles as they all have done at least 3 terms of co-op work placements
- Experience in Javascript applications

Weaknesses:

- Limited understanding of order theory and related algorithms
- New to Go programming language and Distributed Systems
- Spreading time among other coursework
- Difference in schedules and location of each team member

Opportunities:

- There are a variety of resources that detail how to specifically implement collaborative edits

Threats:

- Integrating with a web app UI could pose some technical challenges due to the complexity in communicating between two fundamentally different languages/frameworks (Go vs. Javascript)

References

http://www.ds.ewi.tudelft.nl/~victor/polo.pdf

https://hal.inria.fr/inria-00108523/document

http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=1749FF9B27D628A460A349287D552232?doi=10.1.1.160.2604&rep=rep1&type=pdf

https://github.com/ritzyed/ritzy/blob/master/docs/DESIGN.adoc