CPSC 416 2015W2

WOOT Collaborative Text Editing

Jordan Coblin -f3j8
Chris Li - y7e8
Sean Tyler - c1e8
John Wiebe - n4m8

Abstract

This report details a system providing collaborative editing of a document amongst a network of users as a distributed system. The system developed is written in Go and AngularJS and utilizes a peer-to-peer network not unlike BitTorrent. It implements the WOOT (WithOut Operational Transformation) framework to ensure consistency of intention (every client will agree on the ordering of characters entered as they are entered and spread amongst each other) amongst the various peers' local copy of the document. The report also evaluates the system that was built, details how the distributed events have been logged using Shiviz, and discusses its limitations.

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 other remote copies. This presents the 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 that preserves intention of character ordering amongst all copies.

This project aims to develop a system that will solve this problem by using the WOOT (WithOut Operational Transformation) framework as described by Oster et al. in their paper, “Data Consistency for P2P Collaborative Editing”. This report will give an overview firstly of what the WOOT framework is, and then go into the design, implementation and evaluation of the system we have built. Lastly, we will discuss the limitations we found with our design and take a look back over the efficacy of this system in solving the problem put forth.

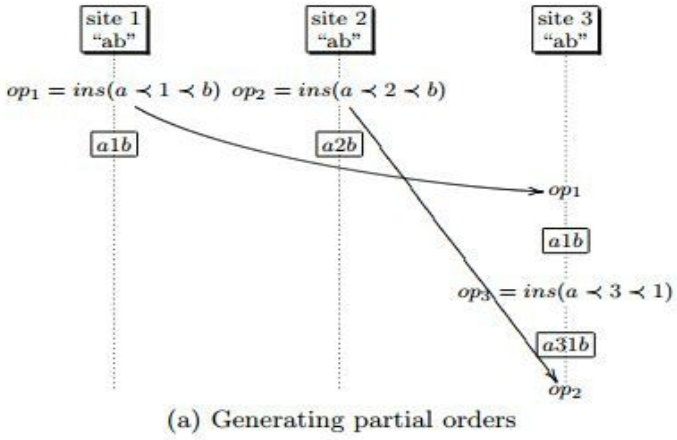
The WOOT Approach

The WOOT framework is meant to be a simpler alternative to the popular Operational Transform approach. WOOT ensures consistency between replicas using three properties:

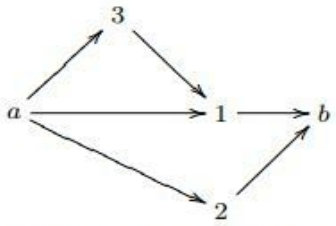
1. Convergence: Replicas of a document will agree on its state once the same set of operations has been performed on each one.
2. Intention Preservation: Performing an operation at a remote replica has the effect that the user intended and any concurrent operations do not interfere.
3. Precondition Preservation: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

To ensure the convergence of different 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. Hence, each site will establish an identical ordering irrespective of the order in which operations are received; it is simply based on the IDs of each character.



(a) Generating partial orders



(b) Ordering relationships

Figure 2: Partial orderings

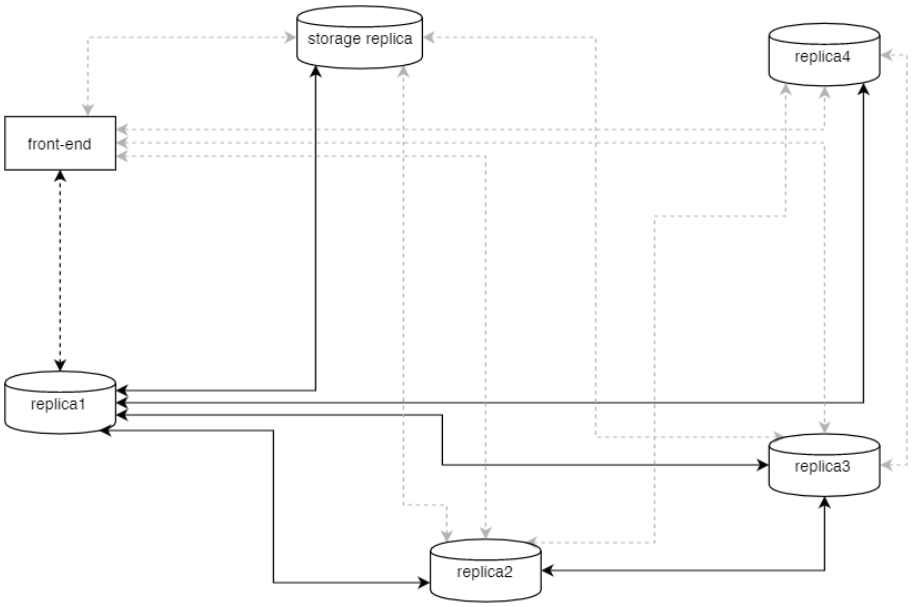
WOOT outlines three stages: The *generation stage* is where an operation is first integrated at its local site, and broadcast to all other sites to be integrated. The *reception stage* is where a site verifies the preconditions of an operation (for example, the existence of *b* if *del(b)* is called). 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).

Design

At a high level, our system is a Peer-to-Peer network where each user has a local copy of the document that they are editing, which sends and receives changes to/from the other peers. Our design makes use of two different nodes: The *front-end* service and the *replicas*.

Starting with the front-end, it performs a similar role to that of the tracker in a BitTorrent system. The front-end is run on a single machine that all clients must know the address of. Clients initially connect to the front-end from which they are given a list of all the other clients currently in the P2P network. Anytime a new client joins, the front-end will also send out the address of this new client to all the other clients so they can update their lists. This is the only function the front-end serves, meaning a front-end death will only result in new clients not being able to join the network; currently connected clients are unaffected.

Replica nodes are where the main implementation lies. These nodes will be run by each client and will hold the local copy of the document they are currently editing. On edits, these nodes must handle the integration and then broadcast the changes to all other nodes on the network. They are also responsible for receiving changes from other nodes as they are made and integrating them into their local copy of the document. By following the WOOT procedure, all nodes will agree on a single ordering of characters to maintain consistency. Users are additionally able to choose different documents by using a document ID they know of.



A special replica on the network will be designated as a storage replica. This replica is assumed to persist. It will allow all peers to leave the network and have all documents remain available for when they rejoin later. Since this project is focused on consistency, not unavailability, the design requires that this node never die, lest documents be permanently lossed.

To work with the WOOT procedure, characters are represented as W-characters, which contain an ID which consists of the replicas ID and a logical clock, the character themselves, the ID of the character before and after them and a visibility flag. The document will be made up of a collection of W-strings, which are in turn a collection of W-characters and special beginning and ending markers. WOOT defines the deletion of a character to be the setting of its visibility flag to false, to eliminate ambiguity in the ordering of characters. For example, if the current

state of a document is “abc”, and one user deletes “c” but then receives an operation from another user to insert a new character between “b” and “c”, problems arise since there is no longer any knowledge of “c”.

Implementation

Our system was implemented using two programming languages: Go and Javascript. Both the front-end and replica are written using Go while the interface for the replica (where the user inputs text and sees the document) is written in Javascript. This resulted in us having a third type of node in our system, the webapp.

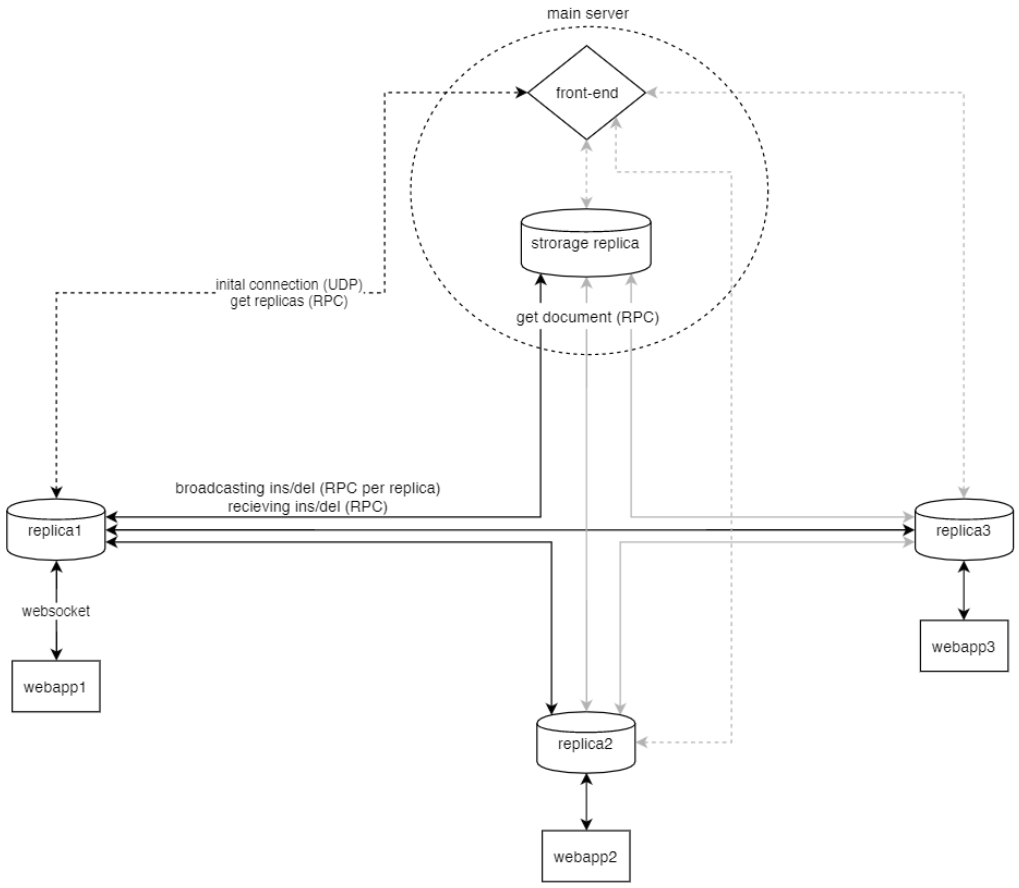
Initial communication between the front-end and a replica is done via UDP with the replica knowing the address of the front-end and sending over its own address and ID. The front-end then connects to the replica through RPC and sends over a list of all replicas in the system (including this new replica) and their address. This list is also sent to the existing replicas so they know of the new node as well.

System initialization involves firstly initializing the front-end and then initializing the storage replica. The storage replica, in particular, shares the exact same codebase as the replica, except that it stores multiple documents in the form of a key-value map where keys are the document IDs and the values are the documents themselves. The storage replica mainly acts as a simple persistence for all created documents. Newly joined replicas will always have access to the storage replica as it will always be the first replica to join the system before any other replica (as our specification requires it to be). Replicas can then retrieve documents from the storage replica and begin to propagate updates to their local copy of the document. Since updates are broadcasted to all replicas and a storage replica is in fact a modified version of a regular replica, the storage replica utilizes the same WOOT algorithms to integrate these changes to all of its documents in its key-value store so that it too remains up-to-date with other replicas.

The webapp is a simple web page being hosted by the replica on the user’s machine on port 8080. The logic of it is written in AngularJS and it simply communicates with the replica via a websocket upon certain events. When the web app is first opened, it requests a copy of the document from the local replica, which in turn requests a copy from the storage replica. The storage replica allocates a new document in its storage and sends it to the replica (via RPC) along with the document’s ID. The replica now has a copy of the document and can begin editing.

When the user enters any text, the web app sends the character and the position it was entered at to the replica. When backspace or delete is pressed, the position they were pressed at is sent. The web app is constantly listening to the replica (via a WebSocket) who will send an updated version of the document to the app whenever a change is made by this client, or any other replica on the network. Other users who intend to edit the same document can simply

open their browser and navigate to “localhost:8080/doc/<document-id>”, where <document-id> is the unique ID of the document (given that they have the replica code running on their machine as well). Note that navigation to such a link tells the replica to retrieve an existing document with that document ID from the storage replica in order to start working on it.



Replicas communicate amongst themselves via RPC. When a replica has inserted or deleted a character, it broadcasts the new W-character or deletion of a W-character to all replicas, who are constantly listening for these updates. Whenever an update is received from another replica it is put into a pool of pending operations. The replica constantly iterates over this pool, attempting to integrate the operations. This integration poses the problem of character ordering as mentioned in the *WOOT Approach* 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.

Edits to the document are processed as dictated by the WOOT procedure, meaning they will go through 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 isExecutable(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 it into its copy of the document. The insert will then be broadcast, via RPC, to all other replicas for them to also integrate. A more indepth look at the algorithms WOOT uses can be seen in “Data Consistency for P2P Collaborative Editing” by Oster et al. (2006)

Evaluation

The main way our system can be evaluated is through use. Running multiple clients who are all editing the same document should all be displaying the same content. This consistency comes from the WOOT procedure and we are merely evaluating the efficacy of the procedure in our system. We have seen that, given a low enough rate of character entry on each individual client (see Limitations section), our system does in fact maintain a consistency of intention among all clients on the network.

In order to test that the WOOT algorithms were functioning properly, preliminary test sets were written to ensure that each specific method worked as intended. These test sets were mostly verified using print statements, and covered the generation and integration stages of the protocol locally (i.e. not on multiple machines). For example, working on a test document, the semantics of generating an insert would be to integrate an insert locally and broadcast the insert to other replicas. Once basic functionality of the WOOT methods were covered, testing of the collaborative components and the web application itself began. This is also where the main testing of the correctness of WOOT occurred. As planned in our proposal, we completed this stage of testing through manual usage of the web app. Most tests were executed on two machines, with two clients making changes to the same document, but basic functionality was also verified on 3-4 machines.

Test cases include:

- Multiple replicas can connect through front-end and receive updated map of active nodes.
- Multiple users can connect to same document upon receiving appropriate URL.
- Once connected to the document, the replica is able to correctly retrieve the document's contents and update the application UI to reflect this.
- Users are able to concurrently modify a document (insert/delete characters), with changes being broadcasted to each other and received properly.
- Concurrent changes to the same part of a document (i.e. users both try to insert characters between "a" and "b") results in consistent state of document on each client. This is where WOOT comes into play to determine a total ordering.
- Calling deletes on an empty document does not break any functionality (e.g. neither special character at the start/end of a document's WString are deleted).
- Application can handle arbitrary sequences of inserts/deletes from multiple clients, with all of the clients ending up in a consistent state.
- Closing client connections and replicas (except for the storage replica) does not affect any functionality of the application.

These test cases all passed, and the application appeared to function correctly - it succeeded in handling collaborative editing in such a way that consistency was maintained.

ShiViz Logging

To create an event log for ShiViz, we incorporated the "govec" package to log events to a file in a ShiViz-compatible format. Each client node produces it's own log as well as the storage replica. The generated files are titled "client-Log.txt" and "replica-Log.txt" respectively. Due to our choice of using RPC to communicate between nodes and the limitations of the govec package, we used only a single function. This function takes an event message string, host ID string, and a properly formatted vector clock string, and writes them to a file. Since this function does not keep track of, or increment a logical clock, we keep track of the logical clock ourselves. Each node has its own logical clock that is independent of the others.

The events that are logged include: generating and integrating inserts and deletes, and receiving an operation. Generations and integrations are local events and do not show dependencies on remote events in the ShiViz visualiser. However, each node broadcasts its own local events to every other node so that they can reflect these events in their own logs. When broadcasting an operation to other nodes, the sending node encodes its own ID and current clock in the operation data structure so that the receivers can properly log a send/receive dependency.

The govec function `govec.Initialize(hostID, fileName)` creates the aforementioned text files and immediately writes the following two lines:

```
hostID{"hostID":1}
Initialization Complete
```

These lines are not needed and should be deleted. The default regular expression used by ShiViz is `(?<event>.*)\n(?<host>\S*) (?<clock>{.*})`, but the govec package prints host and clock first, then the event on the next line. So, the proper regex to use would be `(?<host>\S*) (?<clock>{.*})\n(?<event>.*)`.

Limitations

The main limitation of our project is the speed at which a user can input text before the ordering of their changes is lost (e.g. though you may intend to type “hello”, the protocol might decide the correct string is actually “ellho”). This is not a result of any flaws in the WOOT protocol, but of how our webapp communicates with the replica and the time it can take to integrate a new character. If the user types too quickly, subsequent characters will start to be processed on the replica before previous ones have finished, resulting in the newest character having the *prevChar* field in its W-character not set to the character that should come before it (as it hasn’t been fully integrated yet). To solve this, some sort of buffer or lock would be needed to ensure edits are processed one at a time locally and in the order the user intended.

Since we’ve focused on consistency and not availability, we chose to not consider the case that the storage replica fails. This means that unintended behaviour may be seen if this replica is killed, and it will result in the loss of the document if the other replicas were to leave. Furthermore, any new replicas that join the system will not be able to retrieve current documents, nor initialize new documents to be shared. Our system design has assumed that the machine that the front-end and the storage replica are being run on will always persist and never fail, giving us a large limitation on the possibility of real world use of our system. One approach to handling storage replica failure would be to replicate and maintain its contents amongst *r* replicas. Thus if one fails, our system could carry on.

Another potential issue is that WOOT keeps track of all W-Characters ever generated for a document, since deletion simply sets the character’s flag to invisible. This could cause memory issues as the amount of space needed to store a document could grow very large. One potential solution would be to establish a protocol for clearing a document of invisible characters. For example, the protocol might say that after x seconds of inactivity (not receiving/integrating any operations), a replica may clear all invisible W-Characters from it’s data structures.

Lastly, pushing all the logic off of our web app and onto the replica may have opened us up to some undesired behaviour in the web app, like momentarily losing a newly entered character. For example, if the webapp had just received a new copy of the document from its replica before the replica had integrated its last change, the change may disappear until the replica sends a new copy of the document. This may not be very noticeable in regular use, but it is unwanted.

Discussion

There were some implementation changes that were required to get the whole system communicating correctly when compared to our initial architecture detailed in the proposal. For one, there was the addition of a storage replica that allows for document persistence when there are no clients working on a document. Another change to our system was to forgo the notion of using a gossip protocol to broadcast changes and instead broadcast changes to all replicas. This change was made as broadcasting to only one replica on a change could induce problems with reliability if that replica died while the broadcast was inflight.

The obvious continuation of this project would take on unavailability. The assumption that the storage replica never dies is one that would not work in the real world. Some sort of data permanence would be needed so that documents that no one is editing wouldn't be lost forever if the storage replica died.

We were quite pleased with how the WOOT framework, and the paper describing it, worked. The majority of the project time was spent on implementing all the algorithms needed for it and the system to support it and just trusting that it would work once everyone was written and joined together. Luckily it did end up working, making our project a success and giving us a unique view into turning a research paper into a fully implemented project.

Allocation of Work

Jordan Coblin - Implemented the logic for WOOT, including the methods and data structures needed for the algorithms to function properly. Wrote tests to ensure functionality of WOOT methods. Integrated WOOT functionality with web application, linking the core components of the project together.

Chris Li - Implemented the front-end code for the listening of replica activity and failures. Set up the initial architecture of the system as well communication between multiple concurrent websockets and the replica. Implemented functionality for replicas to act as a storage which includes document initialization and retrieval. Added functionality for the broadcasting and processing of operations from different replicas and propagating these changes to the client's web app.

Sean Tyler - Initial implementation of functions and data structures for WOOT algorithm. Incorporated ShiViz logging to track events and write them to a ShiViz compatible log file.

John Wiebe - Was responsible for the web client, how it communicates with the local replica and the structure of the data sent between the two. This included getting character insertion events and their position, as well backspace/delete key press events, and sending them off to be processed, as well as integrating new changes. Worked on the revised architecture seen in the final project.

References

Oster, Gerald et al. "Data Consistency for P2P Collaborative Editing" HAL (2006)
https://hal.inria.fr/inria-00108523/document