DocEdit (Real-time P2P Collaborative Editor)

Authors (Name, ugrad id, student #)
De Li, u4o8, 33892126
Ke Er Xiong, y5q8, 10733129
Zi Yu Wang, s7l8, 40451122
Bo Yi Zhang, o7l8, 31206121

Abstract

In this report we describe the design and implementation of a real time collaborative text editor on a P2P network with no need of a central server. Peers can start new document and connect to other peers to edit on a shared document. They can also edit the shared document offline and reconnect later. This is solved using TreeDoc, a CRDT designed for collaborative editing.

Introduction

Our editor supports real time collaborative text editing on a P2P network. It enables users of this system to edit a piece of text collaboratively without the need for a central server like Google Docs which requires an account. It also support offline editing that systems like Google Docs do not. The editor can support up to 16 users (and possibly more) collaboratively editing on a single document, though we only expect to support at most 16 users considering collaborative editing is generally among small groups of people. There are two goals for this system. The first goal is to allow user to concurrently edit on a shared document without inconsistency and anomalies. This is solved based on a CRDT design named TreeDoc. The second goal is to build a P2P network topology which can update every connected client's document in near real time, as well as recovering from network failure and network partitioning. This paper will be focused on explaining the design to achieve the goals described above because they form the basis for the editor. Consult README.txt under application repository for installation and usage.

Design

TreeDoc

TreeDoc[1] is a CRDT designed for collaborative editing. It uses a binary tree to order the characters in the document. An in-order traversal of tree gives the text of the document (e.g. the document in figure 1 is "abcdef"). Each node in the binary tree is associated with a PosId. The PosId is a binary string that denotes a way to travel from the root to a node in the tree. A 0 in the PosId means to go the left child and a 1 means to go to the right child. An empty string is the root of the tree. For instance, the PosId "10" is the PosId for the root's right child's left child (in figure 1, this is character d). To insert a character between two characters, just insert it as the child node between the two nodes of the two said characters. For instance, to insert h between c and d in figure 1, insert it at "100" (the left child of d). To delete a character just use the PosId to refer to the character.

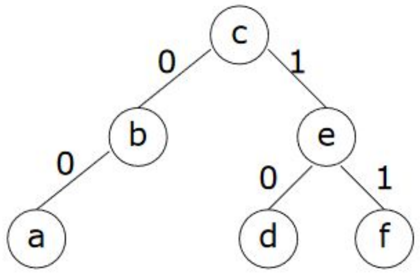

Figure 1: Image taken from the paper

The above paragraph describes TreeDoc when there's only a single peer (or site in the paper) since concurrent insertions to the TreeDoc can refer to the same PosId. To extend the tree structure above to support multiple peers, the TreeDoc uses disambiguators from each peer to order insertions to the same node in the tree (a node with disambiguators is referred to as a major node in the paper). The disambiguator need to be totally ordered (e.g. peer id where the insertion originated from). For instance, in figure 2, there are concurrent edits at 100. The PosId for W is 10(0:dW) and the PosId for Z is 10(0:dY)1. An in-order traversal or the tree will traverse through the disambiguators from the smaller disambiguator to the larger disambiguator. For instance, figure 2's text is "abcWXYZdef".

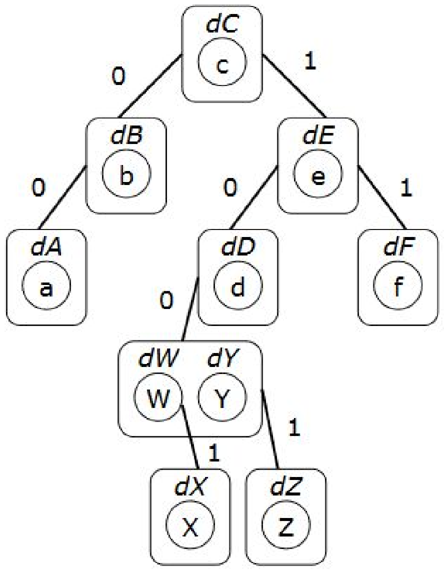

Figure 2: Image taken from the paper

Modifications to Suit Our Use Cases

Due to the open nature of the P2P network and the goal of offline editing, we cannot remove any disambiguators from a major node in the TreeDoc because a failed/disconnected peer can rejoin at any time. This significantly increases the size of the PosId from a single bit per node to the size of a disambiguator (which is 20 bytes in our case). As a result, we came up with the following solution. A disambiguator is unique and is generated from the site UUID concatenated with an incrementing counter (this type of disambiguator is mentioned in the paper). When we we insert or delete a node, instead of referring to the whole PosId, we simply refer to the disambiguator which will uniquely identifies the PosId. While the PosId can grow unbounded (that is, the tree itself can still have unbounded depth), the full PosId is never referred to directly. As a result, any operation send over the network is bounded in size, which is a great improvement over the original TreeDoc.

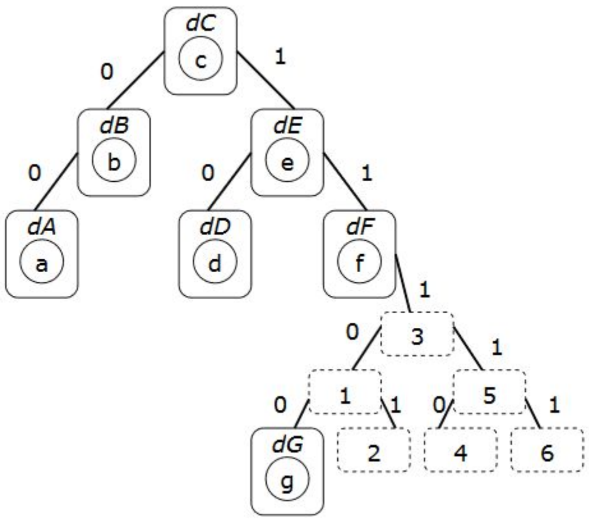

Figure 3: Image taken from the paper

In the TreeDoc paper, if the user types a long string, it causes the TreeDoc to become unbalanced as each character is added as the right child of the rightmost node (linear depth to the tree). The optimization described in the paper is to start inserting characters not directly as the right child but reserve a block of tree for insertion and instead insert at the leftmost child of that block (see figure 3). We developed a similar solution by making the tree n-ary instead of binary (n being 2^16=65536), so instead of having a bit for the left or right child node, we have 2 bytes to denote one of the 65536 nodes. Inserting a long string is simply inserting them as the adjacent child nodes of the parent node.

<u>Version Vectors</u>

The replica of the TreeDoc at each peer is associated with a version vector[2]. Each time a local operation is applied to the TreeDoc, the peer increments its counter in the version vector. Each operation on the TreeDoc is associated with the peer id it originated from and the counter of said peer in the version vector. For instance, let there be three peers A, B, C. An empty TreeDoc has version vector (A=0, B=0, C=0). A inserts a character. A's version vector is now (A=1, B=0, C=0). The insertion operation is operation 1 at peer A.

Replicas of TreeDoc are compared across sites using the version vector. For the rest of the report, v(x, y, z) refers to a version vector with x being the latest operation from peer A, y from peer B, and z from peer C, and op(x, y) refers to operation y at peer x. For instance, if peer A has a replica of TreeDoc with v(2, 1, 3) and node B has a replica of TreeDoc with v(2, 2, 2). Node A and node B has different replicas of the TreeDoc. Node A is missing op(B, 2) and mode B is missing op(C, 3).

Operations Order Anomaly

The TreeDoc paper assumes the operations are delivered in causal order. While it's not explicitly stated why in the paper, it's likely because of anomalies like this: suppose A writes "Hi, I'm A" and B writes "Hi, I'm B" concurrently. The link between A and B experiences congestion and they do not received each other's operation for a while. C, however received the operations from A and B and write "Hi, A and B". C's operations are then send back to A and B. Then one of A or B, let's say B in this case, is going to see "Hi, A and B" before seeing "I'm A". We will resolve the anomaly (like the TreeDoc paper suggests) by ensuring the following ordering constraint: let X be a local operation on the TreeDoc originating from A and Y be a local or remote operation applied to A's TreeDoc before operation X, then X will be applied after Y for all replicas of the TreeDoc in the system. In other words, a peer can't apply a remote operation until it's sure it sees a TreeDoc with enough operations applied to prevent the anomaly described above.

We use an algorithm using version vectors to ensure the above ordering guarantee (described below) that is similar to the algorithm of using vector clock to ensure causal ordering of messages in a distributed system[3]. Each operation is associated with the version vector of the TreeDoc at the time it's applied. An operation received is considered safe to apply to the local replica of the TreeDoc if the version vector associated with the message is less than the version vector of the local TreeDoc (i.e. the local TreeDoc has seen all operations that the remote operation has seen). A remote operation that cannot be safely applied to the local TreeDoc is put in a queue until the missing remote operations are received and applied to the TreeDoc first. For instance: A's current TreeDoc is empty (i.e. the version vector is (0, 0, 0)). If A received operation (C, 1) from C with (0, 1, 0), then it is put into the queue until operation (B, 1) is received. When it receives operation (B, 1) from B with version vector (0, 0, 0), it can be applied to the TreeDoc and operation (C, 1) is now safe to be applied to the TreeDoc.

Network

Our system is designed to support real-time editing in an open P2P network that has no constraints on the underlying topology and is resilient to peer failures. Our network is open in the sense that new peers can join and leave the network freely by connecting to any active peers in the network. To achieve this, at each peer, we keep track of all the peers in the system and whether they have left the network. We will refer to this information as the network metadata (NetMeta). The NetMeta provides the information needed to recover from network failures, deal with voluntary disconnection, and peer failures. Since peers can disconnect and then reconnect or fail and then restart using the same IP, we cannot identify the peers in the network by its IP alone. For example, if peer A fails, but the user restarts it with the same IP to connect to a separate network, we don't want peers in the old network to establish connection with it. Therefore, we use UUID version 1 (which generates an Id from the MAC address and current time) to identify a session in the network. Whenever a peer connect or reconnect, a new session id is created. Although UUID is not guaranteed to be unique, we can probabilistically expect no collisions as long as no one intentionally tries to break the system. Note that we didn't design the system with security as one of a goal.

To guarantee the consistency of the network metadata across peers, we use another CRDT (a grow only set) that contains the session id and the data related to the session (address and whether peer left or not). New peers can join and leave the system freely, but a session that has ended, is ended forever. This makes the changes in network metadata monotonic and eventually consistent across all peers. When a session connects to a peer in another network, its network metadata will be sent and merged with the one in the other peer and vise versa. The changes in network metadata will be broadcasted to all other peers in the now merged network. When a peer disconnects, the change in state will be broadcasted to all other peers eventually. The broadcasting mechanism will be discussed in details below.

To make the system real-time, we make the topology of the communication links between peers as complete as the underlying network condition allows. Each pair of peers in the system will try their best to connect to each other and communicate directly. When a change in the document occurs in a peer, the operation is broadcasted to all the other peers to which the current peer is connected. We refer to these peers as the neighbors of the current peer. This way, the latency of broadcasting operations in the system is minimized. If a connection cannot be established between two peers, they will try to reconnect to each other. Since we don't have any way of distinguishing between network failure and peer failure, we have to keep reconnecting to a peer. Each attempt to reconnect will increase the wait time until the next attempt to reconnect until an upper limit on the time interval is reached. As reconnections continue to fail, there is a higher chance that the peer/network will stay failed for a long period of time, so the peer reconnect less often to save resources. When peers disconnects (i.e. voluntarily leaves the network), they will broadcast this to network so we don't keep trying to reconnect to them. If peer A fails and restarts with the same listening address, and peer B in the previous network reconnects to A, A will notice that its current session id is different from the one expected by B, and reject the connection by sending a message back. B will then be able to determine that the previous session ended and deal with it accordingly.

To adapt to different underlying network topologies where a complete graph is not attainable, the peers recursively broadcast messages to other peers. Each message is tagged with the list of nodes that it's been sent to (i.e. visited peers). When a peer receives a message, it will send the message to nodes not in the visisted peers. For instance, if A is connected to B and C, B and C connected to each other, and B is connected to D. When A broadcast a message, it will tagged the message with {A, B, C}. When B and C receives the message, it will not send them to each other because they are in the visited peers. When B sends it to D, it tags the message with {A, B, C} + {A, C, D} = {A, B, C, D}. Some messages will be duplicated still (e.g. if C is connected to D, then B and C will both send it to D). However, in a complete graph, a message is never broadcasted twice.

Sometimes a message may failed to send properly because of network failure. To account for this, each peer periodically performs a version check by sending the version vector of the treedoc to its neighbors. If the receiver of the version check detects that the sender is missing any operations, it will send the missing operations over, therefore making sure the peers are in sync. The version check is also performed when a new connection is established between two peers to sync them as soon as possible. The version check also helps to heal network partition (and offline editing).

Implementation

The command line frontend GUI is implemented using the Go Termbox library[4]. It's not really tested except by manually using it.

<u>Document Manager</u>

The GUI send any local buffer operations to the Document Manager. The Document Manager atomically applies the local buffer operations, translate it to a TreeDoc operation and applies it to the TreeDoc. The TreeDoc operation is then broadcasted to the other peers through the Network Manager. Similarly, in the reverse case, the Network Manager receives a remote TreeDoc operation and atomically applies it to the local TreeDoc operations, translate it to local buffer operations and applies it to the local buff. All TreeDoc operations are logged. All TreeDoc operations not safe to applied are stored in a queue.

<u>Network Manager</u>

The Network Manager manages the network metadata and connections with other nodes in the system as described in the design section. It keeps a thread for reading and a thread for writing for each node. When a new message arrives it gets processed by a processing thread which then passes information to the Treedoc Manager as needed. When an operation is performed, the Treedoc Manager calls the Network Manager to broadcast it.

Evaluation

We test our editor by examining the behavior of our editor under previously designed test cases. Black Box manual testing is involved here because we also want to test the editor interface.

Condition: Normal network

Test Case	Scale	Action & Behavior
Local Editing	1 client	❖ *<u>Create New Document</u>* ➢ Blank document creation ❖ *<u>Local Insert/Delete</u>* ➢ Document updates on key input
Concurrent Editing	2 - 16 clients	❖ *<u>Connect client A to B</u>* ➢ Consistent merge of A and B's document ❖ *<u>One client edit the document</u>*

		➢ Edit shows in every connected clients' document ❖ *Multiple clients concurrently edit the document* ➢ All edits show in every connected clients' document ❖ *One client disconnects, then edit* ➢ Edit only shows in that client's window ❖ *One client disconnects, make edit, then reconnects* ➢ Offline edit merges with online edits made by other clients during the disconnection period. Every client gets updated with merged document. ❖ *Client Close Document* ➢ Client can only join editing by creating a new document and re-doing connecting ❖ *Client Exit* ➢ Client can only join editing by restarting the editor ❖ *Client Force Quit* ➢ Client can only join editing by restarting the editor

Condition: Network Failure & Network Partition

Test Case	Result
One client loses connection in a four clients network	The client can still do offline edit. It gets automatically connected few seconds after its network condition is set back to normal. Every client gets all the offline and online edits during the disconnection period.
Four clients lose connection in a seven clients network	The four disconnected clients can only update its local document. Once connection is back, every client gets all the offline and online edits during disconnection period.
A network of seven clients is partitioned to two subnetwork of three clients and four clients respectively	Both sub-network updates document based on its member's edits while they disconnect from each other. Once the two sub-network has an available connection, documents from two sub-network merge consistently.

Integration with ShiViz

The GoVector library[5] is used to generate ShiViz compatible logs. Since all messages play some sort of role in our system, we decided to log all distributed messages sent/received by peers. The decode method in UnpackReceive provided by GoVec library helps to unmarshal the messages. The id of the logger is the peer's address. Logging prints off the vector clock and message type information. The type and content of the messages are logged when sending them; while only the type of the messages are logged when receiving them. ShiViz is able to match up the sent and receive messages. An example of a line of log is:

```
localhost:3 {"localhost:3":20, "localhost:2":21}
```

```
send byte sendThread outChan msg versionCheckmap[]
Content: {da63cb28-0036-11e6-94ab-12f02f30986d
map[d959cbd8-0036-11e6-b18c-12f02f30986d:{localhost:2 false}
da63cb28-0036-11e6-94ab-12f02f30986d:{localhost:3 false}] [123 125]}
```

The regular expression parser for the log is :

```
(?<host>\S*) (?<clock>{.*})\n(?<event>[^\n]*)(\n(?<content>Content\: [^\n]*))?
```

Limitations

Safe garbage collection does not seem possible for the remote TreeDoc operations and tombstones in the TreeDoc when offline editing (which is equivalent to arbitrary long network partition) is a goal. As soon as a peer disconnects, it could be possible it reconnects later. Discarding remote operations or tombstones may lead to deleted characters reappearing. This is related to consensus in open networks because to garbage collect the tombstones, we need to make sure every peer that has a replica of the document up to a certain version (i.e. we achieve a consensus among peers what the latest version of the TreeDoc document everyone have) or we could lose information in the garbage collection. It seems impossible because once a peer disconnects, it can join other network, allowing the TreeDoc to spread outside the original network to any other peer, meaning we can never garbage collect safely in the original network.

While the TreeDoc CRDT allows the system to be strongly consistent without consensus, merging two replicas of TreeDoc can have anomalies. For instance, let's say A types "123" while not connected to any other peer, and similarly B types "456" and C types "789". A then connects with B and the resulting merged document is "123456". C then connects with A and B. While the more logical merging result is "123456789" or "789123456", the result can be "123789456" because concurrent disambiguators are ordered by the peer id which is random. This can happen on a smaller scale with subsections of text in offline editing that may be harder to correct for the users. However, other CRDTs we looked at suffers the same problem (e.g. Logoot). The only way to solve this problem seem to require some sort of manual conflict resolution.

Discussion

A lot of the challenges we encountered are implementation based. For instance, in our first attempt to implement the network manager, we didn’t have separate threads for writing to each node. This required a lot of locks on various data structures and resulted in a lot of blocking operations. After we switched to using channels and sending threads, the program became a lot easier to reason about and write. In addition, the lack of proper logging (which is our fault) until toward the end made tracking certain bugs down longer than it really should. On the other hands, the components of Document Manager had good unit test coverage which allows us to be more confident about its correctness

Allocation of Work

De Li: TreeDoc, GUI
Ke Er Xiong: Network Manager
Zi Yu Wang: Testing and Treedoc integration
Bo Yi Zhang: ShiViz integration and Testing

References

1. https://hal.inria.fr/file/index/docid/445975/filename/icdcs09-treedoc.pdf
2. https://en.wikipedia.org/wiki/Version_vector
3. https://www.cl.cam.ac.uk/teaching/0910/ConcDistS/10b-ProcGp-order.pdf
4. https://github.com/nsf/termbox-go
5. https://github.com/arcaneiceman/GoVector