Alex Dalgleish
Robinson College
`amd96`

Computer Science Tripos — Part II
2017

A Library for P2P Collaborative Editing with CRDTs

Proforma

Name:	Alex Dalgleish
College:	Robinson College
Project Title:	A Library for P2P Collaborative Editing with CRDTs
Examination:	Computer Science Tripos — Part II, 2017
Word Count:	—-
Project Originator:	Mr Stephan A. Kollmann
Supervisor:	Mr Stephan A. Kollmann

Original Aim of the Project

In this project I aimed to build a library for doing peer-to-peer collaborative editing with CRDTs as data structures. This could be used easily by an application like a TODO list or general text editor to allow for collaboration over a network. The peer-to-peer approach was taken to mitigate privacy concerns when data flows through untrusted servers.

Work Completed

The library was successfully built, along with an example toy text editor using it. In addition, two contrasting CRDT approaches have been used and compared, support for local undoing of operations has been added, and there is the option of communicating over the Tor overlay network for anonymous collaboration.

Special Difficulties

None.

Declaration of Originality

I, Alexander Dalgleish of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Contents

Chapter 1

Introduction

A common use of the Internet is to perform some kind of online editing of documents. Real-time collaborative editors allow multiple clients to edit a document simultaneously, regardless of location. One popular such editor is provided as part of Google Drive[1]. However, by using this service, you are giving Google access to everything you write, as this extract from the Google Terms Of Service state:

> When you upload, submit, store, send or receive content to or through our Services, you give Google (and those we work with) a worldwide license to use, host, store, reproduce, modify, create derivative works (such as those resulting from translations, adaptations or other changes we make so that your content works better with our Services), communicate, publish, publicly perform, publicly display and distribute such content.

For the privacy conscious, it may be undesirable for Google to be able to *publish* or *distribute* your content. Instead of sending all your writings through large, corporate servers (the client-server architecture) that may inspect the data as they please, my project makes use of the peer-to-peer (P2P) architecture, so that all data about a document is stored only by those editing it.

A P2P real-time collaborative editor is inherently a distributed system, and as such suffers from the usual pitfalls of distributed systems. For an editor we don't necessarily care about *physical* time, only which events happened before others, referred to as *logical* time. However, in some cases events A and B might not be ordered by the happens-before [1] relation (they are *concurrent*).

Imagine a simple situation with two clients A and B, who are editing the same document. A types an a, and tells B about this. Before B receives the message, it types b and tells A about this. Thus, naïvely, A would have a final state of ab and B would have ba. A correct editor would somehow ensure that the final state of any

[1] https://www.google.com/drive/

two clients editing the same document is the same. The first part of my project focuses on how this can be done.

So far, I have considered the privacy concerns for storing data at a node in a network through which data must travel. However, in the Internet, intermediate nodes are not necessarily friendly, and may inspect the data as they wish[2]. Moreover, they can read the packet headers to find the source and destination of all the data, and so potentially can infer who you are collaborating with. The Onion Router (Tor) is a project that aims primarily to mitigate the latter concern. Its operation will be described in the Preparation chapter, but using it means your packets randomly hop around the world before reaching the destination, such that none of the intermediate nodes know both the sender and recipient of the message. Moreover, the use of its Hidden Services means additionally no intermediate node can read your data, so Tor is a convenient way to address both problems presented, and the second part of the project builds up to this.

[2] https://en.wikipedia.org/wiki/Deep_packet_inspection

Chapter 2

Preparation

2.1 Distributed Systems

In order to serve arbitrary volumes of read/write requests for data, we replicate it across different machines and allow the requests to be split amongst them. I will use *replica* to refer to a node in a network that is part of a distributed system. By default, it is connected to all other replicas and has some state that it wishes to keep consistent with them, which it does by sending/receiving updates about that state.

One formulation of the popular CAP Theorem [2] is that for a distributed system you can have at most two of Consistency, Availability and Partition Tolerance, though this has been criticised as misleading [3], and a clearer statement would be that when a network partition occurs, you can choose to have consistency or availability, but not both. A network partition is where the system is divided into disjoint subsets such that each subset is connected internally, but disconnected from every other subset. In such an event, a modification to one of the replicas in a subset would mean that all the replicas in different subsets would have stale values, and the system would be inconsistent, unless all the other subsets went down so that the modified subset only was available. This is the consistency-availability trade-off the theorem mentions.

2.2 CRDTs

Conflict-Free Replicated Data Types (CRDTs) are a class of data structure designed to specifically for distributed systems to provide Strong Eventual Consistency (SEC).

The CRDT approach is to be available and sacrifice strong consistency (the system behaves as if no replication is present - a read will always give the value of the most recent write). That is, in a network partition, all parts of the system can function, they just may give different results on a read as they can't communicate updates.

For some CRDT-based system, let $o \in \mathcal{O}$ be an operation that can be applied to the state, and $s_i \in \mathcal{S}$ be a possible state in replica i. Also, let $u_i \in \mathcal{U}$ be a set of updates to the state seen by replica i.

Eventually consistent systems have the property that any two replicas with the same set of updates will eventually reach equivalent state. However, some conflict resolution process might be needed to reach the state. Strong Eventual Consistency (SEC) goes a step further to assert that as soon as the set of updates are the same, the states will be equivalent. This means no conflict resolution process is needed. That is, $\forall$ replicas $i, j.\ \ u_i = u_j \Rightarrow s_i \equiv s_j$.

CRDTs fall into two broad categories: State-based and Operation(op)-based. A state based system has a function $\textsc{update}$: $\mathcal{O} \times \mathcal{S} \rightarrow \mathcal{S}$ which applies operations locally, then distributes the new state to other replicas. On receipt of the new state, another replica applies the function $\textsc{merge}$: $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ for combining incoming states with the local state. This merge function is associative, commutative and idempotent, so that (assuming the states are distributed properly, a liveness guarantee) all replicas will reach equivalent state which is the composed MERGE of all received states.

The op-based approach is similar, but instead of distributing new states, operations are sent to other replicas. The UPDATE function is split into two. ATSOURCE is performed only at the operation's source replica, whereas DOWNSTREAM is performed at all replicas. This approach additionally requires operations to be delivered only once (or alternatively be idempotent), as for example sending an *increment counter* operation and the network duplicating it means the counter would be incremented twice at other replicas, but only once at the source, giving inconsistent state. Furthermore, if operation A *happens-before* [1] operation B, A should be delivered before B at all replicas. Usually some messaging middleware provides these exactly-once and causally-ordered delivery guarantees. Finally, for all concurrent operations (those not ordered by *happens-before*), the DOWNSTREAM phase commutes. So, for concurrent operations o_1 and o_2:

$$\textsc{downstream}(o_1, \textsc{downstream}(o_2, s)) \equiv \textsc{downstream}(o_2, \textsc{downstream}(o_1, s))$$

Hence, there are advantages and disadvantages of each approach. On the one hand, state-based CRDTs require transmitting the entire state every time an update is made. When you have a large state and lots of replicas, this can use a lot of bandwidth. Conversely, op-based CRDTs may save you in bandwidth, but require extra messaging guarantees. As well as this, a state-based CRDT can send one state representing multiple updates at once, whereas in the other case every single operation has to be distributed, so there is certainly a tradeoff based on the state size and rate of updates the system has as to which makes more efficient use of the network.

For my collaborative editing scenario, in general the size of the document will be much larger than any update to it. So, sending the whole document around the network on every update will be much less efficient than just sending the new operations that need to be performed. However, my editor supports offline editing, so upon reconnecting it may be the case that lots of operations are sent separately and it may have been more efficient just to send the state. Despite this, I decided it was better to optimise for the online case (which I expected to be the more common case, appealing to Amdahl's Law) and hence I chose the op-based variety.

2.2.1 Ordered Lists

I define an ordered list as a set of vertices ordered totally by some binary relation $<_v$. For collaborative editing, I use CRDTs to represent an ordered list $\langle v_1, ..., v_n \rangle$ where vertex $v_i = (a_i, id_i)$ is a tuple of an atom $a_i \in \mathcal{A}$ and an identifier $id_i \in \mathcal{ID}$. The document represented is then the concatenation of atoms projected from the ordered vertices. In my implementation, $\mathcal{A}$ is the set of single characters, though in other work it has been the set of all possible whole lines of characters [4]. For two lists l and l', let $l \equiv_l l'$ if and only if the documents represented by them are the same.

Initially, the state $s = \langle \vdash, \dashv \rangle$, two special, invisible vertices such that $\forall$ other vertices v,

$$\textsc{identifier}(\vdash) < \textsc{identifier}(v) \;\wedge\; \textsc{identifier}(v) < \textsc{identifier}(\dashv)$$

There are different approaches people have taken to representing an ordered list with a CRDT, and I have implemented two of these.

2.2.2 Tombstoning Approach

One approach, called the Replicated Growable Array (RGA) is described in [5]. A vertex in this system looks like:

$$v \equiv (a, (t, rid))$$

The identifier for a vertex is a pair of a timestamp $t \in \mathbb{N}$ and a globally unique replica identifier $rid \in \mathbb{N}$ (so $\mathcal{ID} = \mathbb{N} \times \mathbb{N}$). Identifiers are ordered first by timestamp, then by *rid*. Each vertex additionally has a boolean property `deleted`, which is initially false. Let there be projection functions associated with all these properties named similarly. The timestamp is such that if the insertion of one vertex v *happens-before* the insertion of another v', $\textsc{timestamp}(v) < \textsc{timestamp}(v')$. The state represented is the sequence of projected atoms a of the vertices whose `deleted` property is false only. The ordering $<_v$ follows from the order on the identifiers.

RGA supports the following operations:

```
RGA
Require: $v \in s$
function SUCCESSOR($v$)
    return $\min \{v' \mid v' \in s \ \wedge \ v <_v v'\}$

Require: state $s = \langle ..., v_1, v_l, v_2, ... \rangle$
function AddRight($v_l$, $a$)
atSource:
    $t \leftarrow$ NOW( )                                ▷ Generates a fresh timestamp
    $v \leftarrow (a, (t, rid))$
DOWNSTREAM:
    $l \leftarrow v_l$
    $r \leftarrow$ SUCCESSOR($v_l$)
    while IDENTIFER($v$) < IDENTIFIER($r$) do
        $l, \ r \leftarrow r,$ SUCCESSOR($r$)
    $s \leftarrow \langle ..., l, v, r, ... \rangle$
Ensure: $v \in s \ \wedge \ v_l <_v v \ \wedge \ \forall v'. \ v_l <_v v' <_v v.$ IDENTIFIER($v'$) < IDENTIFIER($v$)

Require: state $s = \langle ..., v_l, v, v_r, ... \rangle$
function Delete($v$)
DOWNSTREAM:
    $v$.deleted = True
Ensure: state $s = \langle ..., v_l, v, v_r, ... \rangle \ \wedge$ deleted($v$)
```

The obvious weakness of this approach is that, since no vertex is ever actually deleted, just marked as deleted (*tombstoned*), the memory used by such a CRDT never decreases with time. So, you could have an empty document in front of you that contains a million vertices all marked as deleted which is obviously not ideal.

2.2.3 Variable-size Identifier Approach

Whereas vertex identifiers consist of a single timestamp and a replica identifier, a different system (used in Logoot [4]) has identifiers of unbounded size, but in doing this vertices are allowed to be properly deleted.

So, identifiers are now a globally unique 3-tuple of a positional identifier ($p \in \mathbb{N}^*$), replica identifier ($rid \in \mathbb{N}$) and timestamp ($t \in \mathbb{N}$). Here, $\mathcal{ID} = \mathbb{N}^* \times \mathbb{N} \times \mathbb{N}$. The positional identifier is a sequence of numbers representing a path in a tree.

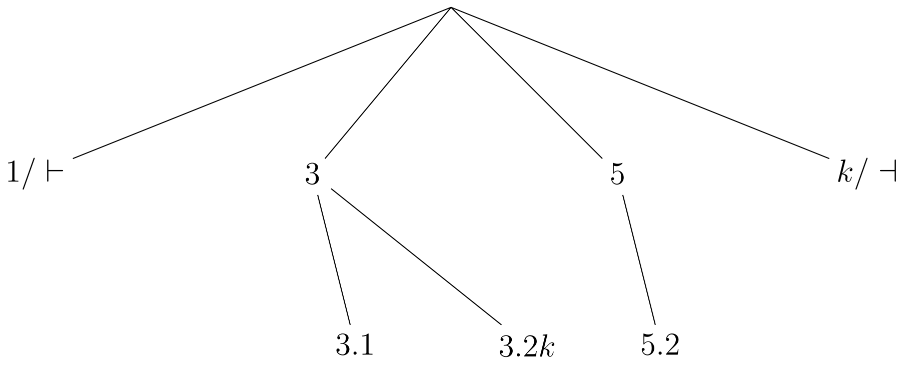


Figure 2.1: The tree-like structure of identifiers. Each number in the position represents a location to the right of its parent one level down in the tree. The outer two nodes at the top level are the positions of the start and end vertex, $\vdash$ and $\dashv$.

Define the ordering of the vertices $<_v$ by the total ordering on their identifiers $\prec$ from [4]: Let:

$$id_1 = (p_1, rid_1, t_1)$$
$$id_2 = (p_2, rid_2, t_2)$$

And, (wlog. assume $n \leq m$)

$$p_1 = p_1^0.p_1^1...p_1^n$$
$$p_2 = p_2^0.p_2^1...p_2^m$$

Then,

$$p_1 \prec p_2 \Leftrightarrow \exists j \leq m.\ (\forall i < j.\ p_1^i = p_2^i) \wedge (j = n + 1 \vee p_1^j < p_2^j)$$
$$p_1 = p_2 \Leftrightarrow (n = m)\ \wedge\ \forall i.p_1^i = p_2^i$$

Finally,

$$id_1 \prec id_2 \Leftrightarrow (p_1 \prec p_2) \vee ((p_1 = p_2) \wedge (rid_1 < rid_2)) \vee ((p_1 = p_2) \wedge (rid_1 = rid_2) \wedge (t_1 < t_2))$$

I use the base-doubling strategy from [4] so that the there are k possible positions at depth 1, $2k$ at depth 2 et cetera. Also, positions are represented internally as single numbers such that the lower $\log_2(base(1))$ bits are the position at depth 1, the next $\log_2(base(2))$ bits are the position at depth 2 et cetera. Therefore, subtraction of positions is just integer subtraction.

```
Logoot
Require: $v \in s$
function SUCCESSOR($v$)
    return $\min \{v' \mid v' \in s \ \wedge \ v <_v v'\}$

function PREFIX($p$, $depth$)
    $p\_copy \leftarrow [\,]$                                ▷ The empty sequence
    $d \leftarrow 1$
    while $d < depth$ do
        if $d < p.length$ then $p\_copy = p\_copy.append(p^d)$
        else$p\_copy = p\_copy.append(0_{base(d)})$         ▷ such that the binary representation of 0 uses $\log_2(base(d))$ digits
        $d \leftarrow d + 1$

function ALLOC($p_l$)                                       ▷ Logoot's boundary strategy
    $p_r \leftarrow$ SUCCESSOR($p_l$)
    $depth \leftarrow 0$
    $interval \leftarrow 0$
    while $interval < 1$ do                                 ▷ Find a gap to insert in
        $depth \leftarrow depth + 1$
        $interval =$ PREFIX($p_r, depth$) $-$ PREFIX($p_l, depth$) $- 1$
    $step \leftarrow \min(boundary, interval)$
    $offset \in_R [0, step]$
    return PREFIX($p_l, depth$) $+ offset$

function ADDRIGHT($v_l$, $a$)
ATSOURCE
    $t \leftarrow$ NOW( )                                   ▷ Generates a fresh timestamp
    $newPosition \leftarrow$ALLOC($v_l.id.p$)
    $v \leftarrow (a, (newPosition, rid, t)$
DOWNSTREAM
    $s \leftarrow s \cup \{v\}$
Ensure: state $s = \langle ..., v_l, v, v_r, ... \rangle$ such that $v_l <_v v <_v v_r$

Require: state $s = \langle ..., v_l, v, v_r, ... \rangle \vee v \notin s$
function DELETE($v$)
    if $v \in s$ then
        $s \leftarrow \langle ..., v_l, v_r, ... \rangle$
Ensure: $v \notin s$
```

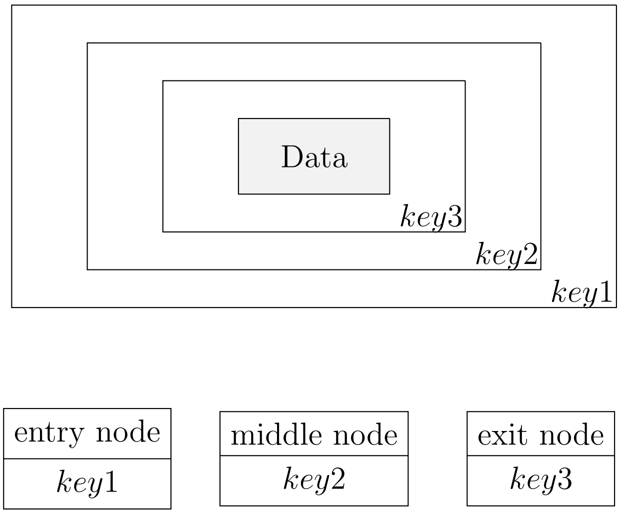
A proposed improvement to this scheme is called LSEQ [6]. It provides simple changes to ALLOC that claims to improve memory efficiency over plain Logoot. For each replica, a mapping from depth to a randomly generated bit is stored. In ALLOC, if the bit for the required depth is 0, return PREFIX($p_l, depth$) + *offset*, otherwise return PREFIX($p_r, depth$) − *offset*. Essentially, for each level in the tree you choose randomly whether to insert close to the left or right elements when inserting between them. If you always insert close to the left element, there will be more free identifiers at this depth between the new element and its successor, but few between the new one and its predecessor. The converse is true when you insert close to the right. To save memory, a good allocation scheme would allocate positions at the smallest depths possible based on future knowledge of what will be inserted, but this obviously isn't possible, so this scheme chooses randomly in the hope of doing well. I implemented LSEQ to be able to contrast it with RGA.

2.3 Tor

2.3.1 Onion routing

The Tor[1] overlay network is designed to provide anonymity to its users online. Its name stems from *The Onion Router*, describing how packets are sent around its network. Data is wrapped up in layers of encryption at the source, then this *onion* (because it has layers) is gradually unwrapped as it hops around the network, until it reaches the destination unencrypted.

When a source S wants to send some data to a destination D, it chooses a number of Tor *relays* to send the data through (typically 3). The first is called the *entry node*, the last the *exit node*, and any intermediate nodes *middle nodes*. S contacts the entry node directly, and establishes a shared key with it. Then, it asks the entry node to extend the Tor *circuit* to the chosen middle node, and establishes a shared key with that. The process is repeated to reach the exit node. When the whole circuit is established (and S has 3 separate keys), S encrypts the data with all the keys sequentially, in reverse of acquisition order, and sends the data to the entry node. The entry node decrypts the outer layer and forwards it to the middle node, similarly for the middle node, then the exit node decrypts the final layer and forwards to the destination D.



[1] https://www.torproject.org

In this way, only the exit node knows the address of D, and only the entry node knows the address of S, and D knows only the address of the exit node. So, nobody except S knows the address of both S and D for sure. However, the link between the exit node and D is unencrypted, so for confidentiality, encryption at the transport or application layer is needed.

2.3.2 Hidden Services

A traditional service can be configured to accept incoming connections only over Tor, when it is called a *Hidden Service* (HS). A client can connect to a HS by specifying its *onion address*, a 16 character string derived from a hash of a special public key it owns. In this way, a HS can hide its location from clients, improving on the asymmetry of knowledge in traditional Onion Routing, so that both parties are anonymous to each other.

After looking up information about the HS in special directories, the client chooses a random Tor relay as a rendezvous point (RP) and both the client and the HS build a Tor circuit to that. The RP relays end-to-end encrypted messages between the client and the HS. Thus the client and HS know only the address of the RP and not each other, whilst the RP knows nothing about either.

2.3.3 Client Authorization

Since the onion address of a HS is derivable from its public key, a connecting client can verify it is connecting to the expected HS. However, the HS knows nothing about the client. It may be desirable for a HS to verify the clients which can build a circuit to it, so the HS protocol supports a couple of methods of client authorization. The one I made use of is called *stealth* authorization.

The HS essentially creates a different identity for each client that wishes to connect, then passes secrets corresponding to each identity to each client through some other channel. When a client looks up the HS in the directory, it finds the entry corresponding to the identity it was told about, and decrypts the HS information using the secret it was given. So, only allowed clients are allowed to even lookup *how* to connect to an authorizing HS. Unfortunately, at the moment the protocol only allows 16 separately authorized clients per HS, but I decided that this was sufficient for my needs on this project.

2.4 Project Development

2.4.1 Choice of platform

At the beginning of the project, I had to decide in what form an application using my library would be in. I narrowed it down to three choices:

- A web application using JavaScript
- An Android application, using Java and the Android API
- A desktop application using Python

I had a little experience with JavaScript and Python on a previous internship, and obviously Java from the Part I Tripos courses. I first ruled out the Android application, as there would have been significant overhead in learning the Android framework properly, and also in actually developing and testing the app. Debugging appeared to be significantly harder remotely. As I knew I wanted to interact with Tor, I needed some way for my library to interact with a Tor daemon running on the same machine. The Tor project actively supports a library for just this called Stem in Python, so that was what finalised my decision to design a desktop application.

2.4.2 Existing Code

I installed Tor locally, which exposes a SOCKS5 proxy to send data to and runs a daemon process which implements the correct protocols for using the Tor network. To interact with the Tor process, I used the Stem[2] Python library, which provides convenient API calls to use hidden services. Other minor libraries used are described in the next section.

2.4.3 Development Plan

I decided to iteratively develop the code for my project based on each of my goals stated in the project proposal. That way, I could plan and evaluate each part separately. Moreover, a single-pass *Waterfall*-like approach would have been problematic as it is then difficult to adapt to changing or refined requirements after everything is planned. Also, planning a large project such as this before any code was written would have been very difficult as writing code gives you a good insight into further design.

[2] https://stem.torproject.org/

Chapter 3

Implementation

3.1 Implementation tools

Since I was building a distributed system, it was useful to find a way to simulate multiple replicas locally on one machine. I therefore used Docker[1] to run instances of an application in a *container*, with the host's X11 socket being shared with each *container*, so that I could interact with each instance's GUI when it was built. This allowed for some basic bug finding in the library and application and so helped me to write unit tests to cover those cases.

I also made use of the JetBrains PyCharm IDE[2] to easily manage large numbers of files and find and replace across them. Its feature-rich debugger was also very helpful in finding sources of errors in the code.

3.2 CRDT Ordered List Library

I began the project by creating classes for common things I would be representing in the project, making use of Python's Object Oriented features. As shown in figure 3.1, I decided to differentiate between operations that originated from this replica (*local*) and from another replica over some network (*remote*). Thus, performing a *local* operation executes both atSource and downstream from the updating functions, whilst performing a *remote* operation executes only downstream. Additionally, performing a local operation uses the *cursor*, state kept locally to each replica, which is an identifier of a vertex v_C. Performing `OpAddRightLocal(`a`)` will insert the character a immediately to the right of v_C. Performing a `OpDeleteLocal()` deletes v_C.

[1] https://www.docker.com/

[2] https://www.jetbrains.com/pycharm/

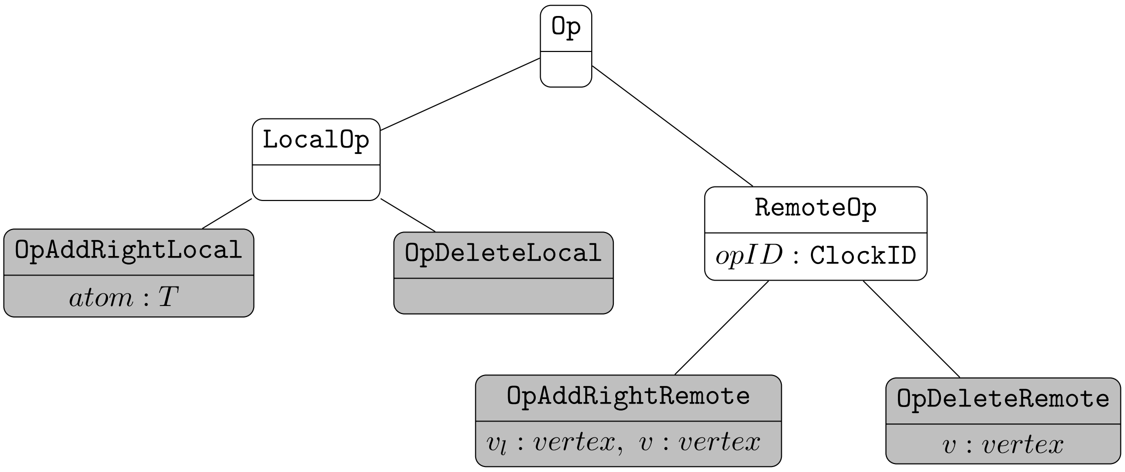


Figure 3.1: Class hierarchy for the supported CRDT operations. Shaded classes are concrete. Type T must be representable as a string.

I also created an `Identifier` abstract class to represent the identifiers in the vertices. Since RGA and LSEQ use different kinds of identifiers, they have different implementations of this (`ClockID` and `PathID` respectively).

I wanted to be able to perform operations on and get representations of the CRDT without knowing its underlying representation (the concept of encapsulation in OOP), so created another abstract class `BaseOrderedList` from which my implementations of RGA and LSEQ (`LLOrderedList` and `LSEQOrderedList` respectively) inherit.

Finally, the class `ListCRDT` was created to take a `BaseOrderedList`, be given `Op` objects to perform on it, and be queried for its representation. Performing a *local* operation returns the equivalent *remote* operation that should be performed at other replicas. For example, performing a `OpDeleteLocal()` with the cursor at v_C returns a `OpDeleteRemote(`v_C`)`, which will have the same effect on the representation of the list at other replicas.

A design decision was made not to support nested CRDTs (each atom is itself a CRDT) for simplicity, as moving cursors around them quickly becomes non-trivial. So, as long as an atom can be represented as a string (which all python objects can through the method STR), it is valid, and the representation of that atom is STR(*atom*).

Here is example Python code for how `ListCRDT` performs a `OpAddRightLocal` operation. It maintains a `ClockID` object *clock*, which has a timestamp, the *rid*, and an INCREMENT function to increment the timestamp. The *clock* holds the timestamp and *rid* of the last locally inserted vertex, so is incremented before inserting a new one. This is the implementation of the NOW function as described in the previous chapter. *self.olist* is a reference to a `BaseOrderedList` object.

```
def addRight(local_add_op):
  # the atom to insert
  atom = local_add_op.atom

  # generate fresh timestamp
  self.clock.increment()

  # returns the new vertex and its predecessor in the list
  left_vertex, vertex_added =
    self.olist.addRight(self.cursor, (atom, self.clock))

  # update the cursor so that a further insert will insert just after the new vertex
  self.cursor = vertex_added.identifier

  return OpAddRightRemote(left_vertex, vertex_added)
```

Figure 3.2: How `ListCRDT` performs a `OpAddRightLocal`

3.2.1 Naïve Implementation - `ArrOrderedList`

My first attempt at an implementation of `BaseOrderedList` for RGA was ArrOrderedList, which uses Python's builtin `list` (an array) to store the vertices. There is then a `dict` to lookup the position of each vertex in the list in constant time. However, inserting into the array takes time linear in the size of the list, giving AddRight a cost of $O(n)$ for n the current number of vertices stored. This wasn't ideal, so I tried again, but kept this implementation for comparison.

3.2.2 RGA - `LLOrderedList`

Since the data structure the CRDTs represent is an ordered list, it makes sense to hold the vertices as a linked list, as insertion and deletion on arbitrary positions in the list are independent of its size (unlike Python's builtin `list`). However, in a standard linked list, the time taken to find an element is linear in its size, so to improve this I store a hash table mapping a vertex's identifier to the node in the list, giving a constant time lookup. Hence, both the AddRight and Delete functions can be performed on the list in constant ($O(1)$) time in the average case (if the dictionary is sufficiently balanced).

3.2.3 LSEQ - `LSEQOrderedList`

For LSEQ, it is possible to know where to insert a vertex purely based on its position. Therefore, ADDRIGHT(v_l, v) can be simplified to ADD(v) (this will come in useful when implementing *undo* in section 3.2.4). However, this means a data structure that can support this is needed. So, the requirements for such a data structure are:

- Easy to find next and previous elements
- Fast lookup of an element given its identifier
- Fast lookup of the element with the largest identifier smaller than a given identifier (the in-order predecessor)

The linked list approach used for RGA would satisfy the first two criteria, but would require a linear scan for the third. The tree-like nature of the positions in LSEQ suggests a tree-like data structure, and indeed I found the SortedList[3]. It offers roughly logarithmic time insertion and deletion (verified in the Evaluation chapter), and has a method BISECT_LEFT for finding the index of the smallest element greater than a given one (meaning one less than this index points to the largest element smaller than the given one).

While implementing LSEQ, I came across a further improvement to the scheme [7]. The motivation was that replicas would all be choosing different schemes, so it would be better if they could collude. The h-LSEQ scheme achieves this by having a pseudorandom sequence generated by a seed shared by all replicas. Thus, each replica uses LSEQ, but all replicas have the same strategies.

3.2.4 Undo/Redo

Having seen a paper implementing a global undo for a CRDT-based editor [8], I decided to also implement an undo function, but only locally. That is, a replica can undo and redo the operations it originated, but no others. Being able to undo all replicas' operations seemed messy and frustrating from a user's perspective.

After an operation is performed, it is stored in a list specific to its originating replica (see section 3.3 for details). As shown in figure 3.3, the operations originating at this replica can be undone by popping them from *opStore* (for this replica), performing their inverse and pushing that to a special stack of undone operations. To redo, pop from *undoStack*, perform the inverse operation, and push that to *opStore*. To avoid a branching history, when a new operation originating at this replica is performed (one that has not been undone or redone), *undoStack* is emptied.

[3] http://www.grantjenks.com/docs/sortedcontainers/sortedlist.html

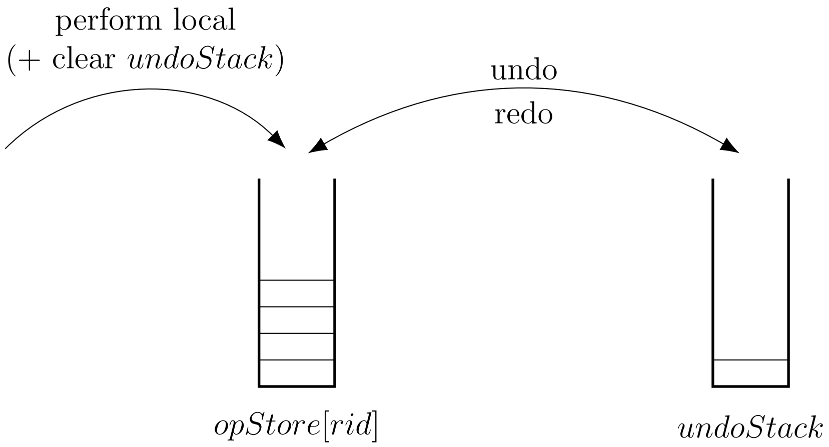


Figure 3.3: How operations are undone and redone at replica with id *rid*

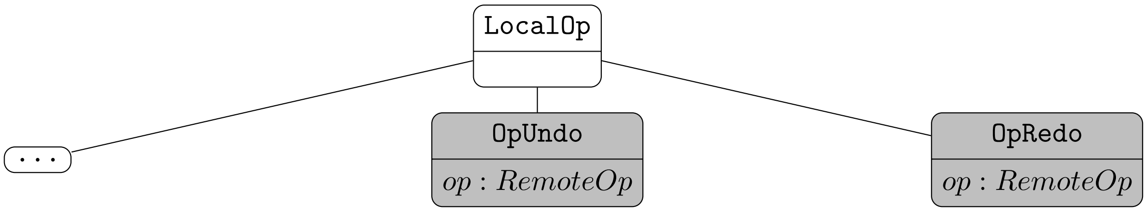


Figure 3.4: How OpUndo and OpRedo fit into the operation hierarchy

When the application initiates an undo (respectively redo), an `OpUndo` (`OpRedo`) operation is created. This is given the most recent operation from *opStack* of this replica (*undoStack*). The `ListCRDT` object has logic for taking an operation and performing the inverse of it on the `BaseOrderedList`. This UNDO function is the same for `OpUndo` and `OpRedo` as to redo an operation is to undo the effects of its undoing.

We see below that it is required for an `Add` to always insert a fresh vertex with the same atom, but a different identifier.

How UNDO inverts operations

```
function UNDO(OpUndo(Add($v$)))
    return Delete($v$)
```

```
function UNDO(OpUndo(Delete($v$)))
    let $(a, (p, rid, t)) \leftarrow v$
    let $t' \leftarrow$ NOW( )
    let $v' \leftarrow (a, (p, rid, t'))$
    return Add($v'$)
```

With only the vertex to be deleted stored in `OpDeleteRemote`, it is not possible to fully invert it to a `OpAddRightRemote` as the left vertex (v_l) is unknown. However, the LSEQ scheme doesn't necessarily require a left vertex, so we may leave out the left vertex property of the `OpAddRightRemote` and just consider a one argument constructor taking the vertex to be inserted. So, for brevity here I will refer to `OpAddRightRemote`(v) as `Add`(v) and `OpDeleteRemote` as `Delete`(v). This means undo isn't supported when using RGA.

3.3 Application

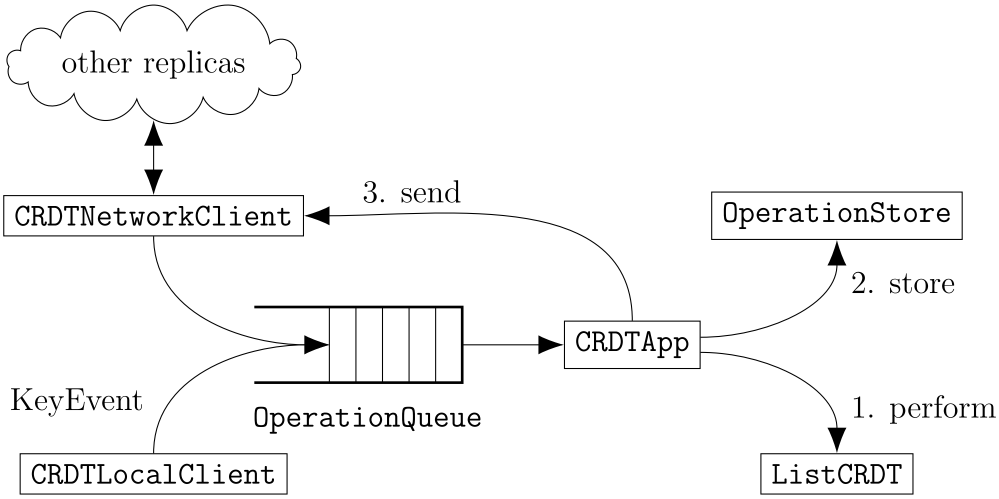


Figure 3.5: The layout of the application; arrows indicate the flow of operations. The numbers indicate the order in which `CRDTApp` does these actions

.

To build an application that uses the CRDT library required a few extra data structures to move around and store operations. The first problem to be dealt with is that operations will arrive asynchronously from both the network and any local GUI the application is connected to. It therefore makes sense to use the *producer-consumer* paradigm and have a queue of operations that the network and GUI feed into, and a separate thread pulls out of to perform them (two producers and one consumer).

`OperationQueue` wraps around Python's `Queue` class, a thread-safe, double-ended queue. It adds a `Semaphore` (from Python's `threading` module) for condition synchronisation, such that the consuming thread can, on seeing an empty queue, block until something is available instead of busy waiting. When a producing thread adds to the queue, it increments the semaphore, and a consumer decrements it on receiving. If the semaphore is 0 a decrement will block until there is an increment. The application's implementation of the `OperationQueue` will be referred to just as *opQueue*.

I then created a class `OperationStore` which just has a `dict` of `lists` (Python builtins) and exposes operations on it.

```
Methods supported by OperationStore
function addOp(key, op)              ▷ stores op in the list indexed by key
function getOpsForKey(key)           ▷ returns the list indexed by key
```

Since it was the case that multiple threads may be accessing instances of this simultaneously, and Python has no *synchronised* primitive like Java, I needed some way of having thread-safe access to the structure. This was achieved with a *synchronised* decorator[4] to decorate the necessary functions such that only one thread can access the object they are tied to at a time.

Recall that operation-based CRDTs require that if A *happens-before* B, A should be delivered (and hence here performed) before B. To provide this, each operation sent (any `RemoteOps`) has an *opID* (instance of `ClockID`), a pair of a timestamp and *rid* showing the operation's source replica. Since a causal order on events implies the order of their timestamps, but not the other way round, from two *opID*s we can only infer one operation *happens-before* the other from their timestamps if their *rid*s are the same.

As will be seen in section 3.4, the way operations are sent guarantees operations with the same source replica will be ordered correctly. However, there needed to be a way of dealing with ordering dependent operations from different source replicas whose causal order cannot be inferred from their *opID*s.

As shown in figure 3.6, it may be that an operation arrives before a causal predecessor over a network. On seeing a timestamp more than one greater than the last we have seen from any replica, we don't know if we have missed one. For example, if there was also an operation 1:C, then on seeing 2:B, replica C wouldn't know if there was a 1:A coming, because it could have been 1:C that incremented B's clock. To resolve this ambiguity, I attempt to perform operations, and if they reference vertices that don't exist yet (ie have broken causal order), they get *held back* and are performed when that vertex does exist.

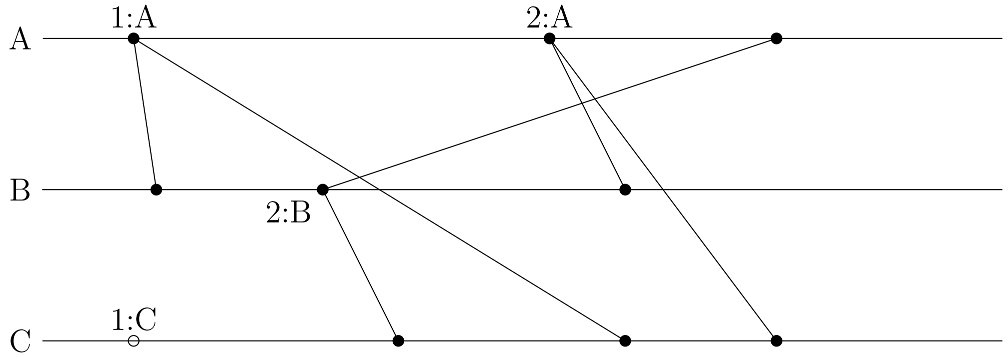


Figure 3.6: An example sending of operations with *opID*s 1:A, 2:A and 2:B. The latter arrives at C before its causal predecessor 1:A.

[4] https://github.com/openstack/deb-python-wrapt

To achieve this, there is an implementation of `OperationStore` called *heldBackOps*, on which ADDOP(*vertexID*,*op*) is called for when the operation *op* fails which references a vertex with identifier *vertexID* (if `OpAddRightRemote` this is the identifier of the left vertex v_l and for a `OpDeleteRemote` is the identifier of the vertex to be deleted). For example, an ADDRIGHT would fail (in RGA) if the left vertex didn't exist, and a delete would fail if the vertex to be deleted didn't exist. Now, after performing an operation with identifier *opID*, we call GETOPSFORKEY(*opID*) on *heldBackOps* and add any operations in the resulting list to the front of *opQueue* (called the *recovery* phase). This works because the *opID* of an `OpAddRightRemote` has the same timestamp and *rid* as the inserted vertex. So, on inserting the vertex, we free and perform any operations waiting on that vertex to exist.

Another structure needed is a way to keep track of which of whose operations a replica has performed, so that it may share operations with new replicas that come online to collaborate. Let this be known as *opStore*. This is an implementation of `OperationStore`, keeping lists of performed operations indexed in the `dict` by source replica. Operations in a list are kept in the order they were performed at this replica. So, each list will be ordered by the timestamps of the *opID*s as two operations with the same source replica cannot be concurrent.

As such, the `CRDTApp` has a thread which loops continuously over the following sequence:

- Pop from *opQueue*
- Perform the popped operation
- Store the operation that results from performing
- Send the operation to any other replicas connected
- Do *recovery*: check if any operations were waiting on this one to complete

Finally, in order to know who to send operations to (ie who you are currently collaborating with), there is a structure `ConnectedPeers` which wraps around Python's `dict` providing friendly methods for storing peer information and a thread-safe way to iterate over it. Applications each have an instance of this known as *connectedPeers*.

3.3.1 GUI

The first extension to the project I implemented was a simple graphical user interface (GUI) to allow quick identification of obvious errors in my code (helped even more by the use of Docker containers). This was achieved with the use of Tkinter[5], a Python interface to the Tk toolkit. Tk has a hierarchy of *widgets*, beginning with

[5] https://wiki.python.org/moin/TkInter

the *root*, which represent objects on the screen, where each child object is contained within its parent. Keyboard and mouse events are handled by a system of callbacks. A function may be bound to a specific event such that when that event occurs the function is called with the event information passed as a parameter.

With that in mind, I created a function to, given a keycode of a printable character, add a new `AddRightLocal` operation inserting that character to *opQueue*. If key was backspace, a `DeleteLocal` is created. This was bound to the *KeyDown* event so the relevant operations are performed when keys are pressed. If the Left (resp. Right) Arrow Key is pressed, code is executed which calls SHIFTCURSORLEFT(RIGHT) in ListCRDT. A reference to the latter function is passed to the object that contains the GUI code, `CRDTLocalClient`. These arrow keys could therefore easily be configured to traverse the document in some other way than just horizontally by one character. The keys to enact an undo or redo work by the same principle.

A Text widget from Tkinter is used to hold the CRDT's representation. This captures keyboard events and sends them to the aforementioned bound functions. It has a method for setting the position of its insertion cursor as well, which takes a line and column index. I decided to restrict the editor to only single-line text for ease of moving the cursor around (the only valid moves are to the next or previous vertices that aren't deleted) and so pressing *return* does not trigger an event as it is not a printable character.

The GUI needs to be updated every time an operation is performed, as all operations have some effect on the representation of the CRDT. To do this, the list of vertices is iterated over and for each vertex (a, id), $\textsc{str}(a)$ is appended to the output (if the vertex isn't deleted). Also, when the identifier held in the *cursor* is encountered, the number of atoms output so far determines the column number that the cursor should appear in the final output. This is needed for specifying the column index when the insertion cursor in the Text widget is set.

3.4 Networking

To exchange operations with each other, replicas need to communicate over some network. I chose TCP/IP as the communication mechanism to allow potentially global collaboration. TCP allows for reliable, inorder delivery of data, which specifically means operations from the same replica will be delivered in the order they were sent, a required property for op-based CRDTs.

To implement such a mechanism, I used Python's builtin `socket` library to create sockets which can be read from and written to. One socket listens for connections (the server socket) whilst the other actively connects to a listening socket (the client socket) and hence a channel is created between them. However, TCP is a stream-oriented protocol, and I needed some way to delimit separate operations.

Firstly, operations are sent by sending the actual `Op` objects. To serialize such objects into a format that can be sent, the `pickle` library[6] is used. The pickled data is sent prefixed by a 4 byte representation of its length. In this way, a receiver knows to first read 4 bytes from the stream (= *length*), then read a further *length* bytes to get the actual data (a technique called framing). It then unpickles this to get the `Op` object. The RECV function on the socket object to read from it takes a maximum number of bytes you wish to receive, and returns the actual number read. As there is no guarantee that you will get the number of bytes you were expecting all at once, I repeatedly call this method until *length* bytes of data are actually received.

In order to support pickling/unpickling, a Python object merely has to override the methods _GETSTATE_ and _SETSTATE_. In the former one must return a `dict` with values of the object's properties necessary to recover the object, and in the latter the properties are set with values from this `dict`.

As my project supports offline editing, when a replica comes online having performed operations offline, it is necessary to both share those operations with other replicas and receive any new operations from them. The operations needed by replica r_k is

$$\bigcup_{i \neq k} \{ops_{r_i}\} - ops_{r_k}$$

where ops_{r_k} are the operations r_k has either in *opStore*, *opQueue* or is currently performing. Any connections the replica makes are in different threads, so that requests for new operations to each replica are pipelined (all requests may be sent before even one response is received). This decision was made over waiting for each response to arrive as the waiting time is bounded only by the timeout of the local TCP implementation which was deemed to be too long.

To calculate this, each replica keeps a vector of `ClockID` objects (the vector clock) for each replica that it has seen an operation originating from. This is updated when *opQueue* receives a new operation. Each object's timestamp is the highest timestamp of all the operations it has seen originating from that object's replica. `OperationStore` has a method for taking a vector clock and outputting a list of operations stored that haven't been seen by that vector clock. That is, for each `ClockID` c in the vector, it finds, in the list of operations it has stored for the replica corresponding to that object, any operations o for which $\text{TIMESTAMP}(c) < \text{TIMESTAMP}(o)$, then combines those into one big list. Since *opStore*'s lists are kept sorted, one can use a variant of the binary search algorithm to find the smallest timestamp in the list greater than c's, then return that element and all subsequent ones in the list as the result. For example, given the vector clock 5:A | 4:B and the lists [1:A, 2:A, 4:A, 6:A, 7:A] and [3:B, 5:B], the procedure would return [6:A, 7:A, 5:B]. Importantly, this process preserves the ordering of the operations, so the operations from each replica are sent in increasing order of their timestamps.

[6] https://docs.python.org/3/library/pickle.html

Finally, in order to know to which peers to send operations to when they are performed, there is a class `ConnectedPeers`, which again wraps around a Python `dict` mapping peer addresses to their socket objects, and is completely threadsafe. So, when sending a new operation, you iterate over a copy of the dictionary's sockets and call SEND on each one.

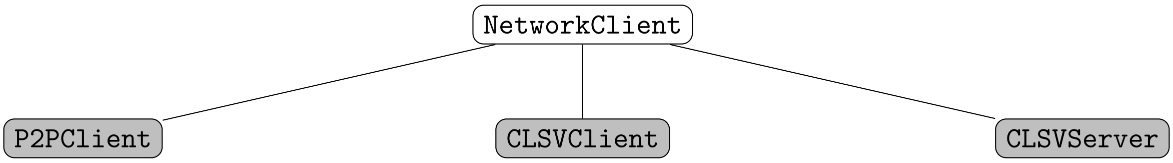


Figure 3.7: The networking classes used in the application. `NetworkClient` contains common code for sending and receiving data as described above.

3.4.1 Client-server Architecture

The first network architecture I implemented was one where there is a central server which all replicas connect to (it is not itself a replica). The server simply relays operations between replicas and stores them locally. Thus, any new replica only needs to send its vector clock to the server rather than all replicas separately. So, the server keeps an instance of `ConnectedPeers` and listens to all connected clients for new operations. On receiving an operation, it stores it locally and sends over all sockets in the `ConnectedPeers` instance.

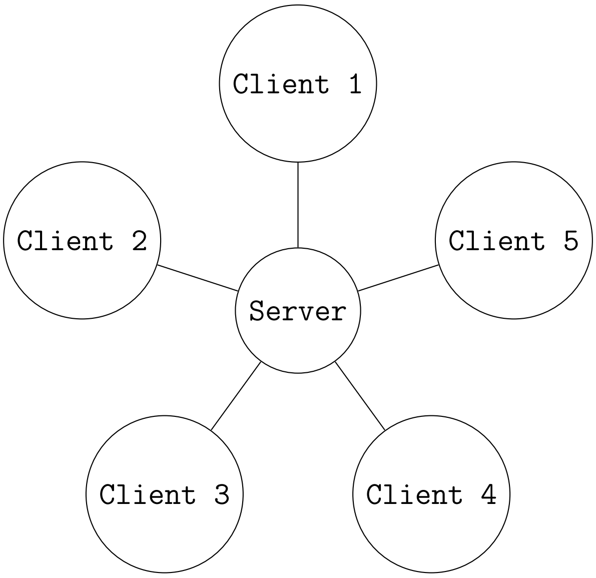


Figure 3.8: The standard client-server architecture with 5 clients

3.4.2 P2P Architecture

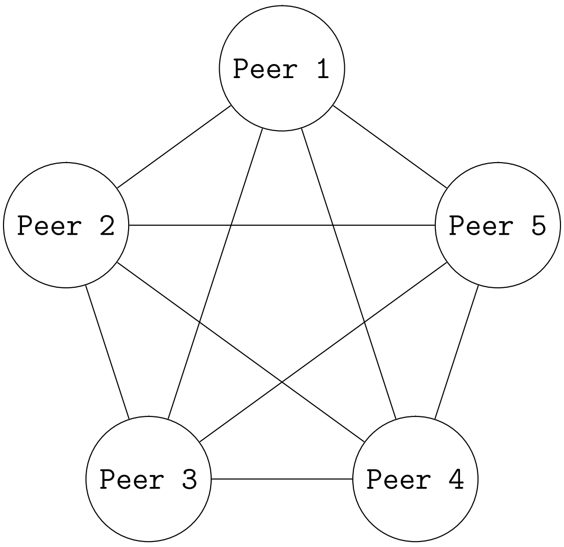


Figure 3.9: The standard P2P architecture with 5 clients

The P2P architecture (as shown in figure 3.9) consists of links from every node to every other node, making a complete graph. Therefore, it is necessary:

1. For each node to know how to reach every other node

2. For each node to listen for connections from as well as initiate connections to all other nodes

To deal with 1, a design decision was made to assume nodes who wish to collaborate know through some other channel the addresses of their collaborators. Another option considered was to have a centralized directory service which stores mappings of documents to node addresses. However, this mirrors the shortcomings of the client-server approach; namely that the directory is a single point of failure, and if compromised could reveal addresses people might want to keep private. Besides, to implement such a directory would require a new client to provide some identifier (such as a key) to the directory so they may be linked to a document. The identifier would have to be shared through some other channel anyway. Moreover, the Tor Hidden Service protocol deals with the issue of collaborators wishing their location to be hidden from the others.

For 2, it was necessary for each peer application to both have a server socket accepting incoming connections and to try and open a socket to every other peer. However, this would result in two connections for each link shown in the diagram, which is undesirable. Therefore, it is necessary to check in *connectedPeers*, and not complete a connection to an already connected peer. Unfortunately, this is still not enough, as due to the socket threads running simultaneously, both connections could make progress simultaneously. So, asymmetry must be introduced so that both ends don't act in the same way. Hence, as shown in figure 3.10, peers send their name, and on detecting multiple connections to the same peer, an incoming connection is allowed only if the other peer has a lower identifier than yours (a total ordering).

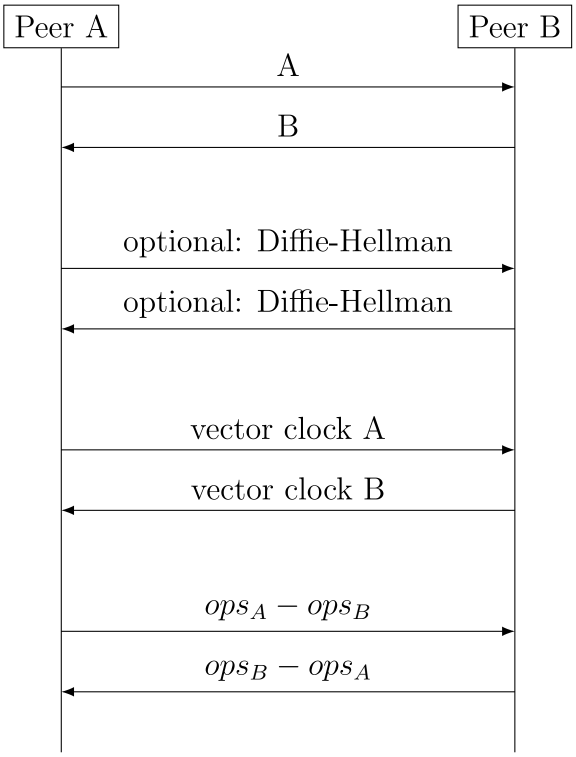


Figure 3.10: The protocol for peers connecting in the P2P architecture. After exchanging peer identifiers, they synchronize their operations.

The implementation of sockets I used meant they block until data is available. To be able to close both ends of a socket, I send a null byte from one end in its own frame then close that end. The receiving end then knows to close on receipt of a single null byte by itself.

3.5 Encryption

As the project has a theme of privacy, encrypting the channels between nodes seemed sensible. This was achieved by adding an optional Diffie-Hellman (DH) key exchange [7] to the protocol, hashing the computed shared secret to generate an AES key, then encrypting all further communications over that channel using the key and AES CCM mode[8], which is Counter mode for confidentiality together with a CBC-MAC for integrity. This is an authenticated encryption scheme that, provided it is used

[7] http://mathworld.wolfram.com/Diffie-HellmanProtocol.html

[8] https://en.wikipedia.org/wiki/CCM_mode

correctly, allows for confidentiality and integrity of all messages sent on the channel. Negotiating session keys for each channel means one can get *perfect forward secrecy*, meaning compromise of nodes doesn't reveal all past keys and thereby breaking confidentiality of all past messages. This use of asymmetric encryption (in DH) to bootstrap a symmetric encryption scheme is commonly used as symmetric encryption generally is faster.

3.6 Tor

The final extension added to the project was the use of the Tor network for communication in the P2P architecture. The idea is that each peer advertises a Hidden Service (HS), and other collaborators know its *onion address* (again through some other channel), so connect to it through Tor's HS protocol.

Stem, the Python library used for interacting with the Tor process, exposes a method createEphemeralHiddenService, which can be called to setup a HS on a machine. It can be given a private key from which the public key (and subsequently the *onion address*) for the service can be derived. Users need to keep a private key for each document they work on and provide one to the application on startup. 'Ephemeral' here refers to how the HS created is kept entirely in memory and does not touch the filesystem at all.

Once the Tor process is running, to connect to a hidden service, one simply routes traffic through the local SOCKS5 proxy server the process creates. To do this, the `PySocks` library[9] contains a socket object with the same interface as the Python's own `socket`, except with a setting to send traffic through a proxy server.

3.6.1 Client Authorization

Using Stem and the createEphemeralHiddenService call, basic authentication can be implemented by simply passing an extra parameter. Namely, a `dict` with names as keys. The value for each of these can be `None`, in which case when the call returns and the HS is created a `dict` is returned mapping those names to randomly generated cookies. Alternatively, previously generated cookies can be specified instead of passing `None`, in which case those are returned. The cookies are somehow distributed to other peers (through some other secure channel), then they use Stem to set their Tor process's `HidServAuth` parameter to, for each other peer c, a pair of c's onion address and the cookie for that peer. That is, telling Tor which cookie to use for each peer.

[9] https://github.com/Anorov/PySocks

Chapter 4

Evaluation

4.1 Success criteria

My original success criteria for this project were:

- To have a tested library for operation-based CRDTs which represent an ordered list of characters
- To have 2 versions of a library which a text editor application might use to collaboratively edit a document with other similar applications over a network. One which will use a central server to pass data, and another where data is sent directly between clients.
- To have a library which, when used by an application, provides collaborative editing functionality with sufficiently small latency between clients to be considered 'realtime'

During the course of the project, I realised that as well as providing sufficiently small latency, it was highly desirable for my library to have reasonable memory usage and reliability as well. I will now discuss to what extend these amended criteria were met.

4.2 Core Library

4.2.1 Unit tests

To have both continuous sanity-checking and more rigorous testing of how concurrency scenarios are handled, I compiled a suite of 50 unit tests as I developed the project. There were unit tests for some basic functions such as the operations for the helper structures, which checked for example that insertion and deletion worked as expected in the given scenarios. The tests focused on concurrency involved applying some events in different specified causal orders and checking they achieved the same result.

4.2.2 CRDT Operation Latency

To evaluate the performance of my library, I timed how long it took the `OperationQueue` to be ready to take from again after popping an operation, which corresponds to how long the operation took to perform and store. I plotted this against the length of the document at that point. For insertion, I applied 10,000 `AddRightLocal` operations sequentially to an empty list, and timed each one (figure 4.1). For deletion, I first inserted 10,000 vertices into the list, then applied 10,000 `DeleteLocal` operations, then reversed the results, so that the first data point is deleting from a one element list (figure 4.2). This procedure was done for the three implementations of `BaseOrderedList` I created: `LLOrderedList`, `LSEQOrderedList` and `ArrOrderedList`.

I broke the latency up into four parts:

- *Insertion* is the time it takes to apply the update to the CRDT
- *Store_op* is the time taken to put the operations in the *opStore*
- *Network_send* is the time taken to iterate over all connected peers and send the operation to them (for simplicity in the measurements there were no others)
- *Recovery* is the time spent checking *heldBackOps* to see if any operations were waiting on this one to finish and if so adding them to the front of the queue (in the measurements there would be no such operations)

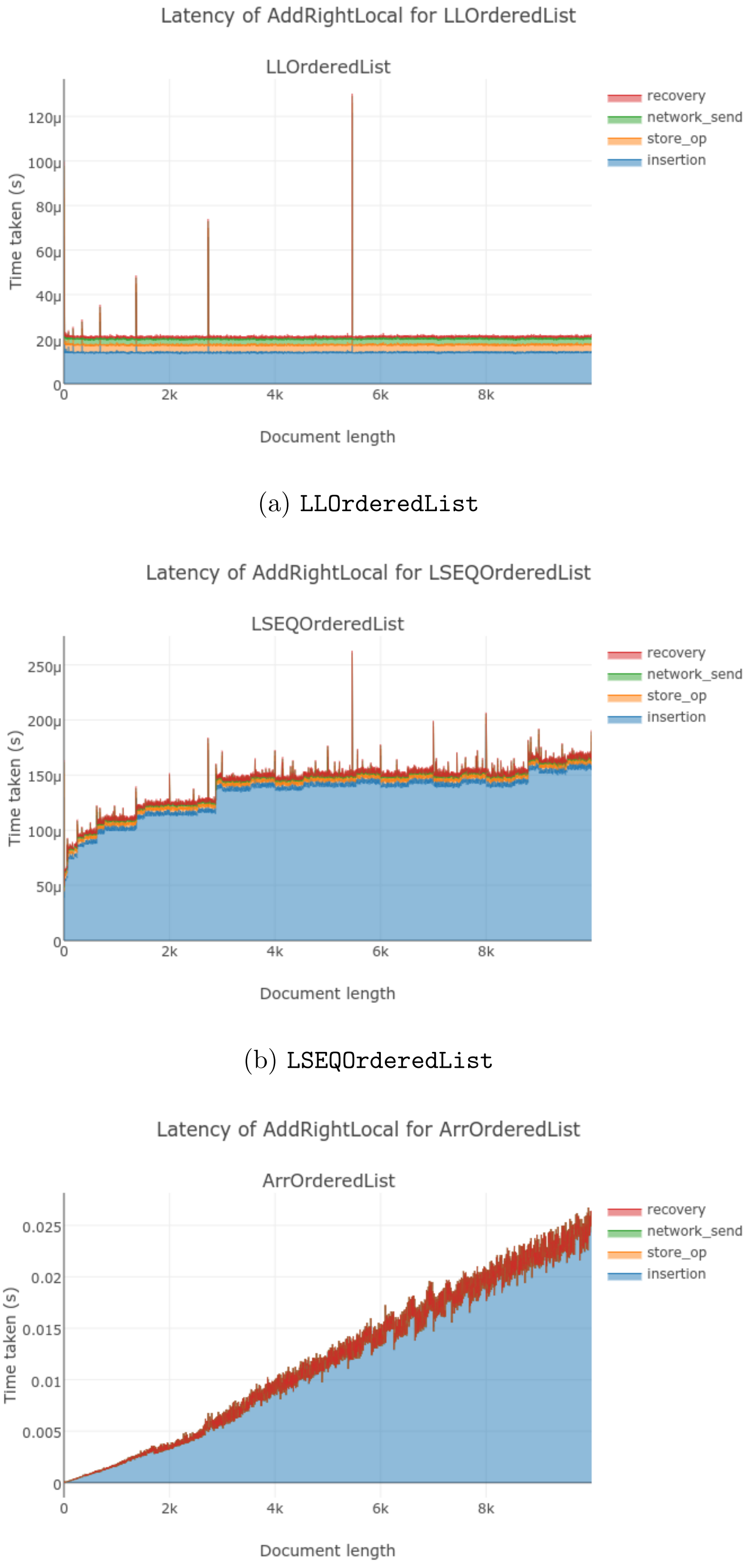
The time for deriving the new state after each operation has been omitted here as it is the same for all implementations, and it dominates the cost for the first two. Since we have to scan through the vertices in order to calculate the column in which the cursor should appear, this is a linear overhead in the size of the list.

For `LLOrderedList`, we see that both times are independent of the length of the list. This is because lookup in the dictionary that maps identifiers to linked list nodes is a constant time operation, meaning so too are insertion and deletion in the linked list. The large spikes occurring for insertions are the resizing of Python's builtin dictionary. When this happens, the size is doubled, which explains why both the spacing between the spikes and their height doubles each time. This isn't seen in the deletion case, because at the point of the first measurement, 10,000 vertices have already been inserted, so the dictionary is already big enough.

For `LSEQOrderedList`, the complex data structure used to house the linked list nodes offers roughly logarithmic-time lookup, so this dominates both insertion and deletion cost. As such, the graphs exhibit a roughly logarithmic shape. For insertion, there are visible *chunks* of time when the cost is roughly constant, and the chunk boundaries are where some resizing is happening. The logarithmic shape is shown in more detail in figure 4.3b, where roughly a straight line emerges with a logarithmic x-axis.

For `ArrOrderedList`, since finding a node from its identifier involves a linear search on all the vertices, we see a linear relationship between the number of vertices and the time it takes to insert or delete.

The graphs nicely show how the asymptotic complexities of performing operations for different implementations - $O(1)$, $O(\log n)$ and $O(n)$ respectively - affect real-world performance. Additionally, we can see in figure 4.3 in both implementations that the resizing operation completely dominates for larger documents, but at a length of around 45,000 characters, the resizing latency is still only around 1ms. In order to get up to a resizing latency of 0.1s (on the edge of being noticeable) requires approximately 7 more resizes ($2^7 = 128$). This happens every time the size doubles, so this time would be reached at $45,000 * 128 \approx 4$ million characters. As of Dec 2016, the largest wikipedia article was around 1.1MB in size, so at one byte per character this would be slightly larger than the largest Wikipedia article.

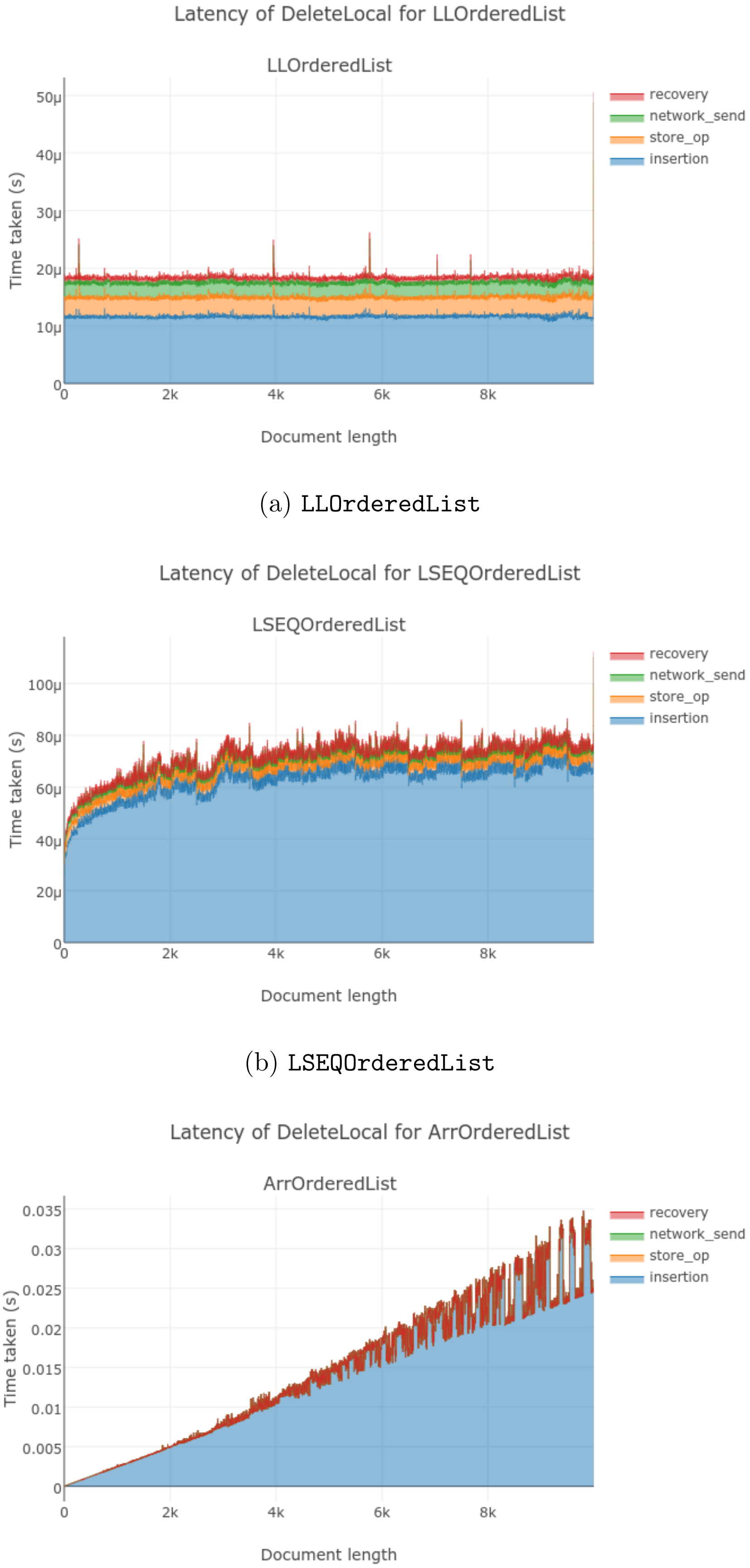


(a) `LLOrderedList`

(b) `LSEQOrderedList`

(c) `ArrOrderedList`

Figure 4.1: AddRightLocal latencies plotted against the length of the document

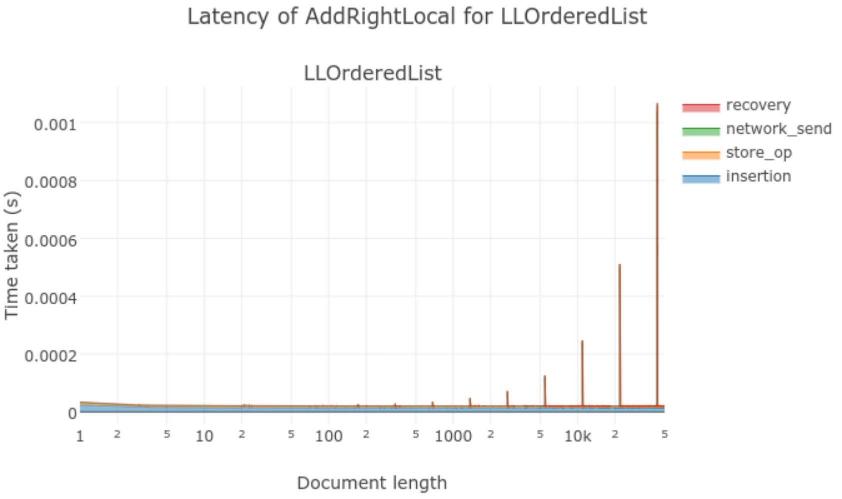


(a) `LLOrderedList`

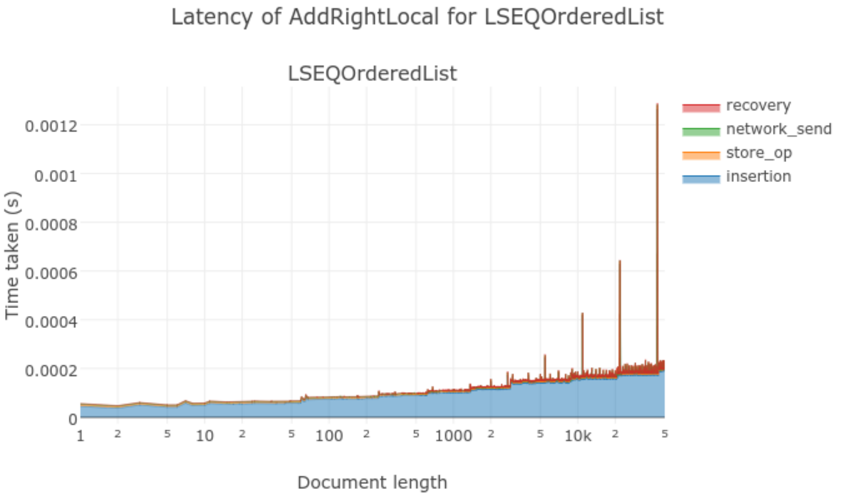
(b) `LSEQOrderedList`

(c) `ArrOrderedList`

Figure 4.2: DeleteLocal latencies plotted against the length of the document



(a) A log-scale plot of insertion time for `LLOrderedList`



(b) A log-scale plot of insertion time for `LSEQOrderedList`

Figure 4.3: Latency for 50,000 AddRightLocal operations on logarithmic scales

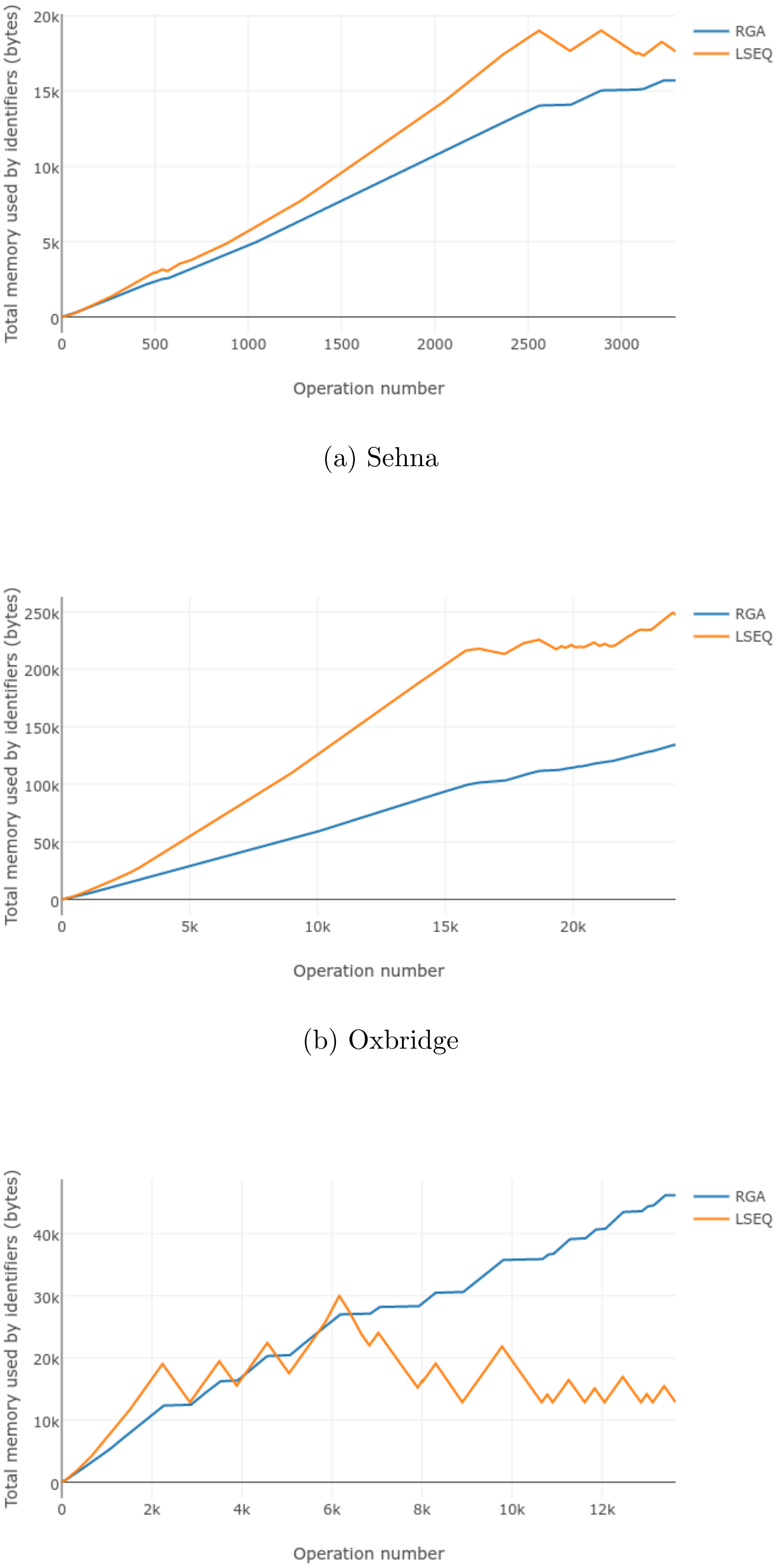
4.2.3 Memory usage

Recall that LSEQ allows vertices to be freed from memory (at the cost of more memory per vertex on average), whereas RGA allows a fixed per-vertex cost, but disallows reclaiming from memory.

To see the real-world memory usage of the different approaches, I replicated an experiment from earlier work [6, 8] simulating editing with the revision history of different Wikipedia pages. I scraped the revisions from the pages histories, then performed a character-level *diff* between consecutive versions to produce operations to transform the first version into the last that could be applied to my CRDTs.

Figure 4.4 shows the results of this. As Figures 4.4a and 4.4b show, for documents with mostly insertions, the overhead of variable size identifiers outweighs not being able to delete elements. However, for documents with more deletions performed, such as figure 4.4c (a noticeboard reporting vandalous users which gets cleared when they are dealt with), we can see the non-decreasing memory usage by RGA is significantly higher. As such, choosing a scheme is a trade-off based on the expected proportion of deletions that will happen.

In the Oxbridge example, we see that for roughly 20,000 characters we are using around 200kB, which is 10 bytes per character for each identifier on average. Recalling the largest Wikipedia article to be $\sim$1.1MB in size and assuming all text and 1 byte per character, my scheme would use $11 + 1.1 = 12.1$ MB to store the vertices (the largest contributor), which is a reasonable figure for an application nowadays.



(a) Sehna

(b) Oxbridge

(c) Administrator intervention against vandalism

Figure 4.4: Memory usage of my implementations of RGA (`LLOrderedList`) and LSEQ when simulating editing on Wikipedia revision history.

4.2.4 Undo correctness (proof)

To show my approach to implement undo is correct, I wish to prove the following:

$$\forall l, op, n \in \mathbb{N}, others.\ length(others) = n.\ p_{undo(op)} \circ p_{others} \circ p_{op}(l) \equiv_l p_{others}(l)$$

Where $p_x(l)$ returns the new state l' obtained by performing operation x on list l, and *others* is a list of operations, so that p_{others} is shorthand for $p_{others_0} \circ p_{others_1} \circ ... \circ p_{others_n}$. Let *op* be either `Add(`v`)` or `Delete(`v`)` for some vertex v wlog (let these be shorthand for `CRDTOpAddRightRemote`(v) and `CRDTOpDeleteRemote`(v) respectively).

In other words, I will show that if a replica r originates an operation *op* then performs any number of operations from other replicas, and then undoes *op* (as the most recent operation originating from r is undone), it is as if *op* was never performed in the first place. I will prove this by induction on the length of the list *others*, $n \in \mathbb{N}$.

Proof Base case $n = 0$:

Want to show:

$$\forall l, op.\ p_{undo(op)} \circ p_{op}(l) \equiv_l l$$

By cases:

Case $op =$ `Add`(v):

Then $undo(op) =$ `Delete`(v). Before the add, $v \notin l$ by assumption that adds are fresh. Performing the `Add` immediately followed by the `Delete` inserts then removes v from the list without modifying any other vertices. Therefore, in the resulting state after both operations l', $v \notin l'$ and $l \equiv_l l'$.

Case $op =$ `Delete`(v):

Then $undo(op) =$ `Add`(v') where v' differs from v only in timestamp. The `Delete` followed by the `Add` removes v and inserts v'. For v's position and *rid*, there is no other vertex in s already as vertices added by the same replica will be allocated different positions always (see definition of ALLOC). So, $\nexists v''.\ v <_v v'' <_v v'$. Thus, by the definition of $<_v$, v' is inserted in the 'same place' as v was, and since their atoms are the same, the resulting state l' is equivalent to l under $\equiv_l$.

Induction Hypothesis (IH):

$$\forall l, op, others.length(others) = k.\ p_{undo(op)} \circ p_{others} \circ p_{op}(l) \equiv_l p_{others}(l)$$

Inductive Step:

Assume IH. Want to show

$$\forall l, op, x, others.\ length(others) = k.\ p_{undo(op)} \circ p_x \circ p_{others} \circ p_{op}(l) \equiv_l p_x \circ p_{others}(l)$$

or alternatively, using IH and with the same quantifications:

$$p_{undo(op)} \circ p_x \circ p_{others} \circ p_{op}(l) \equiv_l p_x \circ p_{undo(op)} \circ p_{others} \circ p_{op}(l)$$

This is shown by cases:

Let x be `Add(`v'`)` or `Delete(`v'`)`.

Case $v \neq v'$:

Insertion and deletion only modify the vertex they reference in a list, so operations referencing different elements trivially commute.

Case $v = v'$:

Then $x =$ `Delete`(v) since adds are always fresh. Now we again case split on the type of *op*:

Case $op =$ `Add`(v):

Then $undo(op) =$ `Delete`$(v) = x$. Since performing a `Delete` does nothing if the vertex is already deleted, $p_{undo(op)} \circ p_x = p_x = p_x \circ p_x = p_x \circ p_{undo(op)}$ (in this context). So, $undo(op)$ and x commute and we are done.

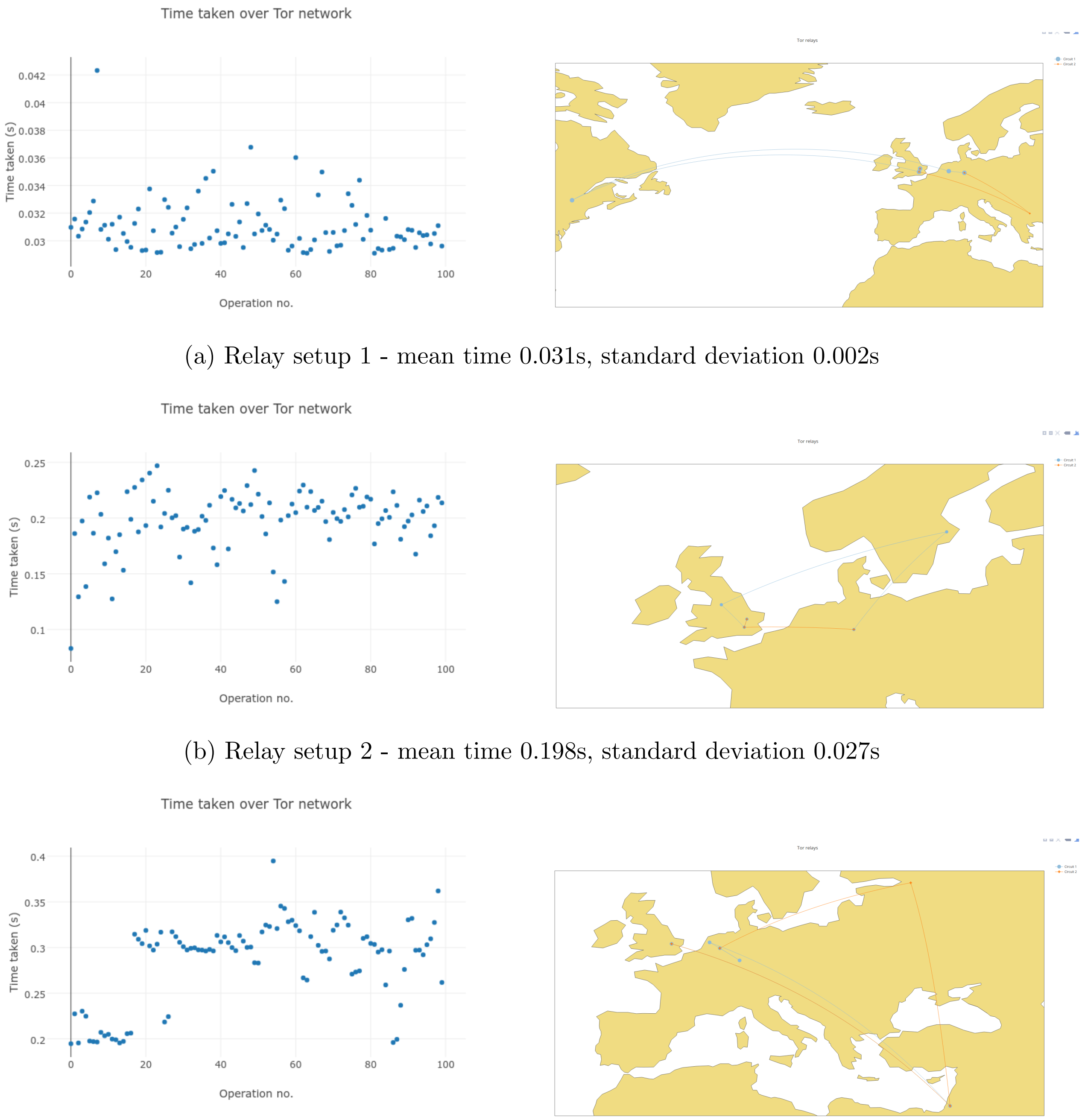
Case $op =$ `Delete`(v):

Then $op = x$ and since deletes are idempotent, p_x is the identity function on lists, so trivially commutes with $undo(op)$ and we are done.

4.3 Networking

4.3.1 Network Latency (show latency profiles of different Tor circuit setups)

To see how quickly operations can be delivered over the Tor network. I setup a hidden service and sent 100 operations to it from the same machine. Therefore, messages would go out to the Tor network and back in, and I would record the time of departure and arrival at my machine. Three different profiles are shown in figure 4.5 corresponding to three different choices of circuits. The maps show the locations of the relays used in the two circuits built to the rendezvous point. As the figures show, the time spent in the network is very variable and highly dependent on the chosen locations of the relays.



(a) Relay setup 1 - mean time 0.031s, standard deviation 0.002s

(b) Relay setup 2 - mean time 0.198s, standard deviation 0.027s

(c) Relay setup 3 - mean time 0.285s, standard deviation 0.046s

Figure 4.5: Locations of Tor relays used in a Hidden Service circuit and the latency profiles for sending 100 operations through them

So, the time between typing a character and noticing another replica deleting that character involves:

- Performing the insertion locally - 0.1s in a large document
- Sending the operation - Say around 0.3s
- The other replica performing the operation - 0.1s
- The other replica's person noticing the change and pressing a key 0.3s for human reaction time
- Their replica performing their operation - 0.1s
- Sending their operation to you - 0.3s
- Performing their operation locally - 0.1s
- You noticing their operation - 0.3s

Giving a total time of around 1.6s which I deemed to be reasonable.

4.3.2 Security

A man-in-the-middle (MITM) attack describes a situation where an adversary intercepts communications along a channel and relays there own messages to each endpoint. Without the use of basic authentication for Hidden Services, my library is vulnerable to such an attack.

To illustrate the attack, consider two peers A and B wanting to edit a document, and an adversary C who can read and write to the channel between them. If C finds out the onion address for A and B by some means, it can connect to A and B. A and B would just assume they are connected to each other, but the key point is that the authentication is unidirectional, from A to C and B to C, but not the other way round. Thus, C can intercept and modify the document any way it wants.

Authentication for Hidden Services, then, solves this by requiring C to authenticate itself to both A and B, by providing their specified cookies. The assumption is that these cookies are exchanged over a secure channel, such that C cannot get them. However, a flaw with this system is that there is no simple revocation available. Once C has a cookie, it is as authentic as any legitimate peer, and my implementation has no way to easily redistribute cookies.

4.3.3 Reliability (tbd)

Once a connection over Tor is established, there is nothing in the Tor specification that restricts its lifetime. Fresh circuits are by default chosen every 10 minutes, but only for new connections. Therefore, a connection opened by one replica to another would only be closed either due to one party explicitly disconnecting or due to some network event that is outside the control of the application, for example one of the relays going offline. I set up a scenario where one replica generates and sends an operation every minute to send to another running on my machine (though going through Tor), and the connection ran successfully for 23 hours before my laptop disconnected because it had to be moved, however as discussed there is no theoretical limit.

4.3.4 Network Architecture

Running a P2P application instead of the client-server approach has several advantages. For example, servers are inherently bottlenecks (and single points of failure), and provide the privacy problems Google Drive poses. Decentralizing means one can potentially be completely anonymous (providing the secondary channels used to distribute onion addresses etc. preserve anonymity) as described in this project when using Tor Hidden Services, and all data completely hidden from anyone other than the specified collaborators. Moreover, the P2P mode adds no infrastructure, so its availability simply relies on the availability of the Internet and the Tor network upon which it relies.

However, the fully connected nature of the P2P approach means for k connected collaborators, a new peer coming online means potentially the entire causal history of the document is sent to the new peer k times (each peer may have a different set of operations so all must be queried). This is obviously wasteful as the client-server approach gives one authoritative answer.

4.3.5 TODO REMOVE

!!!
TODO
!!!

- change legend for deletion not to say insertion
- redo deletion measurements - Arr quite noisy

Chapter 5

Conclusions

Overall, the project was a success. I implemented a library for a P2P collaborative editing application using CRDTs, then showed by building a minimal user interface that it can be utilised properly. I also implemented a number of other extensions to my original proposal, including encryption, allowing communication over the Tor network, a contrasting data structure for the CRDT, and an undo-redo feature. I then showed that the application displayed reasonable memory usage and latency, so was suitable for use in a desktop application. My conclusion on reliability is somewhat weaker, but it seems that my work would not be responsible for a communication failure, though unfortunately I didn't have time to include any retrying logic over TCPs own.

In hindsight, the domain of the project was quite broad; I touched on areas of distributed systems, networking and security. It would have been nice to be able to focus more in-depth in one of these areas, for example looking at the different CRDT schemes more closely. That being said, I was able to go into sufficient detail to understand and implement the LSEQ scheme, which was the topic of a recent research paper, though I would have liked to explore its allocation strategies more, and see if I could come up with one of my own that performed better than their random-choice strategy.

There are a number of future directions this project could be taken in. One would be to focus on the CRDTs, and look at string-level editing (ie an atom is a sequence of characters, rather than just a single one), where strings can be dynamically split up into smaller strings. This approach could save a lot in terms of memory usage. Alternatively, more work could be put into the networking side, and look at other ways of connecting peers together, to prevent the need for a fully connected graph, for example randomly dropping some of the n^2 links and allowing partitions to occur. Also, work could be done to cryptographically sign operations sent over a network, so that the source of the operation could be verified.

Bibliography

[1] L. Lamport, “Time, clocks and the ordering of events in a distributed system,” pp. 558–565, July 1978.

[2] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[3] M. Kleppmann, “A critique of the CAP theorem,” *CoRR*, vol. abs/1509.05393, 2015.

[4] S. Weiss, P. Urso, and P. Molli, “Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks,” in *Distributed Computing Systems, 2009. ICDCS’09. 29th IEEE International Conference on*, pp. 404–412, IEEE, 2009.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, *A comprehensive study of convergent and commutative replicated data types.* PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[6] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils, “Lseq: an adaptive structure for sequences in distributed collaborative editing,” in *Proceedings of the 2013 ACM symposium on Document engineering*, pp. 37–46, ACM, 2013.

[7] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils, “Concurrency effects over variable-size identifiers in distributed collaborative editing,” in *Document Changes: Modeling, Detection, Storage and Visualization*, vol. 1008, pp. 0–7, 2013.

[8] S. Weiss, P. Urso, and P. Molli, “Logoot-undo: Distributed collaborative editing system on p2p networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 1162–1174, Aug 2010.