Chapter 1

Preparation

1.1 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 [**?**] 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 [**?**], 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.

1.2 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 UPDATE: $\mathcal{O} \times \mathcal{S} \to \mathcal{S}$ which applies operations locally, then distributes the new state to other replicas. On receipt of the new state, another replica applies the function MERGE: $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$ 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* [**?**] 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:

$$\text{DOWNSTREAM}(o_1, \text{DOWNSTREAM}(o_2, s)) \equiv \text{DOWNSTREAM}(o_2, \text{DOWNSTREAM}(o_1, s))$$

So, there are advantages and disadvantages of each approach. One 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.

1.2.1 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 [**?**]. 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.

1.2.2 Tombstoning Approach

One approach, called the Replicated Growable Array (RGA) is described in [**?**]. 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.

It supports the following operations:

RGA

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

Require: state $s = \langle ..., v_1, v_l, v_2, ... \rangle$
1: **function** ADDRIGHT(v_l, a)
2: atSOURCE:
3: $t \leftarrow$ NOW()
4: $v \leftarrow (a, (t, rid))$
5: DOWNSTREAM:
6: $l \leftarrow v_l$
7: $r \leftarrow$ SUCCESSOR(v_l)
8: **while** IDENTIFER(v) $<$ IDENTIFIER(r) **do**
9: $l, \ r \leftarrow r, \ $ SUCCESSOR(r)
10: $s \leftarrow \langle ..., l, v, r, ... \rangle$

Ensure: $v \in s \ \wedge \ v_l <_v v \ \wedge \ \forall v'. \ v_l <_v v' <_v v.$ IDENTIFIER(v') $<$ IDENTIFIER(v)

Require: state $s = \langle ..., v_l, v, v_r, ... \rangle$
1: **function** DELETE(v)
2: DOWNSTREAM:
3: **if** $v \in s$ **then**
4: 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.

1.2.3 Variable-size Identifier Approach

Whereas vertex identifiers consist of a single timestamp and a replica identifier, a different system (used in Logoot [**?**]) 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, $\mathcal{ID} = \mathbb{N}^* \times \mathcal{R} \times \mathbb{N}$. The positional identifier is a sequence of numbers representing a path in a tree.

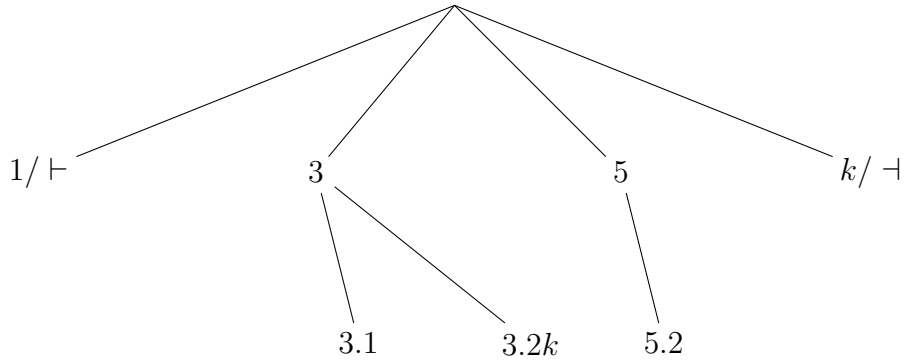


Figure 1.1: The tree-like structure of identifiers. Each number in the position represents a location to the right of its parent one level down in the tree. The outer two nodes at the top level are the positions of the start and end vertex, $\vdash$ and $\dashv$.

Define the ordering of the vertices $<_v$ by the total ordering on their identifiers $\prec$ from [**?**]: 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,

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

And

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

I use the base-doubling strategy from [**?**] so that the there are k 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: $v \in s$

1: **function** SUCCESSOR(v)
2: **return** $\min \{v' \mid v' \in s \ \wedge \ v <_v v'\}$

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

1: **function** ALLOC(p_l) ▷ Logoot's *boundary* strategy
2: $p_r \leftarrow$ SUCCESSOR(p_l)
3: $depth \leftarrow 0$
4: $interval \leftarrow 0$
5: **while** $interval < 1$ **do** ▷ Find a gap to insert in
6: $depth \leftarrow depth + 1$
7: $interval = \text{PREFIX}(p_r, depth) - \text{PREFIX}(p_l, depth) - 1$
8: $step \leftarrow \min(boundary, interval)$
9: $offset \in_R [0, step]$
10: **return** PREFIX($p_l, depth$) + $offset$

1: **function** ADDRIGHT(v_l, a)
2: atSOURCE
3: $t \leftarrow$ NOW( )
4: $newPosition \leftarrow$ALLOC($v_l.id.p$)
5: $v \leftarrow (a, (newPosition, rid, t)$
6: DOWNSTREAM
7: $s \leftarrow s \cup \{v\}$

Ensure: state $s = \langle ..., v_l, v, v_r, ... \rangle$ such that $v_l <_v v <_v v_r$

Require: state $s = \langle ..., v_l, v, v_r, ... \rangle \vee v \notin s$

1: **function** DELETE(v)
2: **if** $v \in s$ **then**
3: $s \leftarrow \langle ..., v_l, v_r, ... \rangle$

Ensure: $v \notin s$

A proposed improvement to this scheme is called LSEQ [**?**]. 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,

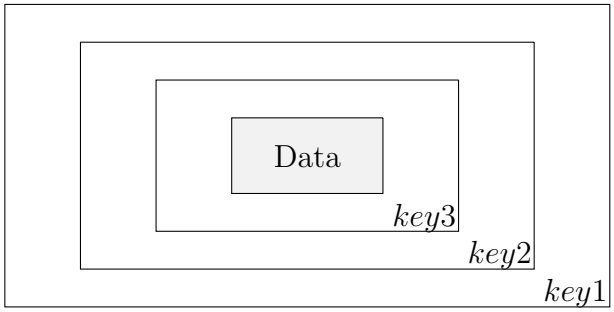
but few between the new one and its predecessor. The converse is true when you insert close to the right. To save memory, a good allocation scheme would allocate positions at the smallest depths possible based on future knowledge of what will be inserted, but this obviously isn't possible, so this scheme chooses randomly in the hope of doing well. I implemented LSEQ to be able to contrast it with RGA.

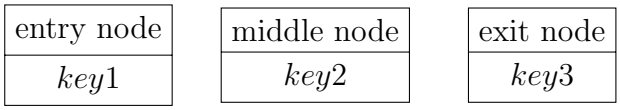
1.3 Tor

1.3.1 Onion routing

The Tor [**?**] 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.





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.

1.3.2 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.

1.3.3 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.

1.4 Project Development

1.4.1 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.

1.4.2 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 [**?**] Python library, which provides convenient API calls to use hidden services. Other minor libraries used are described in the next section.

1.4.3 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.