Lecture Notes on Leader-based Sequence Paxos

An understandable sequence consensus algorithm

Seif Haridi, Lars Kroll, and Paris Carbone

KTH Royal Institute of Technology, Stockholm, Sweden
{haridi, lkroll, parisc}@kth.se
http://www.kth.se

Abstract. Agreement among a set of processes and in the presence of partial failures is one of the fundamental problems of distributed systems. In the most general case, many decisions must be agreed upon over the lifetime of a system with dynamically changing membership. Such a sequence of decisions represents a distributed log, and can form the underlying abstraction for driving a replicated state machine. While this abstraction is at the core of many systems with strong consistency requirements, algorithms that achieve such sequence consensus are often poorly understood by developers and have presented a significant challenge to many students of distributed systems. In these lecture notes we present a complete and practical Paxos-based algorithm for reconfigurable sequence consensus in the fail-recovery model, and a clear path of simple step-by-step transformations to it from the basic Paxos algorithm.

1 Introduction

Agreement among a set of processes one of the fundamental problems of distributed systems. The challenges arise mostly from the possibility of *partial failures* that differentiate distributed programming from parallel programming. Under such conditions, not only must algorithms preserve safety guarantees despite failing nodes, but over the lifetime of a real system, the involved nodes must change to compensate for failures. Thus, in the most general case, not one, but many decisions must be agreed upon by a dynamically changing set of processes. A sequence of such agreed-upon decisions forms a *distributed log*, and often forms the basis upon which we build abstractions such as a *replicated state machine* (RSM). RSMs can be used as replicated databases [5], lock services [1], or configuration management services [4], for example.

While this replicated log abstraction is at the core of many such systems with strong consistency requirements, the actual algorithms that achieve what we call *sequence consensus* are often poorly understood by developers, and have presented a significant challenge to many students (and teachers) of distributed systems over the years.

In these notes we will incrementally describe an algorithm for reconfigurable sequence consensus in the fail-recovery model [2]. Our algorithm is based on the

well-known Paxos algorithm by Leslie Lamport [8], which we will present as a starting point in section 2. In section 3 we will describe the sequence consensus abstraction and a first simple algorithm to implement it. To improve our implementation, we will first take a detour into leader election in section 4, before using section 5 to reduce the communication cost and the memory footprint of our algorithm. Once we have a working and efficient algorithm in the *fail-stop* model, we will extend our implementation to function in the fail-recovery model in section 6, and particularly describe how to deal with (TCP) link session loss. In section 7 we extend our algorithm to allow the introduction of new processes, by describing how to move from one *system configuration* to the next. As the algorithm uses some state with unbounded growth at this point, we discuss garbage collection mechanisms in section 8, before discussing literature and concluding in sections 9 and 10 respectively.

2 Paxos

Given a set of processes in a partially-synchronous system model, i.e. an asynchronous system with stable periods of "sufficient" length, we wish a single value v to be agreed upon. That is, all processes should "decide" on the same value, such that the following properties hold:

UC1 (Validity) Only proposed values may be decided.
UC2 (Uniform Agreement) No two processes decide different values.
UC3 (Integrity) Each process can decide a value at most once.
UC4 (Termination) Every correct process eventually decides a value.

As part of the model we are given a channel abstraction that allows message duplication, losses, and out-of-order delivery. On top of the $\diamond P$ failure detection abstraction, we use an eventual leader election abstraction Ω. The Paxos algorithm provides the **UC** properties by using Ω's leader to impose a value to be decided. The algorithm guarantees safety during unstable periods and Ω provides liveness during a stable period. In Paxos each process plays one or more, quite often all, of the following roles:

Proposer Wants a particular proposed value to be decided.
Acceptor Acknowledges acceptance of proposed values.
Learner Decides based on acceptance of values.

In the typical majority quorum setup, a *proposer* tries to get a majority of *acceptors* to accept its proposal v. If a proposal has a majority of acceptors, then it is called *chosen* and the *learners* will decide it, once they discover that this is the case. Other quorum variants than majority have also been proposed for use with Paxos, and can improve performance at the cost of resilience. For brevity we will only discuss majority quorums in this article.

The process described at a high level above, is split into two phases. Multiple instance of each phase can run concurrently during any execution. Many of these instances may abort, until eventually a single v is decided.

Prepare Phase A *proposer* starts by picking a unique sequence number n and sending a message $\langle \text{PREPARE} \mid n \rangle$ to all acceptors. Upon receiving such a message an *acceptor* will either promise to not accept any proposal with a sequence number $n' < n$ or ignore/refuse the prepare, if it has already promised the same to someone with a higher n. If it promises, it will reply with a message $\langle \text{PROMISE} \mid n', v' \rangle$, where v' is the highest numbered proposal it has accepted so far (if any) and $n' < n$ its proposal number. The *proposer* collects all the promises it receives until it has a majority.

Accept Phase Once a *proposer* has collected a majority S of promises, it picks the highest numbered value v in S – or whatever value it wishes to propose, if there are no values in S – and sends a message $\langle \text{ACCEPT} \mid n, v \rangle$ to all acceptors. When an *acceptor* receives such a message, it replies with a simple $\langle \text{ACK} \rangle$ message, unless it has issued a higher numbered promise in the meantime, in which case it will reject with a $\langle \text{NACK} \rangle$. If the *learners* are separate processes from the *proposer* the $\langle \text{ACK} \rangle$ messages will need to be broadcast, otherwise they can just be sent to the relevant *proposer* acting in both roles. In either case, once a majority of acks is collected v can be decided, usually via broadcasting $\langle \text{DECIDE} \mid v \rangle$. If nacks make a majority impossible, the procedure must be aborted and started over.

During unstable periods the algorithm is guaranteed to satisfy its safety conditions (**UC1**-**UC3**), while termination is guaranteed only if the stable period is long enough for a solo proposer to perform the prepare and accept phases with no contention.

Fail-recovery In order to work in the *fail-recovery* model, acceptors have to commit some of their state to stable storage and restore it during recovery. Concretely acceptors need to store the *highest proposal* (n', v') they accepted and the *highest sequence number* n they promised.

Optimisations All $\langle \text{NACK} \rangle$ messages above are technically optimisations, as they can be replaced with timeouts on the proposer side.
In addition to the necessary rejection of any accepts with $n < m$ where the acceptor previously promised m, there are a number of optimisations that cause earlier aborts and thus waste less time on attempts already doomed to fail.

a) Reject $\langle \text{PREPARE} \mid n \rangle$ if answered $\langle \text{PREPARE} \mid m \rangle$ with $m > n$.
b) Reject $\langle \text{ACCEPT} \mid n, v \rangle$ if answered $\langle \text{ACCEPT} \mid m, u \rangle$ with $m > n$.
c) Reject $\langle \text{PREPARE} \mid n \rangle$ if answered $\langle \text{ACCEPT} \mid m, u \rangle$ with $m > n$.

Additionally, the algorithm should ignore any old messages for proposals that already got a majority.
Since a value "chosen" will always be decided in any higher round, a proposer can also skip the *accept phase* if a majority of acceptors return the same v in the *prepare phase*.

Algorithm Algorithm 1 shows the *fail-stop* version of Paxos described above with some of the optimisations. For simplicity of presentation the algorithm uses Perfect Links and no Ω and there are thus cases where it would never terminate, but end up in a race condition. Augmentation for Fair-loss Links and Ω is straight forward, if somewhat tedious, though. With Fair-loss Links messages can be lost and duplicated, thus augmentation requires resending, duplicate filtering, and also incrementing timeouts while waiting for majorities to avoid getting stuck. For Ω the required change is simply to only allow proposals while being leader. As Ω guarantees to eventually have a single leader, the race condition from before is thus avoided.

Algorithm 1: Abortable Paxos – Proposer

Implements: Uniform Consensus
Requires: Perfect Link
Algorithm:

A ; /* set of acceptors */
L ; /* set of learners */
$s \leftarrow 0$; /* local sequence number */
$(n_p, v_p) \leftarrow (\bot, \bot)$; /* unique round number and value */
$promises \leftarrow \emptyset$;
$acks \leftarrow 0$;

Upon $\langle$PROPOSE $\mid v\rangle$
  $s \leftarrow s + 1$
  $n_p \leftarrow$ UNIQUE(s) ; /* use pid to make n globally unique */
  $v_p \leftarrow v$
  $promises \leftarrow \emptyset$;
  $acks \leftarrow 0$;
  foreach $a \in A$ **do**
    send $\langle$PREPARE $\mid n_p\rangle$ **to** a;

Upon $\langle$PROMISE $\mid n, n', v'\rangle$ **from** a **s.t.** $n = n_p$
  $promises \leftarrow promises \cup \{(a, n', v')\}$; /* add a for acceptor disambiguation */
  if $|promises| = \left\lceil \frac{|A|+1}{2} \right\rceil$ **then**
    $v \leftarrow$ MAXVALUE$(promises)$; /* value with the largest n */
    $v_p \leftarrow$ **if** $v \neq \bot$ **then** v **else** v_p; /* adopt v if present */
    foreach $a \in A$ **do**
      send $\langle$ACCEPT $\mid n_p, v_p\rangle$ **to** a;

Upon $\langle$ACK $\mid n\rangle$ **from** a **s.t.** $n = n_p$
  $acks \leftarrow acks + 1$;
  if $acks = \left\lceil \frac{|A|+1}{2} \right\rceil$ **then**
    foreach $l \in L$ **do**
      send $\langle$DECIDE $\mid v_p\rangle$ **to** l;

Upon $\langle$NACK $\mid n\rangle$ **from** a **s.t.** $n = n_p$
  ABORT() ; /* Goto $\langle$PROPOSE $\mid v\rangle$ and pick a new n_p immediately to avoid old messages being handled */

Algorithm 1: Abortable Paxos – Acceptor

Implements: Uniform Consensus
Requires: Perfect Link
Algorithm:

```
1: n_prom ← 0 ;                    /* promise not to accept in lower rounds */
2: (n_a, v_a) ← (⊥, ⊥) ;            /* sequence number and value accepted */
```

```
3: Upon ⟨Prepare | n⟩ from p
4:     if n_prom < n then
5:         n_prom ← n;
6:         send ⟨Promise | n, n_a, v_a⟩ to p;
       else
7:         send ⟨Nack | n⟩ to p;                    /* optimisation only */
8: Upon ⟨Accept | n, v⟩ from p
9:     if n_prom ≤ n then
10:        n_prom ← n;
11:        (n_a, v_a) ← (n, v);
12:        send ⟨Accepted | n⟩ to p;
       else
13:        send ⟨Nack | n⟩ to p;                    /* optimisation only */
```

Algorithm 1: Abortable Paxos – Learner

Implements: Uniform Consensus
Requires: Perfect Link
Algorithm:

```
1: v_d ← ⊥ ;                                        /* decided value */
```

```
2: Upon ⟨Decide | v⟩
3:     if v_d = ⊥ then
4:         v_d ← v;
5:         trigger ⟨Decide | v_d⟩;
```

3 Sequence Paxos

If our goal is to build a replicated state machine (RSM) based on Paxos, deciding a single value will not suffice. Instead we want to agree on a *sequence of values* that is fed into a deterministic automaton on each replica, such that replicas that have seen the same sequence will have the same state (assuming they started from the same initial state). Such a sequence of values can be seen as a *replicated log* of state machine *commands*.

Naïve Approach One way to extend Paxos for this scenario is to augment the proposals with instance numbers and run a single value Paxos for each instance in sequential rounds. Thus, at round i each process starts a new instance of Paxos. If it has commands it wants to propose (in a set *proCmds*) and it has not proposed (variable *proposed* = FALSE) in the current round already, it proposes some command $C \in proCmds$ as $\langle \text{PROPOSE} \mid C, p, i \rangle$ (where p is the process id of the client that sent C) and sets *proposed* $\leftarrow$ TRUE. Once it sees a $\langle \text{DECIDE} \mid C', p', i \rangle$, it removes (C', p') from *proCmds* and appends (C', p', i) to *log*. It then executes the command on the state machine $(s_i, res_i) = C(s_{i-1})$, and returns res_i to p'. At this point the round ends, it resets *proposed* $\leftarrow$ FALSE and moves to the next round $i + 1$.

The issue with this approach is, that it is completely sequential, working on one round after the other and taking (at least) 4 communication steps (2 round-trips) for each round. Trying to improve performance by pipelining is not straight-forward, as duplicate command entries or log holes must be avoided. However, the obvious optimisation of preparing multiple instance ahead of time and only running the *accept phase* sequentially, since v is not needed in the *prepare phase*, halves the required number of communication steps on the "hot path" (and with some batching also reduces it overall) [9].

3.1 Sequence Consensus

In order to match our abstraction better with the requirements of a replicated log, a change in interface and desired properties is needed. We will still propose a single command C, but we now decide on a sequence of commands CS. The original *Uniform Consensus* properties UC1-4 are altered as shown below:

SC1 (Validity) If process p decides CS then CS is a sequence of proposed commands (without duplicates[1]).

SC2 (Uniform Agreement) If process p decides CS and process q decides CS' then one is a prefix of the other.

SC3 (Integrity) If process p decides CS and later decides CS' then CS is a strict prefix of CS'.

[1] It is also possible to allow duplicates in the log and filter them out at the state-machine level instead. This simplifies the implementation.

SC4 (Termination) If a command C is proposed infinitely often by a correct process, then eventually every correct process decides a sequence containing C. If duplication is allow, then the decided sequence will contain C infinitely often.

3.2 Initial Sequence Paxos Implementation

To make it easy to see that the algorithm is correct, we will start with a very simple and inefficient variant of single value Paxos to implement *Sequence Consensus*, and then later add optimisation transformations to it step by step, preserving correctness with each change. We start with the basic *Paxos* presented in algorithm 1 and make the following changes: All values are now sequences and the empty value $\perp$ becomes the empty sequence $\langle\rangle$. After adopting the sequence (value) with the highest proposal number (alg. 1 l. 18), the sequence is extended by one or more new commands instead of replacing the value. Instead of deciding only if there has been no previous decision, learners will now decide whenever the received sequence is longer than the previously decided one. In order to abstract over the **SC1** variants with or without duplication we use the *append* operator $\oplus$ with the following two definition variants:

No Duplicates

$$\langle C_1, \ldots, C_n \rangle \oplus C \stackrel{def}{=} \begin{cases} \langle C_1, \ldots, C_n \rangle, & \text{if } C \text{ is equal to some } C_i \\ \langle C_1, \ldots, C_n, C \rangle, & \text{otherwise} \end{cases}$$

Duplicates Allowed

$$\langle C_1, \ldots, C_n \rangle \oplus C \stackrel{def}{=} \langle C_1, \ldots, C_n, C \rangle$$

Algorithm 2 shows the described implementation.

Correctness The only changes that have been made affect how values are treated, while the round numbers have been left untouched. The same mechanism ensuring that chosen values are never replaced, now ensures that chosen sequences are always extended and thus no sub-sequences disappear. It is easy to see that this algorithm is as correct as algorithm 1.

Performance As far as efficiency is concerned, however, we have gained nothing, so far, over the naïve multi-instance Paxos described above. In fact, algorithm 2 is actually worse, as it sends whole sequences of commands in every step, so its performance actually degrades with the growth of the log.

Before we get to a more efficient leader-based implementation where all roles run in each process, we must first make a short excursion into leader election.

Algorithm 2: Initial Sequence Paxos – Proposer

Implements: Sequence Consensus
Requires: Perfect Link
Algorithm:

```
1: A ;                                                  /* set of acceptors */
2: L ;                                                  /* set of learners */
3: s ← 0 ;                                              /* local sequence number */
4: (n_p, v_p) ← (⊥, ⟨⟩) ;                               /* unique round number and sequence */
5: C_p ← ⊥ ;                                            /* command we are currently trying to append */
6: promises ← ∅;
7: acks ← 0;
---
8: Upon ⟨Propose | C⟩
9:     s ← s + 1
10:    n_p ← unique(s) ;                                /* use pid to make n globally unique */
11:    C_p ← C
12:    promises ← ∅;
13:    acks ← 0;
       foreach a ∈ A do
14:        send ⟨Prepare | n_p⟩ to a;
15: Upon ⟨Promise | n, n', v'⟩ from a s.t. n = n_p
16:    promises ← promises ∪ {(a, n', v')};             /* add a for acceptor disambiguation */
17:    if |promises| = ⌈(|A|+1)/2⌉ then
18:        v ← maxValue(promises);                      /* sequence with the largest n */
19:        v_p ← v ⊕ C;                                 /* adopt v and append */
           foreach a ∈ A do
20:            send ⟨Accept | n_p, v_p⟩ to a;
21: Upon ⟨Ack | n⟩ from a s.t. n = n_p
22:    acks ← acks + 1;
23:    if acks = ⌈(|A|+1)/2⌉ then
24:        C_p ← ⊥;
           foreach l ∈ L do
25:            send ⟨Decide | v_p⟩ to l;
26: Upon ⟨Nack | n⟩ from a s.t. n = n_p
27:    abort() ;    /* Goto ⟨Propose | C⟩ and pick a new n_p immediately to avoid old messages being handled */
```

Algorithm 2: Initial Sequence Paxos – Acceptor

Implements: Sequence Consensus
Requires: Perfect Link
Algorithm:

1: $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
2: $(n_a, v_a) \leftarrow (\bot, \langle\rangle)$; /* round number and sequence accepted */

3: **Upon** $\langle$PREPARE | $n\rangle$ **from** p
4: **if** $n_{prom} < n$ **then**
5: $n_{prom} \leftarrow n$;
6: **send** $\langle$PROMISE | $n, n_a, v_a\rangle$ **to** p;
else
7: **send** $\langle$NACK | $n\rangle$ **to** p; /* optimisation only */
8: **Upon** $\langle$ACCEPT | $n, v\rangle$ **from** p
9: **if** $n_{prom} \leq n$ **then**
10: $n_{prom} \leftarrow n$ $(n_a, v_a) \leftarrow (n, v)$;
12: **send** $\langle$ACCEPTED | $n\rangle$ **to** p;
else
13: **send** $\langle$NACK | $n\rangle$ **to** p ; /* optimisation only */

Algorithm 2: Initial Sequence Paxos – Learner

Implements: Sequence Consensus
Requires: Perfect Link
Algorithm:

1: $v_d \leftarrow \langle\rangle$; /* decided sequence */

2: **Upon** $\langle$DECIDE | $v\rangle$
3: **if** $|v_d| < |v|$ **then**
4: $v_d \leftarrow v$;
5: **trigger** $\langle$DECIDE | $v_d\rangle$;

4 Ballot Leader Election

As described at the end of section 2, the algorithms presented so far may actually never terminate under the given assumptions without the use of an Ω leader election abstraction. In this section we want to integrate Ω into the Sequence Paxos algorithm 2, but at the same time we want to outsource the generation of ballot numbers to the leader election, in order to make the replicated log part of the algorithm easier to follow. Thus, the idea is to elect a leader together with a ballot number that is globally unique and locally monotonically increasing. This leader-ballot pair will then be used by the Sequence Paxos algorithm to start a prepare phase. The new abstraction is called *Ballot Leader Election* (*BLE*) and has a single event $\langle \textsc{Leader} \mid p, n \rangle$ where p is the process that is now leader and n its ballot number. *BLE* implementations must fulfil the following properties, which are an extension of Ω's properties:

BLE1 (Completeness) Eventually, every correct process elects some correct process, if a majority of processes is correct.

BLE2 (Eventual Agreement) Eventually, no two correct processes elect different correct processes.

BLE3 (Monotonic Unique Ballots) If a process L with ballot n is elected as leader by a process p, then all previously elected leaders by p have ballot numbers m with $m < n$, and the pair (L, n) is globally unique.

In the implementation we will allow a process p to "inaccurately" drop a correct leader, as long as the new leader has a higher ballot number. We will also require that a process is elected as a leader only if a majority of processes are correct and alive. As this is anyway required for Sequence Paxos, it does not constitute a limitation in any noticeable way.
We will start by assuming a fail-noisy model, that is processes fail by crashing, partially-synchronous system model, and perfect links channel abstraction. However, the final algorithm will turn out to actually work fine in a slightly weaker model that allows message loss and crash-recovery.

4.1 Gossip Leader Election

The basic idea for the algorithm is as follows: Each process p has its own unique ballot n formed from a sequence number s and its process id, such that $n = (s, pid_p)$. This pair is trivially unique, as pid_p is unique. For an implementation this can be folded into a single (potentially long) number, by taking the size of the process id set Π and multiplying it with s, such that $n = s \cdot |\Pi| + pid_p$. If Π is not known a-priori, any number guaranteed to be larger than $|\Pi|$ can be substituted, for example `Int.MAX`. Given this, each process gossips the its ballot number along with the usual failure-detection heartbeats (with a repeating delay, adjustable by a constant Δ) to all other processes. Eventually, each correct process will elect the process with the highest rank (max ballot), given good network conditions (**BLE2**). However, a process will only trust a leader, if the leader's ballot is

among the collected max ballots from a majority of processes. If a process does not find its current leader's ballot in that set, it will increase its own sequence number s, recalculate its ballot n as above and wait until a new ballot gets a majority. This satisfies **BLE3** and also **BLE1** assuming a sufficiently long stable period as per the partially-synchronous model. Algorithm 3 shows the pseudocode for the described implementation, with the ballot generation part hidden behind the INCREMENT(*ballot*) function.

Algorithm 3: Gossip Leader Election

Implements: Ballot Leader Election
Requires: Perfect Link
Algorithm:

```
1:  Π;                                                    /* Process set */
2:  round ← 0 ;                                           /* round number */
3:  ballots ← ∅;
4:  n ← (0, pid) ;                                        /* ballot number */
5:  L ← ⊥ ;                                               /* leader */
6:  n_max ← n ;                                           /* largest ballot number seen */
7:  d ← Δ ;                                               /* heartbeat delay */
8:  startTimer(d);                  /* schedule a timeout event in d timeunits */
```

```
9:  Fun checkLeader()
10:     top = (topProcess, topN) ← maxByBallot(ballots ∪ {(self, n)});
        if topN < n_max then
            while n ≤ n_max do
11:             n ← increment(n);
12:         L ← ⊥;
        else
            if top ≠ L then
13:             n_max ← topN;
14:             L = top;
15:             trigger ⟨Leader | topProcess, topN⟩;
16: Upon ⟨Timeout⟩
        if |ballots| + 1 > ⌈(|Π|+1)/2⌉ then
17:         checkLeader();
18:     ballots ← ∅;
19:     round ← round + 1;
        foreach p ∈ Π s.t. p ≠ self do
20:         send ⟨HeartbeatRequest | round, n_max⟩ to p;
21:     startTimer(d);
22: Upon ⟨HeartbeatRequest | r, b_max⟩ from p
        if b_max > n_max then
23:         n_max ← b_max;
24:     send ⟨HeartbeatReply | r, n⟩ to p;
25: Upon ⟨HeartbeatReply | r, b⟩ from p
        if r = round then
26:         ballots ← ballots ∪ {(p, b)};
        else
27:         d ← d + Δ ; /* Increase delay to make sure all replies from the
                           current round are received within the time window. */
```

5 Leader-based Sequence Paxos

In this section we will make step-by-step transformations to improve the performance of Sequence Paxos.

5.1 BLE and Uniform Processes

Now that we have a leader election abstraction that also provides ballot numbers, we will adapt the Sequence Paxos algorithm from section **??** to use *BLE*. At the same time we will make an additional assumption, that all roles are available for every process, i.e. each process is a proposer, acceptor, and learner. This a common way of running a log replication service in practice, and it will allow each process (a replica) to share the acceptor and learner state information. At each process p we also introduce a *state* variable, that tracks both p's current role in the algorithm, LEADER or FOLLOWER, and the phase it is currently in, PREPARE or ACCEPT (or none $\perp$). Every process starts as a FOLLOWER and can move to a LEADER state by being elected by *BLE*, in which case it will stay until overrun by another leader, at which point it will revert to acting as a follower.
As we outsourced the leader election part to *BLE* now, we will also introduce some optimisations that assume leaders are long-lived. Particularly, we want to pipeline ACCEPT messages while in the *accept phase*. That is, a leader that completed the *prepare phase* will only send ACCEPT for every proposed command, extending the previous sequence and thus ensuring that chosen sequences are incrementally extended, until the round is aborted by a new election. As Perfect Links do not guarantee ordering, both *acceptor* logic must now ensure they only store longer sequences than they already have in the same round. This guarantees that the longest chosen sequence is still a prefix of the accepted sequence, thereby satisfying the *Agreement* property (SC2). Additionally, the leader has to find out what the longest prefix that it has seen ACCEPTED (formerly ACK) messages from a majority, i.e. the longest chosen prefix, before deciding on a new sequence. Only if that prefix is longer than what it has already decided (variable l_c), can it issue a DECIDE and only for that prefix.
The complete pseudocode can be seen in algorithm 4.

Correctness By adding Leader Election, the new algorithm fixes the liveness issues of the previous algorithms as discussed at the end of section 2. Now a replica plays the role of a proposer, acceptor, and learner. Since there were no assumptions on role distribution before, making a stronger assumption here does not affect correctness at all. The introduction of the states is simply a consequence of the previous two decisions and introduction of pipelining, and does not cause any issues by itself. The pipelining is an optimisation that is guaranteed to be safe, as *acceptors* that have moved to a new leader already, will ignore ACCEPT messages from their old leader that has a lower ballot. As DECIDE messages are only issued for chosen sequences, **SC2** and **SC3** are satisfied.

Performance The prepare phase takes a single round trip, after which all commands can be "pipelined". Pipelining allows many commands to be "in flight" in parallel, and for each new command it only takes a single round-trip to get decided locally, as long as the leader remains in place. However, we still send full sequences causing degrading performance with the growth of the log. And, as we have merged the roles now, we are actually also keeping (mostly) redundant sequences in memory, specifically v_L, v_a, and v_d.

Algorithm 4: Sequence Paxos 2 – State

Implements: Sequence Consensus
Requires: Perfect Link, BLE
Algorithm:

1: Π ; /* set of processes */
2: $state \leftarrow (\text{FOLLOWER}, \bot)$; /* role and phase state */
/* Proposer State */
3: $(n_L, v_L) \leftarrow (\bot, \langle\rangle)$; /* Leader's round number and sequence */
4: $promises \leftarrow \emptyset$;
5: $las \leftarrow [0]^{|\Pi|}$; /* Length of longest accepted sequence per acceptor */
6: $propCmds \leftarrow \emptyset$; /* A set of commands that need to be appended to the log. */
7: $l_c \leftarrow 0$; /* Length of longest chosen sequence */
/* Acceptor State */
8: $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
9: $(n_a, v_a) \leftarrow (\bot, \langle\rangle)$; /* round number and sequence accepted */
/* Learner State */
10: $v_d \leftarrow \langle\rangle$; /* decided sequence */

Algorithm 4: Sequence Paxos 2 – Acceptor&Learner

```
   /* Acceptor Code                                                    */
1: Upon ⟨Prepare | n⟩ from p
2:     if n_prom < n then
3:         n_prom ← n;
4:         send ⟨Promise | n, n_a, v_a⟩ to p;
5: Upon ⟨Accept | n, v⟩ from p
6:     if n_prom ≤ n then
7:         n_prom ← n;
8:         (n_a, v_a) ← max((n, v), (n_a, v_a));
9:         send ⟨Accepted | n, |v_a|⟩ to p;
   /* Learner Code                                                     */
10: Upon ⟨Decide | v⟩
11:    if |v_d| < |v| then
12:        v_d ← v;
13:        trigger ⟨Decide | v_d⟩;
14: Fun max((n, v), (n', v'))
       if n ≠ n' then
          if n > n' then
             (n, v)
          else
             (n', v')
       else
          if |v| > |v'| then
             (n, v)
          else
             (n', v')
```

Algorithm 4: Sequence Paxos 2 – Proposer

```
   /* Proposer Code                                                          */
1: Upon ⟨Leader | L, n⟩
       if self = L ∧ n > n_L then
2:         n_L ← n;
3:         promises ← ∅;
4:         las ← [0]^|Π|;
5:         l_c ← 0;
6:         state ← (leader, prepare);
           foreach p ∈ Π do
7:             send ⟨Prepare | n_L⟩ to p;
       else
8:         state ← (follower, state.2);
9: Upon ⟨Propose | C⟩ s.t. state = (leader, prepare)
10:        propCmds ← propCmds ∪ {C};
11: Upon ⟨Propose | C⟩ s.t. state = (leader, accept)
12:        v_L ← v_L ⊕ C;
           foreach p ∈ Π do
13:            send ⟨Accept | n_L, v_L⟩ to p;
14: Upon ⟨Promise | n, n', v'⟩ from a s.t. n = n_L ∧ state = (leader, prepare)
15:        promises ← promises ∪ {(a, n', v')};          /* add a for acceptor
            disambiguation */
16:        if |promises| = ⌈(|Π|+1)/2⌉ then
17:            v ← maxValue(promises);          /* sequence with the largest n,
                longest if equal */
18:            v_L ← v ⊕ C forall C ∈ propCmds;          /* adopt v and append */
19:            propCmds ← ∅;
               foreach p ∈ Π do
20:                send ⟨Accept | n_L, v_L⟩ to p;
21:            state ← (leader, accept);
22: Upon ⟨Accepted | n, l_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
23:        las[a] ← max(l_a, las[a]);
24:        m ← max{l ∈ ℕ | ∃_{M⊆Π} |M| ≥ ⌈(|Π|+1)/2⌉ ∧ ∀_{p∈M} l ≤ las[p]};      /* longest
            chosen */
25:        if m > l_c then
26:            l_c ← m;
               foreach p ∈ Π do
27:                send ⟨Decide | prefix(v_L, m)⟩ to p;  /* Send the chosen prefix
                    to p */
   /* Drop Nack as getting stuck is avoided by BLE re-election               */
```

5.2 Removing Redundant State

As all the roles are run by each process now, each role's code actually has access to the state variables of the others. Particularly, v_L, v_a, v_d have significant overlap, since $v_d \prec v_a$ and $v_d \prec v_L$. Thus, *if* we can guarantee that ACCEPT messages always arrive before the corresponding DECIDE messages, we can simply replace v_d with a pointer l_d into v_a that marks the decided prefix, i.e. $v_d = \text{PREFIX}(v_a, l_d)$. Additionally, we can get rid of v_L, if we skip sending PREPARE messages to and receiving promises from the leader itself, by simply writing the equivalent changes directly into its local state.
In order to enable the first optimisation, we are going to strengthen our assumption on the channel abstraction. We will now require *FIFO Perfect Links* that provide message ordering guarantees. This requirement costs us nothing in performance, as before out-of-order commands anyway had to be buffered before being decided. Additionally, it is a simple extension to Perfect Links, achieved by adding sequence numbers and buffering delivery. If our implementation of Perfect Links before was, in fact, TCP[2], then we get FIFO Perfect Links for free.
The complete pseudocode can be seen in algorithm 5.

Correctness FIFO Perfect Links guarantee that ACCEPT messages always arrive before the corresponding DECIDE, thus it always holds that for any replica q v_d at $_q \prec v_a$ at $_q$. To remove v_L notice that the leader has access to its own v_a. When a process becomes leader it can update its local v_a directly. This is guaranteed to be accepted, since n_L is guaranteed to be higher or equal to the current n_{prom}.

Performance Apart from saving one message per command (the self-message), these optimisations have mostly reduced the memory footprint of the algorithm. Instead of storing three sequences, we now store only one sequence and one new pointer/index.

[2] Note that TCP only fulfils this abstraction during a single session.

Algorithm 5: Sequence Paxos 3 – State

Implements: Sequence Consensus
Requires: FIFO Perfect Link, BLE
Algorithm:

1: Π ; /* set of processes */
2: $\Pi_o \leftarrow \Pi - \{self\}$;
3: $state \leftarrow (\text{FOLLOWER}, \bot)$; /* role and phase state */
/* Proposer State */
4: $n_L \leftarrow 0$; /* leader's round number */
5: $promises \leftarrow \emptyset$;
6: $las \leftarrow [0]^{|\Pi|}$; /* length of longest accepted sequence per acceptor */
7: $propCmds \leftarrow \emptyset$; /* set of commands that need to be appended to the log */
8: $l_c \leftarrow 0$; /* length of longest chosen sequence */
/* Acceptor State */
9: $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
10: $(n_a, v_a) \leftarrow (\bot, \langle\rangle)$; /* round number and sequence accepted */
/* Learner State */
11: $l_d \leftarrow 0$; /* length of the decided sequence */

Algorithm 5: Sequence Paxos 3 – Acceptor&Learner

```
   /* Acceptor Code                                                    */
1: Upon ⟨Prepare | n⟩ from p
2:     if n_prom < n then
3:         n_prom ← n;
4:         send ⟨Promise | n, n_a, v_a⟩ to p;
5: Upon ⟨Accept | n, v⟩ from p
6:     if n_prom ≤ n then
7:         n_prom ← n;
8:         (n_a, v_a) ← max((n, v), (n_a, v_a));
9:         send ⟨Accepted | n, |v_a|⟩ to p;
   /* Learner Code                                                     */
10: Upon ⟨Decide | v, n⟩ s.t. n = n_prom
11:     if l_d < |v| then
12:         l_d ← |v|;
13:         trigger ⟨Decide | prefix(v_a, l_d)⟩;
14: Fun max((n, v), (n′, v′))
        if n ≠ n′ then
            if n > n′ then
                (n, v)
            else
                (n', v')
        else
            if |v| > |v′| then
                (n, v)
            else
                (n', v')
```

Algorithm 5: Sequence Paxos 3 – Proposer

```
   /* Proposer Code                                                          */
1: Upon ⟨Leader | L, n⟩
       if self = L ∧ n > n_L then
2:         (n_L, n_prom) ← (n, n);
3:         promises ← {(self, n_a, v_a)};
4:         las ← [0]^|Π|;
5:         l_c ← 0;
6:         state ← (leader, prepare);
           foreach p ∈ Π_o do
7:             send ⟨Prepare | n_L⟩ to p;
       else
8:         state ← (follower, state.2);
9: Upon ⟨Propose | C⟩ s.t. state = (leader, prepare)
10:        propCmds ← propCmds ∪ {C};
11: Upon ⟨Propose | C⟩ s.t. state = (leader, accept)
12:        v_a ← v_a ⊕ C;
13:        las[self] ← |v_a|;
           foreach p ∈ Π_o do
14:            send ⟨Accept | n_L, v_a⟩ to p;
15: Upon ⟨Promise | n, n', v'⟩ from a s.t. n = n_L ∧ state = (leader, prepare)
16:        promises ← promises ∪ {(a, n', v')};          /* add a for acceptor
            disambiguation */
17:        if |promises| = ⌈(|Π|+1)/2⌉ then
18:            v ← maxValue(promises);          /* sequence with the largest n,
                longest if equal */
19:            v_a ← v ⊕ C forall C ∈ propCmds;          /* adopt v and append */
20:            propCmds ← ∅;
21:            las[self] ← |v_a|;
               foreach p ∈ Π_o do
22:                send ⟨Accept | n_L, v_a⟩ to p;
23:            state ← (leader, accept);
24: Upon ⟨Accepted | n, l_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
25:        las[a] ← l_a;
26:        M ← {p ∈ Π | las[p] ≠ ⊥ ∧ las[p] ≥ l_a};          /* support set for l_a */
27:        if l_a > l_c ∧ |M| ≥ ⌈(|Π|+1)/2⌉ then
28:            l_c ← l_a;
               foreach p ∈ Π do
29:                send ⟨Decide | prefix(v_a, l_a)⟩, n_L to p;    /* send chosen prefix
                    */
```

5.3 Avoid Sending Sequences

The current algorithm still sends full sequences with every single message. As discussed before, these sequences will overlap in most positions and at most differ slightly at the end. Furthermore, in the $\langle \text{DECIDE} \mid v, n \rangle$ message we are not even using v at all, but only its length, as we already have access to v_a and we know $v \prec v_a$. Thus we will simply replace v with $|v| = l_c$ in the DECIDE message.

In addition to this simple change, we also want to trim the data we send in the prepare phase as much as possible. To preserve correctness a leader *must* extend a sequence that contains v_d at the end of the prepare phase. But both the leader and some acceptors may be ahead or behind v_d with their v_a during the prepare phase. We want to synchronise everyone by sending as little data as possible. As a majority of acceptors had at least the chosen sequence before the leader change, any majority now must include at least a single acceptor that still knows it (as a majority may not fail). If the leader tells everyone what its decided sequence is (i.e. add l_d to PREPARE messages), the acceptors can either catch up the leader (if *it* is behind) or ask to be caught up themselves, if *they* are behind. That is, for a leader L at every acceptor a, if $l_d \text{ at } _a > l_d \text{ at } _L$ then a will send $\text{SUFFIX}(v_a, l_d \text{ at } _L)$ as part of its PROMISE to catch up the leader. Otherwise it will send $\langle\rangle$ and wait for the leader to catch it up with the first ACCEPT message. After collecting a majority, the leader will adopt the *max suffix*, that is the suffix with highest round number or, if the round numbers are equal, the longest. It will then append it to its decided sequence ($v_d = \text{PREFIX}(v_a, l_d)$) and append all the commands it wanted to propose. That value will becomes its new v_a and that is also the sequence it will impose on all followers.

At this point only the ACCEPT messages continue to send full sequences. In order to avoid this, we split off the first ACCEPT at the end of the *prepare phase* into a new message ACCEPTSYNC, as this message has a different purpose than the ACCEPT messages being generated from proposals. The purpose of the ACCEPTSYNC message is get every replica to the same state of the leader, by sending them only as much data as they really need. The purpose of the ACCEPT message is to append a single new command to v_a. If the replicas are in sync with the leader (i.e. have been part of the prepare phase), then simply sending this command alone will suffice, considering that the FIFO Perfect Links will preserve the sequence order. In order to avoid sending the whole sequence in ACCEPTSYNC, replicas have to inform the leader of their l_d in the PREPARE message, which the leader stores in a new map *lds* from each acceptor to the length of their decided sequence. Instead of sending v_a the leader then sends $\text{SUFFIX}(v_a, lds[a])$ to every acceptor a.

As only a majority of processes have responded when the leader transitions into the *accept phase*, we must add extra message handlers to catch up late replicas. Particularly, we must *not* send ACCEPT messages to a replica until we have processed its PROMISE, as those replicas are out of sync. The same reasoning goes for DECIDE messages for late replicas. That is, if a leader has already issued decides during its *accept phase* when it gets a promise from a late replica, it must send both an up-to-date ACCEPTSYNC, immediately followed by

a DECIDE for the currently longest chosen sequence l_c.
The complete pseudocode for these optimisations can be seen in algorithm 6.

Correctness Sending $|v| = l_c$ in DECIDE messages instead of v, is a trivial change, as only $|v|$ was used before anyway. Sending only diffs instead of full sequences also does not affect the correctness as long as the correct diffs are being sent. The renaming of the first ACCEPT message to ACCEPTSYNC itself has no effect, it simply reflects the usage more accurately. It would also be possible to infer this information (whether a message is ACCEPT or ACCEPTSYNC) from context, but that is less readable. The crucial part here is to use FIFO Perfect Links and *lds* to make sure that an ACCEPTSYNC always arrives before the first ACCEPT in that round.

Performance Not sending full sequences anymore is a huge improvement in performance, especially as it finally removes the performance degradation issue with large sequences. At this point, we made leader changes as cheap as we can, and during stable periods we have full pipelining for single commands. If this leads to bad performance due to network packets being too small, it could easily be extended again to do some kind of batching. Although whether this should happen within *Sequence Paxos* or in an external management component is an implementation decision.

Algorithm 6: Sequence Paxos 4 – State

Implements: Sequence Consensus
Requires: FIFO Perfect Link, BLE
Algorithm:

1: Π ; /* set of processes */
2: $\Pi_o \leftarrow \Pi - \{self\}$;
3: $state \leftarrow (\text{FOLLOWER}, \bot)$; /* role and phase state */
/* Proposer State */
4: $n_L \leftarrow 0$; /* leader's round number */
5: $promises \leftarrow \emptyset$;
6: $las \leftarrow [0]^{|\Pi|}$; /* length of longest accepted sequence per acceptor */
7: $lds \leftarrow [\bot]^{|\Pi|}$; /* length of longest known decided sequence per acceptor */
8: $propCmds \leftarrow \emptyset$; /* set of commands that need to be appended to the log */
9: $l_c \leftarrow 0$; /* length of longest chosen sequence */
/* Acceptor State */
10: $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
11: $(n_a, v_a) \leftarrow (0, \langle\rangle)$; /* round number and sequence accepted */
/* Learner State */
12: $l_d \leftarrow 0$; /* length of the decided sequence */

Algorithm 6: Sequence Paxos 4 – Acceptor&Learner

```
/* Acceptor Code                                                              */
Upon ⟨Prepare | n, ld⟩ from p
    if n_prom < n then
        n_prom ← n;
        state ← (follower, prepare);
        suffix ← suffix(v_a, ld);
        send ⟨Promise | n, n_a, suffix, l_d⟩ to p;
Upon ⟨AcceptSync | n, suffix, ld⟩ from p s.t. state = (follower, prepare)
    if n_prom = n then
        n_a ← n;
        v_a ← prefix(v_a, ld) ++ suffix;
        state ← (follower, accept);
        send ⟨Accepted | n, |v_a|⟩ to p;
Upon ⟨Accept | n, C⟩ from p s.t. state = (follower, accept)
    if n_prom = n then
        v_a ← v_a ⊕ C;
        send ⟨Accepted | n, |v_a|⟩ to p;
/* Learner Code                                                               */
Upon ⟨Decide | l, n⟩ s.t. n = n_prom
    if l_d < l then
        l_d ← l;
        trigger ⟨Decide | prefix(v_a, l_d)⟩;
Fun max((n, v), (n', v'))
    if n ≠ n' then
        if n > n' then
            (n, v)
        else
            (n', v')
    else
        if |v| > |v'| then
            (n, v)
        else
            (n', v')
```

Algorithm 6: Sequence Paxos 4 – Proposer

```
   /* Proposer Code                                                          */
1: Upon ⟨Leader | L, n⟩
     if self = L ∧ n > n_L then
2:     (n_L, n_prom) ← (n, n);
3:     promises ← {(self, n_a, suffix(v_a, l_d))};
4:     las ← [0]^|Π|;
5:     lds ← [⊥]^|Π|; lds[self] ← l_d;
6:     l_c ← 0;
7:     state ← (leader, prepare);
       foreach p ∈ Π_o do
8:       send ⟨Prepare | n_L, l_d⟩ to p;
     else
9:     state ← (follower, state.2);
10: Upon ⟨Propose | C⟩ s.t. state = (leader, prepare)
11:    propCmds ← propCmds ∪ {C};
12: Upon ⟨Propose | C⟩ s.t. state = (leader, accept)
13:    v_a ← v_a ⊕ C;
14:    las[self] ← |v_a|;
       foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
15:      send ⟨Accept | n_L, C⟩ to p;
16: Upon
    ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, prepare)
17:    promises ← promises ∪ {(a, n', suffix_a)};
18:    lds[a] ← ld_a;
19:    if |promises| = ⌈(|Π|+1)/2⌉ then
20:      suffix ← maxValue(promises);   /* suffix with max n, longest if
          equal */
         /* adopt v_d ++ suffix and append commands                          */
21:      v_a ← prefix(v_a, l_d) ++ suffix ⊕ C forall C ∈ propCmds;
22:      propCmds ← ∅;
23:      las[self] ← |v_a|;
24:      state ← (leader, accept);
         foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
25:        send ⟨AcceptSync | n_L, suffix(v_a, lds[p]), lds[p]⟩ to p;
26: Upon
    ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
27:    lds[a] ← ld_a;
28:    send ⟨AcceptSync | n_L, suffix(v_a, lds[a]), lds[a]⟩ to p;
29:    if l_c ≠ 0 then
30:      send ⟨Decide | l_c, n_L⟩ to a;      /* also inform what got decided
          already */
31: Upon ⟨Accepted | n, l_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
32:    las[a] ← l_a;
33:    M ← {p ∈ Π | las[p] ≠ ⊥ ∧ las[p] ≥ l_a};          /* support set for l_a */
34:    if l_a > l_c ∧ |M| ≥ ⌈(|Π|+1)/2⌉ then
35:      l_c ← l_a;
         foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
36:        send ⟨Decide | l_c, n_L⟩ to p;  /* send length of chosen sequence
            */
```

5.4 Final Optimisations

At this point we have a pretty efficient algorithm and all that is left is some minor optimisations and convenience fixes. One thing to notice is that the PROMISE at acceptor a currently sends a suffix to leader L even when $n_a \text{ at } _L > n_a \text{ at } _a$, although MAXVALUE(*promises*) at the leader would never pick that suffix to be adopted. Naïvely, we might think this situation would never occur, as a process that lags behind one round, probably should not have a longer v_a anyway. But this is not true. Consider a leader L_1 in round 1 getting disconnected from the rest of the group, but still extending its sequence locally. In the mean time a new leader L_2 is elected in round 2 with the remaining nodes, that L_1 never knows about. Shortly after, the connection stabilises and a new leader L_3 is elected in round 3 with L_1 part of the majority. If L_2 didn't have much time to add commands, it might be that case that $|v_a \text{ at } _{L_1}| > |v_a \text{ at } _{L_3}|$, even though $n_a \text{ at } _{L_3} > n_a \text{ at } _{L_1}$. While this is an edge case, L_1 could have millions of messages in its local log, that will never make it into the new v_a as they conflict with the decisions made in its absence. Thus sending them is unnecessary. To avoid this situation the leader will add its local n_a to every PREPARE message, which an acceptor will check before calculating the suffix to send in the PROMISE.
The other optimisation in this section is meant to reduce redundancy between the replicated log service of *Sequence Consensus* and the state machine on top of it executing commands. Currently a DECIDE event includes the full log. However, the vast majority if not all state machine implementation will not start with the *initial state* and apply the full log every time a decision is made. Instead they will only apply the new commands one at a time to the previously stored state. As we are already keeping track of previously decided prefixes in *Sequence Paxos*, it is redundant to keep track of this again in the state machine. To avoid this we will make another small alteration to the *Sequence Paxos* interface, in which we will only decide a single command at a time, with the semantic meaning of this being appended to the previously decided commands (or simply immediately consumed by the state machine).
The complete pseudocode for the final version can be seen in algorithm 7.

Correctness Both changes are minor and trivially correct, with the caveat that the latter change technically implements a different interface.

Performance The performance is the same as in the previous version, with the exception of the edge case described above where $n_a \text{ at } _L > n_a \text{ at } _a$ for some leader L and acceptor a.

Algorithm 7: Sequence Paxos Final – State

Implements: Sequence Consensus
Requires: FIFO Perfect Link, BLE
Algorithm:

1: Π ; /* set of processes */
2: $\Pi_o \leftarrow \Pi - \{self\}$;
3: $state \leftarrow (\text{FOLLOWER}, \bot)$; /* role and phase state */
/* Proposer State */
4: $n_L \leftarrow 0$; /* leader's round number */
5: $promises \leftarrow \emptyset$;
6: $las \leftarrow [0]^{|\Pi|}$; /* length of longest accepted sequence per acceptor */
7: $lds \leftarrow [\bot]^{|\Pi|}$; /* length of longest known decided sequence per acceptor */
8: $propCmds \leftarrow \emptyset$; /* set of commands that need to be appended to the log */
9: $l_c \leftarrow 0$; /* length of longest chosen sequence */
/* Acceptor State */
10: $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
11: $(n_a, v_a) \leftarrow (0, \langle\rangle)$; /* round number and sequence accepted */
/* Learner State */
12: $l_d \leftarrow 0$; /* length of the decided sequence */

Algorithm 7: Sequence Paxos Final – Acceptor&Learner

```
   /* Acceptor Code                                                    */
1: Upon ⟨Prepare | n, ld, na_L⟩ from p
2:     if n_prom < n then
3:         n_prom ← n;
4:         state ← (follower, prepare);
5:         suffix ← if n_a ≥ na_L then suffix(v_a, ld) else ⟨⟩;
6:         send ⟨Promise | n, n_a, suffix, l_d⟩ to p;
7: Upon ⟨AcceptSync | n, suffix, ld⟩ from p s.t. state = (follower, prepare)
8:     if n_prom = n then
9:         n_a ← n;
10:        v_a ← prefix(v_a, ld) ++ suffix;
11:        state ← (follower, accept);
12:        send ⟨Accepted | n, |v_a|⟩ to p;
13: Upon ⟨Accept | n, C⟩ from p s.t. state = (follower, accept)
14:    if n_prom = n then
15:        v_a ← v_a ⊕ C;
16:        send ⟨Accepted | n, |v_a|⟩ to p;
   /* Learner Code                                                     */
17: Upon ⟨Decide | l, n⟩ s.t. n = n_prom
18:    while l_d < l do
19:        trigger ⟨Decide | v_a[l_d]⟩;        /* assuming 0-based indexing */
20:        l_d ← l_d + 1;
```

Algorithm 7: Sequence Paxos Final – Proposer

```
   /* Proposer Code                                                          */
1: Upon ⟨Leader | L, n⟩
       if self = L ∧ n > n_L then
2:         (n_L, n_prom) ← (n, n);
3:         promises ← {(self, n_a, suffix(v_a, l_d))};
4:         las ← [0]^|Π|;
5:         lds ← [⊥]^|Π|; lds[self] ← l_d;
6:         l_c ← 0;
7:         state ← (leader, prepare);
           foreach p ∈ Π_o do
8:             send ⟨Prepare | n_L, l_d, n_a⟩ to p;
       else
9:         state ← (follower, state.2);
10: Upon ⟨Propose | C⟩ s.t. state = (leader, prepare)
11:     propCmds ← propCmds ∪ {C};
12: Upon ⟨Propose | C⟩ s.t. state = (leader, accept)
13:     v_a ← v_a ⊕ C;
14:     las[self] ← |v_a|;
        foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
15:         send ⟨Accept | n_L, C⟩ to p;
16: Upon
    ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, prepare)
17:     promises ← promises ∪ {(a, n', suffix_a)};
18:     lds[a] ← ld_a;
19:     if |promises| = ⌈(|Π|+1)/2⌉ then
20:         suffix ← maxValue(promises);   /* suffix with max n, longest if
                equal */
            /* adopt v_d ++ suffix and append commands                        */
21:         v_a ← prefix(v_a, l_d) ++ suffix ⊕ C forall C ∈ propCmds;
22:         propCmds ← ∅;
23:         las[self] ← |v_a|;
24:         state ← (leader, accept);
            foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
25:             send ⟨AcceptSync | n_L, suffix(v_a, lds[p]), lds[p]⟩ to p;
26: Upon
    ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
27:     lds[a] ← ld_a;
28:     send ⟨AcceptSync | n_L, suffix(v_a, lds[a]), lds[a]⟩ to p;
29:     if l_c ≠ 0 then
30:         send ⟨Decide | l_c, n_L⟩ to a;      /* also inform what got decided
                already */
31: Upon ⟨Accepted | n, l_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
32:     las[a] ← l_a;
33:     M ← {p ∈ Π | las[p] ≠ ⊥ ∧ las[p] ≥ l_a};          /* support set for l_a */
34:     if l_a > l_c ∧ |M| ≥ ⌈(|Π|+1)/2⌉ then
35:         l_c ← l_a;
            foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
36:             send ⟨Decide | l_c, n_L⟩ to p;  /* send length of chosen sequence
                    */
```

6 Fail-Recovery

At his point we have a fairly efficient algorithm as long as a majority never crashes. This limitation arises from our use of the fail-stop model, so far, where correctness means never crashing. For any long-running service an assumption of never losing a cumulative majority becomes impractical, as even small failure probabilities accumulate over time. We will treat this issue in two distinct steps: This section will deal with transient failures, where the OS process crashes and then is started again or a physical host reboots, for example. In section 7 we will describe how to deal with permanent process failures by reconfiguring the members of the replication group.
In the *fail-recovery* model, a process is considered *correct* if it crashes and consequently recovers a finite number of times in an execution. During the crash a process may lose all its state (called *amnesia*) and an arbitrary suffix of the most recent messages (*omission* failures). However, a node may store some of its state in a *persistent* manner, which can be loaded during recovery. In reality this translates to storage on disk, for example. As persistent storage typically has a performance impact, we want to store only the minimum necessary state in this manner.

6.1 Sequence Paxos Recovery

In order to augment algorithm 7 for the fail-recovery model, we introduce a new *state* called RECOVER, which is automatically entered, when the system detects that it has state from a previous run available. The following variables need to be stored in persistent storage and loaded during recovery: n_{prom}, n_a, v_a, l_d. We will also require our *BLE* implementation to start with $ballot_{max} = n_{prom}$, reusing the already stored variable from *Sequence Paxos* for efficiency.
During recovery a process p starts with $state = (\text{FOLLOWER}, \text{RECOVER})$ and restores all the persistent variables. It then waits for a $\langle \text{LEADER} \mid L, n \rangle$ message, ignoring all other messages.

Case 1 ($p = L$). In this case p has been elected leader and should simply run a *prepare phase* as normal. Whatever state and messages it has missed while crashed will be synced up during the *prepare phase*.

Case 2 ($p \neq L$). In this case p is a follower, and there is some other leader actively sending ACCEPT messages already. It is unclear whether p had completed the *prepare phase* before crashing, but in any case it needs to sync up again with the other nodes before it can handle any ACCEPT messages. To facilitate this with minimal code changes, we will introduce a new message called PREPAREREQ which p will send to L. Then p will wait for L to send it a PREPARE message. From this point the algorithm proceeds as if p was simply a late process without any further changes. If L was stuck in the *prepare phase* due to a missing majority without p, resending PREPARE to p will help it make progress. If L already moved to the *accept* phase, it will know how to catch up p with an ACCEPTSYNC message.

The complete pseudocode for this version can be seen in algorithm 8.

Correctness All the state variables apart from n_{prom}, n_a, v_a, l_d is going to be overwritten during the *prepare phase*, which we are running after recovery anyway, so storing it would be redundant. As the purpose of the *prepare phase* is to get all the replicas in sync, it can easily be seen that the proposed behaviour will counter any *amnesia* or *omission* that occurred during the crash at p.

Algorithm 8: Sequence Paxos Fail-Recovery – State

Implements: Sequence Consensus
Requires: FIFO Perfect Link, BLE
Algorithm:

1: Π ; /* set of processes */
2: $\Pi_o \leftarrow \Pi - \{self\}$;
3: $state \leftarrow (\text{FOLLOWER}, \bot)$; /* role and phase state */
/* Proposer State */
4: $n_L \leftarrow 0$; /* leader's round number */
5: $promises \leftarrow \emptyset$;
6: $las \leftarrow [0]^{|\Pi|}$; /* length of longest accepted sequence per acceptor */
7: $lds \leftarrow [\bot]^{|\Pi|}$; /* length of longest known decided sequence per acceptor */
8: $propCmds \leftarrow \emptyset$; /* set of commands that need to be appended to the log */
9: $l_c \leftarrow 0$; /* length of longest chosen sequence */
/* Acceptor State */
10: **persistent** $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
11: **persistent** $(n_a, v_a) \leftarrow (0, \langle\rangle)$; /* round number and sequence accepted */
/* Learner State */
12: **persistent** $l_d \leftarrow 0$; /* length of the decided sequence */

Algorithm 8: Sequence Paxos Fail-Recovery – Part 1

```
   /* Proposer Code                                                    */
1: Upon ⟨Leader | L, n⟩
     if self = L ∧ n > n_L then
2:     (n_L, n_prom) ← (n, n);
3:     promises ← {(self, n_a, suffix(v_a, l_d))};
4:     las ← [0]^|Π|;
5:     lds ← [⊥]^|Π|; lds[self] ← l_d;
6:     l_c ← 0;
7:     state ← (leader, prepare);
       foreach p ∈ Π_o do
8:       send ⟨Prepare | n_L, l_d, n_a⟩ to p;
     else
       if state = (_, recover) then
9:       send ⟨PrepareReq⟩ to L;
       else
10:      state ← (follower, state.2);
11: Upon ⟨PrepareReq⟩ from a s.t. state = (leader, _)
12:   send ⟨Prepare | n_L, l_d, n_a⟩ to a;
13: Upon ⟨Propose | C⟩ s.t. state = (leader, prepare)
14:   propCmds ← propCmds ∪ {C};
15: Upon ⟨Propose | C⟩ s.t. state = (leader, accept)
16:   v_a ← v_a ⊕ C;
17:   las[self] ← |v_a|;
      foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
18:     send ⟨Accept | n_L, C⟩ to p;
19: Upon
    ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, prepare)
20:   promises ← promises ∪ {(a, n', suffix_a)};
21:   lds[a] ← ld_a;
22:   if |promises| = ⌈(|Π|+1)/2⌉ then
23:     suffix ← maxValue(promises);   /* suffix with max n, longest if
          equal */
        /* adopt v_d ++ suffix and append commands                      */
24:     v_a ← prefix(v_a, l_d) ++ suffix ⊕ C forall C ∈ propCmds;
25:     propCmds ← ∅;
26:     las[self] ← |v_a|;
27:     state ← (leader, accept);
        foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
28:       send ⟨AcceptSync | n_L, suffix(v_a, lds[p]), lds[p]⟩ to p;
29: Upon
    ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
30:   lds[a] ← ld_a;
31:   send ⟨AcceptSync | n_L, suffix(v_a, lds[a]), lds[a]⟩ to p;
32:   if l_c ≠ 0 then
33:     send ⟨Decide | l_c, n_L⟩ to a;      /* also inform what got decided
          already */
```

Algorithm 8: Sequence Paxos Fail-Recovery – Part 2

```
   /* Proposer Code (continued)                                          */
1: Upon ⟨Accepted | n, l_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
2:     las[a] ← l_a;
3:     M ← {p ∈ Π | las[p] ≠ ⊥ ∧ las[p] ≥ l_a};        /* support set for l_a */
4:     if l_a > l_c ∧ |M| ≥ ⌈(|Π|+1)/2⌉ then
5:         l_c ← l_a;
           foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
6:             send ⟨Decide | l_c, n_L⟩ to p;  /* send length of chosen sequence
                */
   ──────────────────────────────────────────────────────────────────────
   /* Acceptor Code                                                      */
7: Upon ⟨Prepare | n, ld, na_L⟩ from p
8:     if n_prom < n then
9:         n_prom ← n;
10:        state ← (follower, prepare);
11:        suffix ← if n_a ≥ na_L then suffix(v_a, ld) else ⟨⟩;
12:        send ⟨Promise | n, n_a, suffix, l_d⟩ to p;
13: Upon ⟨AcceptSync | n, suffix, ld⟩ from p s.t. state = (follower, prepare)
14:    if n_prom = n then
15:        n_a ← n;
16:        v_a ← prefix(v_a, ld) ++ suffix;
17:        state ← (follower, accept);
18:        send ⟨Accepted | n, |v_a|⟩ to p;
19: Upon ⟨Accept | n, C⟩ from p s.t. state = (follower, accept)
20:    if n_prom = n then
21:        v_a ← v_a ⊕ C;
22:        send ⟨Accepted | n, |v_a|⟩ to p;
   ──────────────────────────────────────────────────────────────────────
   /* Learner Code                                                       */
23: Upon ⟨Decide | l, n⟩ s.t. n = n_prom
24:    while l_d < l do
25:        trigger ⟨Decide | v_a[l_d]⟩;        /* assuming 0-based indexing */
26:        l_d ← l_d + 1;
```

6.2 Link Session Drop

It was alluded to before, that TCP can be used to implement the FIFO Perfect Link abstraction, but only during a single session. This naturally begs the question of how to behave when a session drop event does occur. The semantics of session drop are equivalent to the *omission* failures that occur during crash-recovery, that is an arbitrary suffix of the most recent messages will be lost. As this is clearly a sub-variant of a full crash-recovery event, i.e. a recovery without *amnesia*, we will treat it very similarly. That is, at every process p we will behave as follows in response to a $\langle \textsc{ConnectionLost} \mid q \rangle$ event, indicating that the link session with process q was dropped:

Case 1 (state $= (\textsc{leader}, \_)$*).* If we are leader, we simply continue as normal. The algorithm already handles the case where we are lacking a majority to proceed, so there is nothing else we can do except hope that q comes back up and re-establishes connection in the future.

Case 2 (state $= (\textsc{follower}, \_) \wedge q = L$*).* If we are a follower and we lose connection to the leader, we are in the same situation as if we had recovered from a failure. Thus we will behave in the same way, moving to *state* $= (\textsc{follower}, \textsc{recover})$ and waiting for the $\langle \textsc{Leader} \mid L, n \rangle$ message. Then behave just as in section 6.1.

The complete pseudocode for this version can be seen in algorithm 9.

Correctness The argument for correctness is the same as in section 6.1, considering that session loss in a sub-case of fail-recovery.

Algorithm 9: Sequence Paxos Fail-Recovery&Session Loss – State

Implements: Sequence Consensus
Requires: FIFO Perfect Link, BLE
Algorithm:

1: Π ; /* set of processes */
2: $\Pi_o \leftarrow \Pi - \{self\}$;
3: $state \leftarrow (\text{FOLLOWER}, \perp)$; /* role and phase state */
/* Proposer State */
4: $n_L \leftarrow 0$; /* leader's round number */
5: $promises \leftarrow \emptyset$;
6: $las \leftarrow [0]^{|\Pi|}$; /* length of longest accepted sequence per acceptor */
7: $lds \leftarrow [\perp]^{|\Pi|}$; /* length of longest known decided sequence per acceptor */
8: $propCmds \leftarrow \emptyset$; /* set of commands that need to be appended to the log */
9: $l_c \leftarrow 0$; /* length of longest chosen sequence */
/* Acceptor State */
10: **persistent** $n_{prom} \leftarrow 0$; /* promise not to accept in lower rounds */
11: **persistent** $(n_a, v_a) \leftarrow (0, \langle\rangle)$; /* round number and sequence accepted */
/* Learner State */
12: **persistent** $l_d \leftarrow 0$; /* length of the decided sequence */

Algorithm 9: Sequence Paxos Fail-Recovery&Session Loss – Part 1

```
   /* General Code                                                          */
1: Upon ⟨Leader | L, n⟩
     if self = L ∧ n > n_L then
2:     (n_L, n_prom) ← (n, n);
3:     promises ← {(self, n_a, suffix(v_a, l_d))};
4:     las ← [0]^|Π|;
5:     lds ← [⊥]^|Π|; lds[self] ← l_d;
6:     l_c ← 0;
7:     state ← (leader, prepare);
       foreach p ∈ Π_o do
8:       send ⟨Prepare | n_L, l_d, n_a⟩ to p;
     else
       if state = (_, recover) then
9:       send ⟨PrepareReq⟩ to L;
       else
10:      state ← (follower, state.2);
11: Upon ⟨ConnectionLost | q⟩ s.t. state = (follower, _) ∧ q = L
12:    state ← (follower, recover);
   /* Leader Code                                                           */
13: Upon ⟨PrepareReq⟩ from a s.t. state = (leader, _)
14:    send ⟨Prepare | n_L, l_d, n_a⟩ to a;
15: Upon ⟨Propose | C⟩ s.t. state = (leader, prepare)
16:    propCmds ← propCmds ∪ {C};
17: Upon ⟨Propose | C⟩ s.t. state = (leader, accept)
18:    v_a ← v_a ⊕ C;
19:    las[self] ← |v_a|;
       foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
20:      send ⟨Accept | n_L, C⟩ to p;
21: Upon
     ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, prepare)
22:    promises ← promises ∪ {(a, n', suffix_a)};
23:    lds[a] ← ld_a;
24:    if |promises| = ⌈(|Π|+1)/2⌉ then
25:      suffix ← maxValue(promises);   /* suffix with max n, longest if
           equal */
         /* adopt v_d ++ suffix and append commands                        */
26:      v_a ← prefix(v_a, l_d) ++ suffix ⊕ C forall C ∈ propCmds;
27:      propCmds ← ∅;
28:      las[self] ← |v_a|;
29:      state ← (leader, accept);
         foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
30:        send ⟨AcceptSync | n_L, suffix(v_a, lds[p]), lds[p]⟩ to p;
31: Upon
     ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
32:    lds[a] ← ld_a;
33:    send ⟨AcceptSync | n_L, suffix(v_a, lds[a]), lds[a]⟩ to p;
34:    if l_c ≠ 0 then
35:      send ⟨Decide | l_c, n_L⟩ to a;      /* also inform what got decided
           already */
```

Algorithm 9: Sequence Paxos Fail-Recovery&Session Loss – Part 2

```
   /* Leader Code (continued)                                              */
1: Upon ⟨Accepted | n, l_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
2:     las[a] ← l_a;
3:     M ← {p ∈ Π | las[p] ≠ ⊥ ∧ las[p] ≥ l_a};        /* support set for l_a */
4:     if l_a > l_c ∧ |M| ≥ ⌈(|Π|+1)/2⌉ then
5:         l_c ← l_a;
           foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
6:             send ⟨Decide | l_c, n_L⟩ to p;  /* send length of chosen sequence
                */
   ──────────────────────────────────────────────────────────────────────
   /* Acceptor Code                                                       */
7: Upon ⟨Prepare | n, ld, na_L⟩ from p
8:     if n_prom < n then
9:         n_prom ← n;
10:        state ← (follower, prepare);
11:        suffix ← if n_a ≥ na_L then suffix(v_a, ld) else ⟨⟩;
12:        send ⟨Promise | n, n_a, suffix, l_d⟩ to p;
13: Upon ⟨AcceptSync | n, suffix, ld⟩ from p s.t. state = (follower, prepare)
14:    if n_prom = n then
15:        n_a ← n;
16:        v_a ← prefix(v_a, ld) ++ suffix;
17:        state ← (follower, accept);
18:        send ⟨Accepted | n, |v_a|⟩ to p;
19: Upon ⟨Accept | n, C⟩ from p s.t. state = (follower, accept)
20:    if n_prom = n then
21:        v_a ← v_a ⊕ C;
22:        send ⟨Accepted | n, |v_a|⟩ to p;
   ──────────────────────────────────────────────────────────────────────
   /* Learner Code                                                        */
23: Upon ⟨Decide | l, n⟩ s.t. n = n_prom
24:    while l_d < l do
25:        trigger ⟨Decide | v_a[l_d]⟩;          /* assuming 0-based indexing */
26:        l_d ← l_d + 1;
```

7 Reconfiguration

Having dealt with transient failures, we must now deal with permanent node failures. This can be physical hardware failing to the point that a node has to be permanently removed, but it could also simply be the desire to add or remove nodes to a running system in order to deal with changed load. As *reconfiguration* has many use cases in a practical system, we will separate out the *policy*, that is "why" and "when" we are reconfiguring, from the *mechanism*, i.e. "how" we are reconfiguring, given that the decision has been made already. Policy questions a very application dependent and could be anything from a human operator making decisions in a control room, to a fully automated system making decisions based on some monitoring information. In this section, we will only cover the mechanisms needed to reconfigure a *Sequence Paxos* group.

7.1 Configurations

We will call a "group membership instance" a *configuration* c_i. For example, for four processes $p_1, \ldots, p_4$ we might start in configuration $c_0 = \{p_1, p_2, p_3\}$, but at some point the policy decides to move to a new configuration $c_1 = \{p_1, p_2, p_4\}$. We make no restrictions on the membership in each configuration, that is, in general it might happen that $c_0 \cap c_1 = \emptyset$. Most of the time, though, we will simply replace a single node that is considered failed by the policy.
We model our system such that every configuration c_i is a logically separate instance of *Sequence Paxos*, with its own instance of *Ballot Leader Election*. Each process $p_j \in c_i$ acts as a replica r_{ij} in c_i. Thus, a process p_j may be a replica in multiple configurations at the same time. Reusing the same example as before, we would have configuration $c_0 = \{r_{01}, r_{02}, r_{03}\}$ and later $c_1 = \{r_{11}, r_{12}, r_{14}\}$. In this case, for example, p_1 is part of two configurations in $\{r_{01}, r_{11}\}$.

The RSM executes in a configuration until a reconfiguration event occurs, then it moves to the new configuration. At each process this transitions happens asynchronously, but only a single configuration is (locally) *active* (or *running*) at a time, that is it can *extend* its sequence. A new configuration is considered (globally) *active*, once it has a majority of active members.

In order to mark the end of a configuration c_i, we will issue a special command called the *stop-sign* SS_i. Once a leader p_L proposes a sequence σ_i containing SS_i in configuration c_i, it must be such that SS_i is the last command in σ_i and p_L may not issue a longer sequence in c_i. Once σ_i is *decided* no proposer may ever extend it, thus we call σ_i the *final sequence* of c_i and we call c_i *stopped*.
When the σ_i is decided in c_i by at least one process, the new configuration c_{i+1} can start, as it is guaranteed that σ_i will not change. The stop sign SS_i for configuration c_i contains the all the information necessary to start c_{i+1}. Concretely that is Π_{i+1}, the set of processes in c_{i+1}, i, the *configuration number*, and for each process $p_j \in \Pi_{i+1}$ its *replica identifier* $r_{(i+1)j}$. We want to use σ_i as initial sequence for all replicas in c_{i+1}. There are three cases that we have to deal with at every process $p_j \in \Pi_{i+1}$:

Case 1 ($c_i = c_0$). We are starting the first configuration and there is no previous sequence. The we pick $\langle\rangle$ as the initial sequence on all replicas.

Case 2 ($c_i \neq c_0 \wedge p_j \in \Pi_i$). This process is a replica in both the c_i and c_{I+1}. Once it has locally decided σ_i in the instance for configuration c_i, it will pass σ_i as an initial parameter to the instance for configuration c_{i+1} locally. (Since σ_i is immutable, it could even pass a reference, thus sharing the memory in an implementation that allows such things.)

Case 3 ($c_i \neq c_0 \wedge p_j \notin \Pi_i$). This is a process that does not have the initial sequence locally already. It must fetch it from some other node, either from the old configuration or from a shared persistent storage, before it can start up in the new configuration. This transfer process can be very time-consuming if the log is large. Practically, it is advisable to begin this process in parallel to the old configuration still running and only once the new nodes are close to being caught up, issue the stop-sign command. Additional optimisations such as state compression (snapshotting) and garbage collection on the log and state machine's state are highly recommended (s. section 8).

To make the proposed changes work with our *Sequence Paxos* algorithm we will additionally extend the *round numbers* to contain the *configuration number* as well, such that rounds in higher configurations will always be ordered higher than rounds from older configurations. Thus instead having round number $n = b$ for a ballot number b, we will now have $n = (i, b)$ for a configuration c_i.

Algorithm 10 describes the execution of a replica after it has acquire its initial sequences σ_{i-1}. The startup stage is assumed to be handled by an external component, which starts the replica after obtaining the initial sequence. Given the many and application-dependent options for handling that part of the algorithm, we chose not provide pseudocode for it here.

Correctness Since configurations are totally ordered, our new round numbers that include the configration identifier are also totally ordered across configurations. Furthermore, the acceptor state at the start of a new configuration is the same as the state of the acceptors at the end of the last configuration. In this way we maintain the invariant, that if a sequence v is issued in round $n = (i, b)$, then v is an extension of all sequences chosen in previous rounds $n' \leq n$.

Performance The performance of the proposed approach depends mostly on how efficient the distribution of the final sequences to the members of the next configuration is. During an active configuration the described method has little or no performance impact at all. However, the necessity to keep replicas in old configurations around to catch up recovering members introduces a certain amount of clutter over time, which we will have to deal with at some point.

Algorithm 10: Sequence Paxos Reconfiguration – State&General

Implements: Sequence Consensus
Requires: FIFO Perfect Link, BLE
Algorithm:

c_i ; /* configuration this replica is running in */
Π_i ; /* set of processes in configuration c_i */
$R \leftarrow \{r_{ij} \mid p_j \in \Pi_i\}$; /* set of replicas in configuration c_i */
$R_o \leftarrow R - \{self\}$;
$rself \leftarrow r_{ij}$ **s.t.** $self = p_j \in \Pi_i$; /* our own replica id for this configuration */
σ_{i-1} ; /* the final sequence from the previous configuration or $\langle\rangle$ if $i = 0$ */
$state \leftarrow (\text{FOLLOWER}, \bot)$; /* role and phase state */
/* Proposer State */
$n_L \leftarrow (i, 0)$; /* leader's round number */
$promises \leftarrow \emptyset$;
$las \leftarrow [\ |\sigma_{i-1}|\ ]^{|R|}$; /* length of longest accepted sequence per acceptor */
$lds \leftarrow [\bot]^{|R|}$; /* length of longest known decided sequence per acceptor */
$propCmds \leftarrow \emptyset$; /* set of commands that need to be appended to the log */
$l_c \leftarrow |\sigma_{i-1}|$; /* length of longest chosen sequence */
/* Acceptor State */
persistent $n_{prom} \leftarrow (i, 0)$; /* promise not to accept in lower rounds */
persistent $(n_a, v_a) \leftarrow ((i, 0), \sigma_{i-1})$; /* round number and sequence accepted */
/* Learner State */
persistent $l_d \leftarrow |\sigma_{i-1}|$; /* length of the decided sequence */

/* General Code */
Fun STOPPED()
  return $v_a[l_d] = SS_i$;
Upon $\langle \text{LEADER} \mid L, b \rangle$
  $n \leftarrow (i, b)$;
  if $self = L \wedge n > n_L$ **then**
    $(n_L, n_{prom}) \leftarrow (n, n)$;
    $promises \leftarrow \{(rself, n_a, \text{SUFFIX}(v_a, l_d))\}$;
    $las \leftarrow [|\sigma_{i-1}|]^{|R|}$;
    $lds \leftarrow [\bot]^{|R|}$; $lds[rself] \leftarrow l_d$;
    $l_c \leftarrow |\sigma_{i-1}|$;
    $state \leftarrow (\text{LEADER}, \text{PREPARE})$;
    foreach $r \in R_o$ **do**
      send $\langle \text{PREPARE} \mid n_L, l_d, n_a \rangle$ **to** r;
  else
    if $state = (\_, \text{RECOVER})$ **then**
      send $\langle \text{PREPAREREQ} \rangle$ **to** L;
    else
      $state \leftarrow (\text{FOLLOWER}, state.2)$;
Upon $\langle \text{CONNECTIONLOST} \mid q \rangle$ **s.t.** $state = (\text{FOLLOWER}, \_) \wedge q = L$
  $state \leftarrow (\text{FOLLOWER}, \text{RECOVER})$;

Algorithm 10: Sequence Paxos Reconfiguration – Leader (1)

```
   /* Leader Code                                                          */
1: Upon ⟨PrepareReq⟩ from a s.t. state = (leader, _)
2:     send ⟨Prepare | n_L, l_d, n_a⟩ to a;
3: Upon ⟨Propose | C⟩ s.t. state = (leader, prepare)
4:     propCmds ← propCmds ∪ {C};
5: Upon ⟨Propose | C⟩ s.t. state = (leader, accept) ∧ ¬stopped()
6:     v_a ← v_a ⊕ C;
7:     las[rself] ← |v_a|;
       foreach p ∈ {r ∈ R_0 | lds[r] ≠ ⊥} do
8:         send ⟨Accept | n_L, C⟩ to r;
9: Upon
     ⟨Promise | n, n', suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, prepare)
10:    promises ← promises ∪ {(a, n', suffix_a)};
11:    lds[a] ← ld_a;
12:    if |promises| = ⌈(|Π|+1)/2⌉ then
13:        suffix ← maxValue(promises);   /* suffix with max n, longest if
            equal */
           /* adopt v_d ++ suffix and append commands                       */
14:        v_a ← prefix(v_a, l_d) ++ suffix;
           if SS_i = v_a.last then
15:            propCmds ← ∅;          /* commands will never be decided */
           else
               if SS_i ∈ propCmds then
                   /* Could also just drop other outstanding commands
                      instead of ordering them before SS_i                  */
16:                v_a ← v_a ⊕ C forall C ∈ propCmds − {SS_i};
17:                v_a ← v_a ⊕ SS_i;
               else
18:                v_a ← v_a ⊕ C forall C ∈ propCmds;
19:            las[self] ← |v_a|;
20:            state ← (leader, accept);
           foreach r ∈ {r ∈ R_0 | lds[r] ≠ ⊥ ∧ lds[r] ≠ |v_a|} do
21:            send ⟨AcceptSync | n_L, suffix(v_a, lds[r]), lds[r]⟩ to r;
```

Algorithm 10: Sequence Paxos Reconfiguration – Leader (2)

```
   /* Leader Code (continued)                                          */
1: Upon
     ⟨Promise | n, n′, suffix_a, ld_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
2:   lds[a] ← ld_a;
3:   send ⟨AcceptSync | n_L, suffix(v_a, lds[a]), lds[a]⟩ to a;
4:   if l_c ≠ |σ_{i−1}| then
5:     send ⟨Decide | l_c, n_L⟩ to a;        /* also inform what got decided
         already */
6: Upon ⟨Accepted | n, l_a⟩ from a s.t. n = n_L ∧ state = (leader, accept)
7:   las[a] ← l_a;
8:   M ← {p ∈ Π | las[p] ≠ ⊥ ∧ las[p] ≥ l_a};          /* support set for l_a */
9:   if l_a > l_c ∧ |M| ≥ ⌈(|Π|+1)/2⌉ then
10:    l_c ← l_a;
       foreach p ∈ {p ∈ Π_0 | lds[p] ≠ ⊥} do
11:      send ⟨Decide | l_c, n_L⟩ to p;  /* send length of chosen sequence
           */
```

Algorithm 10: Sequence Paxos Reconfiguration – Acceptor & Learner

```
   /* Acceptor Code                                                          */
1: Upon ⟨Prepare | n, ld, na_L⟩ from p
2:     if n_prom < n then
3:         n_prom ← n;
4:         state ← (follower, prepare);
5:         suffix ← if n_a ≥ na_L then suffix(v_a, ld) else ⟨⟩;
6:         send ⟨Promise | n, n_a, suffix, l_d⟩ to p;
7: Upon ⟨AcceptSync | n, suffix, ld⟩ from p s.t. state = (follower, prepare)
8:     if n_prom = n then
9:         n_a ← n;
10:        v_a ← prefix(v_a, ld) ++ suffix;
11:        state ← (follower, accept);
12:        send ⟨Accepted | n, |v_a|⟩ to p;
13: Upon ⟨Accept | n, C⟩ from p s.t. state = (follower, accept)
14:    if n_prom = n then
15:        v_a ← v_a ⊕ C;
16:        send ⟨Accepted | n, |v_a|⟩ to p;
   /* Learner Code                                                           */
17: Upon ⟨Decide | l, n⟩ s.t. n = n_prom
18:    while l_d < l do
19:        C ← v_a[l_d];                    /* assuming 0-based indexing */
20:        trigger ⟨Decide | C⟩;
21:        l_d ← l_d + 1;
```

8 Garbage Collection

As was alluded to in the last paragraph, we have been silently ignoring a certain build-up of state over the lifetime of systems running the algorithms presented so far. In particular there are two areas, the log v_a and the number of stopped configurations c_i, that continue to grow without bounds throughout the system's lifetime. In this section we will explore some ideas on how to manage these issues in a real deployment.

8.1 Snapshots and Truncating the Log

In the most general case, we can not truncate log, as we may have to catch up a new replica that was added in a new configuration. That is, without any knowledge about how the state machine S on top of the replicated log works, the only way to catch up a new member is to replay the whole log v_d onto S' initial state s_0. However, if S allows us to persist intermediate states s_i, which we shall call *snapshots*, we may in fact truncate the log up to all the commands included in s_i, an then transfer s_i and the remaining log to a new replicate to catch it up.

For the vast majority of RSMs this approach will be orders of magnitude faster than transferring and replaying the whole log. Consider for example a key-value store on top of a replicated log, such that the commands are PUT and GET. The log might contain millions of messages after minutes already, but the state of the store would not include any GET commands and it would only grow in the size of PUT commands on separate keys (or the size of the values, if that is not constant). It is easy to see that for most workloads (which tend to be GET-heavy) the size of any state s_i is going to be orders of magnitude smaller than the size of the decided sequence v_d.

We must however ensure, that every replica has persisted its snapshot before we truncate the log anywhere to make sure that we can still catch up replicas that crashed and are recovering during the period between snapshot and truncation[3]. Explain tradeoff To that end we leave the frequency of snapshots up to the implementation of S, but require the interface to inform the *Sequence Paxos* algorithm when a snapshot is finished locally at p_j by proposing a special $\langle \text{SNAPSHOT} \mid j, k, l_k \rangle$ command where k is the identifier of the snapshot $\dot{s}_k$ and l_k a pointer to the last command in v_d that is included in the state s_{l_k} being captured by $\dot{s}_k$. In general we can truncate v_a up to any position m where we have decided $\langle \text{SNAPSHOT} \mid j, k, l_k \rangle$ with $l_k \geq m$ for all $p_j \in \Pi_i$. However, it is most convenient to deterministically take snapshots at the same point in the log at all replicas, such that we only need to keep track of the largest l_k seen from all replicas. Once the log is truncated, we need to take care to translate offset-based pointers like l_d by the last l_k before doing lookups into v_a, such that that $v_a[l_d]$ before

[3] This isn't technically true, we could take a majority snapshot and if a node recovers too late we could change configuration without changing membership and re-use the mechanism for distributing the initial sequence to catch it up.

truncation becomes $v_a[l_d - l_k]$. Similarly all usages of $|v_a|$ need to be translated to $|v_a| + l_k$.

Additionally, to avoid integer overflows of l_d, for example, we can reset all counters to 0 after reconfiguration, iff we make sure the final sequence is always transferred as a pure snapshot (and not a combination of snapshot and truncated log).

One issue we have to deal with during log truncation has to do with command duplication as we described in section 3.2. Recall that we defined the append $\oplus$ implementation *without duplicates*, such that it would check the log before appending a command C, to see if iC already existed and in such a case skip it. If we are truncating the log, we can not check it for commands anymore, and depending on the RSM implementation it may in fact be impossible to see from a snapshot whether or not a command C was already applied to it. For example, in a key-value store, if the snapshot is the mapping from keys to current values, then even a $\mathrm{PUT}(k, v)$ may have been overwritten in the snapshot already, just making its prior application impossible to deduce. For this reason we must find another way to deduplicate commands, if we are to truncate the log.
If we assume that all clients execute *sequentially*, that is sending only a single command C at a time and waiting for it be acknowledged (possibly resending C occasionally after timeouts) before sending a new command C', we can use the following mechanism for deduplication: The RSM on each server maintains a mapping from a client id p_c to the last command C_c that was submitted by p_c and decided, together with the result of executing C_c on the RSM state S (if it is an operation with a result, such as a GET). Whenever a new command C'_c is from p_c is decided, we check if $C_c = C'_c$ and if so we do not execute it again, but simply send the stored result to p_c. If, on the other hand, $C_c \neq C'_c$ then we execute it on S and store it and the result in the map, replacing C_c. In this way we never execute duplicate commands, as for sequential clients duplicate commands must directly follow each other without a different command occurring in between. It is assumed that clients simply ignore additional responses to commands they consider complete.
When doing snapshots now, we must ensure that during log replay after reconfiguration (or recovery) we also do not apply duplicate commands. To achieve this we must store the state of this client map at the time a snapshot is taken together with snapshot. Then we start from the snapshot we also load its client map and apply commands replayed from the log, as if they were newly incoming commands, that is checking for duplicates and only applying the first instance of each command together with its result.

8.2 Configuration Cleanup

Once a new configuration c_i is started, the previous configuration c_{i-1} is not needed anymore. It seems trivial to just shut down all resources related to it, but the possibility of long-running network partitions makes cleanup of these replicas a surprisingly difficult problem without external input. To see why this is the case, consider the following scenario: In configuration c_1 we have three

replicas $\Pi_1 = \{p_1, p_2, p_3\}$ and after running for a while p_3 is disconnected from the rest. The reconfiguration policy ρ decides that p_3 is to be replaced by a new process p_4 by transitioning to configuration c_2 with $\Pi_2 = \{p_1, p_2, p_4\}$. Eventually SS_1 is decided in c_1 and the final sequence σ_1 is transferred to p_4 (and locally to p_1 and p_2) allowing the replicas in c_2 to start up. Now fast forward a few failures and maybe we are in configuration c_5 with $\Pi_5 = \{p_4, p_5, p_6\}$ and we have shut down (either on purpose or due to failure) the replicas at p_1 and p_2. However, due to the partition we have been unable to inform p_3 so far that it is not needed anymore. Say at this point suddenly p_3 reconnects. Being alone it can't do anything wrong, but it has no idea that its supposed to be shutting down. There is no one left it can talk to, p_1 and p_2 being dead, but it can't just shutdown either, since it might just still be disconnected from the other two. Without some way to get external input p_3 is going to stay in this orphaned state forever.

There are multiple way to deal with this, but in the end they all come down to outsourcing the shutdown decision to some kind of policy ρ. If ρ happens to be a human administrator, they can simply physically go to the disconnected machine p_3 and shut down the process for the orphaned replica. Obviously, this isn't very convenient. A more automated system could make use of infrastructure existing at the data-centre for service discovery (e.g., ARP) or name resolution (e.g., DNS) to allow p_3 to discover another working process such as p_4 to acquire information about what happened to its configuration by inspecting the active log with a client API, for example.

9 Literature

While Paxos was originally presented by Leslie Lamport in [8], the way we describe it in these lecture notes comes from a later paper [9], with some additional ideas borrowed from [3]. The description of sequence consensus, which was originally presented by Lamport in [6], is purposefully aligned with the way that *Raft* [10] is presented, to make it easy to compare the two implementations. Additionally, the ideas for the Sequence Paxos reconfiguration are based on "Stoppable Paxos" [7].

10 Conclusion

We have described how to get from single value Paxos via fail-stop Sequence Paxos, fail-recovery Sequence Paxos, to a reconfigurable implementation of the Sequence Consensus abstraction that works in the fail-recovery model. We have also discussed systems issues like garbage collection, snapshots and log truncations, and state transfers. The final algorithm we presented is ready to be be used as is, or adapted to a particular RSM implementation for efficiency.

References

1. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 335–350. USENIX Association (2006)
2. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to reliable and secure distributed programming. Springer Science & Business Media (2011)
3. De Prisco, R., Lampson, B., Lynch, N.: Revisiting the paxos algorithm. In: International Workshop on Distributed Algorithms. pp. 111–125. Springer (1997)
4. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX annual technical conference. vol. 8. Boston, MA, USA (2010)
5. Kroll, L.: Load balancing in a distributed storage system for big and small data (2013)
6. Lamport, L.: Generalized consensus and paxos (2005)
7. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News **41**(1), 63–73 (2010)
8. Lamport, L., et al.: The part-time parliament. ACM Transactions on Computer systems **16**(2), 133–169 (1998)
9. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
10. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). pp. 305–319 (2014)