University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Kamil Braun
Student no. 346840

LPaxos: a fault-tolerant distributed algorithm for building linearizable services without replicated logs

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Informatics

Warsaw, December 2020

Abstract

A general idea employed in fault-tolerant distributed systems is replication — keeping redundant copies of the same piece of information. Popular algorithms focus around replicating a log of commands. Replicas deterministically apply the commands in the same order, in effect going through the same sequence of states and arriving at the same results. This technique is often used to build strongly consistent (linearizable) services. Unfortunately, replicated logs require complex handling that obscures the algorithms and makes systems hard to implement. In this thesis we research an alternative approach, drawing inspiration from ideas used in eventually consistent systems, but using consensus as a building block to achieve strong consistency. We present an algorithm for building fault-tolerant linearizable services that is not based around replicated logs, prove its correctness, and use it to build a distributed key-value store which provides strictly serializable transactions. The absence of a distributed log makes the algorithm relatively easy to understand and implement.

Keywords

Paxos, consensus, state machine replication, linearizability, distributed services, fault-tolerance, key-value store, strong consistency

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Computer systems organization — Reliability
Theory of computation — Distributed algorithms

Tytuł pracy w języku polskim

LPaxos: odporny na awarie rozproszony algorytm do tworzenia linearyzowalnych serwisów bez replikowanych dzienników

Contents

Chapter 1

Introduction

Distributed systems are often structured around *clients* and *services*. A service provides a certain set of features (say, a database) and consists of one or more *servers*; clients communicate with services over the network by sending *requests* (say, database transactions) to the servers and receiving responses.

The service may be running on a single server or on multiple servers that communicate with each other. The single-server option might be easier to implement, but it doesn't provide much *fault-tolerance* — if the server fails, the service becomes unavailable. Thus modern engineers are struggling to correctly implement *distributed services*. This turns out to be a highly non-trivial task. The mere notion of "correctness" is not easy to formally define in the context of such services.

A general method for implementing fault-tolerant distributed services is the state machine approach [12, 20]. First, the engineer specifies the service as a sequential algorithm which, given a sequence of inputs, produces a sequence of outputs. The engineer uses the language of (deterministic) *state machines* to express the algorithm. Then the engineer uses a *State Machine Replication* (SMR) protocol to run copies of the state machine on a set of servers.

This replication process is performed in such a way that the clients of the service get the illusion of working with a non-distributed service: each client sends a command to the service and obtains an output; the inputs and corresponding outputs can be ordered so that the obtained sequence appears as if it was produced simply by the sequential algorithm running on a single server (it is *linearizable* [7]). If the replication algorithm is advanced enough, it may provide a high degree of fault tolerance. For example, it may ensure availability of the service as long as at least a majority of servers are still up and running (given certain assumptions about the types of possible server failures).

Probably the most popular SMR algorithms used today are *Paxos* [13] and *Raft* [17]. The algorithms are quite similar [8] and both focus on solving the problem of *consensus*, where a set of servers agree on something; in this case, they agree on the sequence of commands for the state machine. As new commands are being agreed on and appended to the resulting *log of commands*, each server feeds the commands to its own copy of the state machine. The properties of consensus ensure that each server observes the same sequence, so each copy of the state machine is given the same commands in the same order. Because the state machine is deterministic (a

necessary assumption), each copy will perform the same transitions and return the same sequence of outputs.

Implementing these algorithms is not an easy task. One reason is that the log of commands requires careful bookkeeping. The algorithms don't require the replicas (copies of the state machine) to apply commands in synchrony — some of the replicas may "lag behind". It is possible that some of the replicas didn't apply any commands at all even though the service has been answering to client requests for days, e.g. because these replicas were down. However, because the subset of replicas that are currently available might change, they must have the possibility to catch up: apply the commands they've been missing.

Thus the entire log of commands must be stored so that lagging replicas can learn any of the commands agreed upon at any time. But if that is the case the log can potentially grow endlessly. With services that process thousands of requests per second the servers would quickly run out of disk space. Hence any practical implementation must provide a *truncation* mechanism, where a prefix of the log is forgotten after the system determines that a sufficient number of replicas have applied the commands from that prefix.

But if the log is truncated, what happens to the replicas that failed to apply the now forgotten commands? These replicas must have another way of recovering the recent state. This is usually provided by a *snapshotting* mechanism, where each replica stores a *snapshot* — a copy of the state machine saved on disk corresponding to some prefix of the entire log. Each replica thus maintains two copies of the state machine: one residing in memory which is being frequently updated using the latest learned commands, and another — the snapshot — residing on disk, which is updated less often, used when the replica recovers after restarting and to catch up lagging replicas which cannot use the log of commands since it has been truncated. Snapshotting must be coordinated with truncation to ensure that enough replicas make a snapshot that includes the commands being truncated before they are lost unrecoverably. The process of snapshotting and truncating the log is called *log compaction*.

Even if the engineer understands the core consensus and log replication algorithm, which is by itself not trivial, they must also take care of the other mechanisms — such as log compaction — which further complicates the implementation of the system. For example, the Raft thesis [17] has an entire chapter dedicated to the problem of log compaction which is almost as long as the chapter describing the core Raft algorithm. The formal specification of Raft given in the appendix does not take log compaction into account.

A recent paper introduced CASPaxos [19], a new algorithm that achieves the same effect as the SMR algorithms discussed above but doesn't require maintaing a replicated log of commands. The paper comes with a proof-of-concept implementation of a key-value store based on the algorithm. Both the algorithm and the implementation appear to be exceptionally simple.

Unfortunately, to handle each request, CASPaxos requires the replicas to send the entire state of the replicated state machine over the network. Thus, as the paper itself admits, the algorithm is impractical for state machines with large state. The key-value store deals with this problem by maintaining separate independent state

machines, one for each key — each state machine representing a read-write register. This leads to an efficient implementation assuming that the value stored under each key is small (storage-wise), e.g. an 8-byte integer. However, since the state machines are independent, the system doesn't provide atomic multi-key operations. One could deal with this problem by implementing a transaction mechanism on top of the set of independent registers, but that introduces a completely new set of problems.

Can we find a protocol that efficiently replicates large state, without having to send the entire state between servers on each request, that doesn't maintain a distributed log of commands, and in effect is much easier to implement?

The author of this thesis wanted to give a positive answer to this question, and the thesis is the result.

In standard consistent replication approaches, each replica stores a full copy of the state corresponding to some prefix of a sequence of commands which was agreed on using a consensus algorithm. The logless-replication solution presented in this work is based around the idea of storing *partial information* about the state. We start our discussion by proposing an abstraction — the *patch machine* — that formalizes this idea (section 2.2), which we compare to the commonly used state machines (section 2.1). While state machines, given a command and a state, produce an output and the next state, patch machines produce an output and a *patch* that modifies the state. The patch can be *merged* with the current state to obtain the next state. However, these patches can also be merged together out of order and as we will see, if certain conditions are satisfied, the end result will be the same as if we applied them in the same order as they were obtained from the provided commands.

After introducing patch machines and discussing some examples, we introduce *LPaxos*, a fault-tolerant distributed algorithm for replicating patch machines that does not use a replicated log of commands (chapter 3). The algorithm uses a variation of a consensus protocol — the *multi-decree Synod* — which was originally used in the standard Paxos SMR algorithm.

We give a formal specification of LPaxos (section 3.4) and a complete proof of correctness (section 3.5). In particular, we prove that it can be used to build a linearizable service (section 3.5.4).

Finally, we discuss the design an implementation of *LattiStore*, a distributed key-value store providing strictly serializable [7] multi-key transactions (chapter 4). The possible transactions are arbitrary functions of the stored set of key-value pairs, not just reads or writes.

Chapter 2

Specifying services

In order to build a fault-tolerant distributed service we can use a replication algorithm. But *what* do we replicate? This is the first thing that must be determined. In the state machine approach the entity being replicated is a state machine. The language of state machines provides a formal framework for specifying services and is commonly used when implementing distributed systems.

In this work we focus on an alternative abstraction which we refer to as *patch machine*s. They are very similar to state machines and serve the same purpose — they can be used to formally specify a service and can be plugged into a replication algorithm (we describe one in the next chapter) in order to build a fault-tolerant system. Before we describe patch machines, however, we give a formal treatment of state machines so we can later compare the two abstractions. We also discuss examples of using state machines to express some useful problems and corresponding examples for patch machines.

2.1. State Machines

There are many equivalent definitions of state machines. One that the author finds elegant is presented in this section [11]. Here we deal only with *deterministic* state machines where the transition relation is a function, meaning that from the given input and state we can arrive at exactly one output and new state.

Definition 2.1.1. A (deterministic) **state machine** is a tuple (S, I, O, δ, s_0), where:

- S, I, O are sets, called the *set of states*, the *set of inputs* (or *commands*), and the *set of outputs*, respectively.
- δ is a function $\delta : S \times I \to S \times O$, called the *transition function*,
- s_0 is a state $s_0 \in S$ called the *initial state*.

□

State machines by themselves don't "do" anything — they just are. But we usually think of state machines as objects that can be "executed": given a sequence of inputs, the machine performs a sequence of steps, producing a sequence of outputs as the result. The following definition formalizes this idea.

Definition 2.1.2. An **execution** of a state machine (S, I, O, δ, s_0) is a sequence

$$s_0, i_1, (s_1, o_1), i_2, (s_2, o_2), i_3, (s_3, o_3), \ldots$$

where:

- $s_j \in S$ for $j = 0, 1, \ldots,$
- $i_j \in I$ for $j = 1, 2, \ldots,$
- $o_j \in O$ for $j = 1, 2, \ldots,$

and $(s_{j+1}, o_{j+1}) = \delta(s_j, i_{j+1})$ for $j = 0, 1, \ldots$

The sequence might be infinite or finite. If it's finite, it either ends on s_0 or on (s_j, o_j) for some j (it doesn't end on an input). □

The definition should be obvious: an execution starts in the initial state s_0; at this point, the state machine did not produce any output. Each time an input i_{j+1} is provided, the machine performs a transition from the current state s_j to the next state s_{j+1} and provides an output o_{j+1} as defined by the transition function δ.

We now give some examples of formally expressing services using state machines.

Example 2.1.1 (Read-write register)**.** Consider the following informal description of a service:

> The service provides access to an integer. The value of the integer can be modified using a "write" command and read from using a "read" command.

The following state machine $\mathcal{M}$ is a possible formalization of this description. Let $\mathcal{M} = (S, I, O, s_0, \delta)$, where:

- $S = \mathbb{Z}$.
- $I = \{read\} \cup \{write(x) : x \in \mathbb{Z}\}$.
- $O = \{ret(x) : x \in \mathbb{Z}\} \cup \{ack\}$.
- $s_0 = 0$.
- $\delta(s, read) = (s, ret(s))$.
- $\delta(s, write(x)) = (x, ack)$.

Example execution:

$$0, read, (0, ret(0)), write(42), (42, ack), read, (42, ret(42)).$$

The *read* command is used to retrieve the current state of the register (and leave it unmodified), and $write(x)$ is used to overwrite the state with x. The response to *read* is $ret(x)$, where x is the state prior to executing the command, and the response to $write(x)$ is a simple acknowledgement message, ack. □

Example 2.1.2 (CAS register)**.** Below is a definition of a *CAS register*: an object storing an integer that supports an atomic *compare-and-set* command.

- $S = \mathbb{Z}$.
- $I = \{cas(x, y) : x, y \in \mathbb{Z}\}$.
- $O = \{ok, fail\}$.
- $s_0 = 0$.
- $\delta(s, cas(x, y)) = \begin{cases} (y, ok) & \text{if } s = x, \\ (s, fail) & \text{otherwise.} \end{cases}$

Example execution:

$$0, cas(1,2), (0, fail), cas(0,42), (42, ok).$$

If the current state is x, $cas(x, y)$ changes the state to y and returns ok. Otherwise it leaves the state intact and returns $fail$. □

An easy exercise is to define a state machine representing a register that supports all operations from the above examples: read, write, and compare-and-set.

Example 2.1.3 (Key-value store)**.** This is a simple *key-value store* that holds a mapping from some set of keys to some set of values. The supported commands are functions that read the state and overwrite the values under a subset of keys (they can use the result of the read to decide what the new values will be). The output of each command is the state of the store (all key-value pairs) prior to applying the command. Returning the entire state on each output might not be practical but we do it here for simplicity.

Let K be a set of keys and V be a non-empty set of values. The set of states is the set of all partial functions from K to V. The initial state is the empty function. For a partial function s, we will use the notation $dom(s)$ to denote the domain of definition of s.

- $S = K \dashrightarrow V$.
- $I = (K \dashrightarrow V) \to (K \dashrightarrow V)$.
- $O = (K \dashrightarrow V)$.
- $s_0 = \emptyset$.
- $\delta(s, f) = (s', s)$, where $dom(s') = dom(s) \cup dom(f(s))$ and

$$s' = \begin{cases} k \mapsto f(s)(k) & \text{for } k \in dom(f(s)), \\ k \mapsto s(k) & \text{for } k \in dom(s) \setminus dom(f(s)). \end{cases}$$

For $s : K \dashrightarrow V$ and $k \in K$, let

$$get(s, k) = \begin{cases} s(k), \text{ if } k \in dom(s) \\ 0 \text{ otherwise.} \end{cases}$$

Suppose $K = V = \mathbb{Z}$. Let $f_1, f_2 \in I$ be defined as follows:

$$f_1(s) = \begin{cases} 1 \mapsto get(s, 0) + 1 \\ 2 \mapsto get(s, 1) + 2 \end{cases}$$

$$f_2(s) = \begin{cases} 2 \mapsto get(s, 1) + 3 \\ 3 \mapsto get(s, 2) + 4. \end{cases}$$

An example execution of the machine is:

$$s_0, f_1, (s_1, o_1), f_2, (s_2, o_2),$$

where:

$$
\begin{aligned}
s_0 &= \emptyset \\
s_1 &= \begin{cases} 1 \mapsto 1 \\ 2 \mapsto 2 \end{cases} & o_1 &= \emptyset \\
s_2 &= \begin{cases} 1 \mapsto 1 \\ 2 \mapsto 4 \\ 3 \mapsto 6 \end{cases} & o_2 &= s_1.
\end{aligned}
$$

□

Suppose we are given a state machine definition and are asked to implement a service. We could do it as follows. A server stores the current state of the machine, initially s_0. A single process running on the server sequentially handles requests from clients. After receiving a request and checking that it contains a valid input $i \in I$, the process uses δ and the currently stored state s to produce $(s', o) = \delta(s, i)$, the next state and the output. It overwrites the current state with s' and returns o as the response to the client who sent the request.

This is a very high-level description, and a lot of details are purposefully left undiscussed. But we can immediately see one problem with this approach — the process running on a single server is a single point of failure. If it crashes, or if something bad happens to the server, the service becomes unavailable.

One way to deal with that is to use a state machine replication algorithm, which maintains multiple processes running on different servers, each providing access to a copy of the state machine; the algorithm ensures that each copy receives the same sequence of inputs, and each time a new input is appended to the sequence, it uses one of the copies — one that has already consumed all previous inputs — to produce an output and return it to the client who sent the input.

In this work we discuss a different approach based on patch machines. The definition of a patch machine might seem a bit more complex than that of a state machine at first glance, but the effort pays off by enabling a relatively simple replication algorithm. With appropriate modeling of real-world problems using this abstraction engineers may be able to quite easily implement efficient and fault-tolerant distributed services.

The definition of state machine and examples above were given for comparison; we will see that patch machines are in a sense equivalent to state machines, i.e. they have the same power of expression, hence any problem that can be expressed using a state machine can also be expressed using a patch machine and vice-versa.

2.2. Patch Machines

Before we define patch machines let us provide some motivation.

In Paxos, the consensus algorithm requires a majority of participating servers to be alive to choose a new entry for the replicated log, so a majority of running servers is a minimal requirement for the algorithm to achieve progress (*choosing an entry* here means that the processes reach consensus about what the entry will be). The chosen entries are state machine commands and we want all the copies of the machine to

eventually apply all commands. Therefore, after chosing the nth entry, if we want to proceed without requiring all servers to be currently available, we must remember that entry even after the $(n+1)$th entry has been chosen, so replicas that still haven't applied the nth entry can eventually learn about it and catch up (the alternative would be to obtain a snapshot, but we usually want to minimize the number of snapshot transfers as they are expensive). Wouldn't it be convenient if we could drop the nth entry immediately after only a majority replicas have applied it?

The idea used here is to drop the requirement of every replica applying all chosen entries in the same order. Instead of choosing state machine commands, we will be choosing patches that can be merged together into larger patches; each patch represents partial information about the state. The patches we will choose in the replication algorithm will not have to be applied to all replicas, but only to a majority of them, and as soon as this is done the patch can be forgotten — we don't have to keep it when choosing the next patch. This way we do not end up with a log of commands and do not have to worry about log compaction.

For state machines, the next state is calculated from the current state using a command. Similarily, for patch machines, we calculate the next patch from the current state using a command. To do this we must be able to obtain the *current state*, which is defined to be the result of applying all previously chosen patches in order starting from the initial state. However, we don't necessarily store the current state on any replica — instead, pieces of the state are distributed across them, each replica having applied only a subset of the previously chosen patches. The trick is that we can recover the current state by *merging* the partial states stored on replicas. To do that, we contact a majority of replicas. Since we've ensured that each previous patch is applied to some majority of replicas, and since any two majorities intersect, each previous patch is applied by at least one of the replicas in the contacted majority. Together the majority has all the information required to recover the current state.

We will come back to this discussion in the next chapter.

Definition 2.2.1. A **patch machine** is a tuple $(P, I, O, \delta, p_0, \sqcup)$, where:

- P, I, O are sets, called the *set of patches*, the *set of inputs* (or *commands*), and the *set of outputs*, respectively.
- δ is a function $\delta : P \times I \to P \times O$, called the *transition function*,
- p_0 is a patch $p_0 \in P$ called the *initial patch* (and also the *initial state*),
- $\sqcup$ is a binary operation on patches, $\sqcup : P \times P \to P$, called *merge*,

such that:

- $\forall p_1, p_2, p_3 \in P : (p_1 \sqcup p_2) \sqcup p_3 = p_1 \sqcup (p_2 \sqcup p_3)$ (merging is *associative*),
- $\forall p \in P : p \sqcup p = p$ (merging is *idempotent*),
- Suppose $p_1, p_2, \ldots, p_n$ for $n \geq 0$ is a sequence of patches satisfying

  $$\forall m \in \{0, \ldots, n-1\} : \exists o \in O, i \in I : (p_{m+1}, o) = \delta(p_0 \sqcup p_1 \sqcup \cdots \sqcup p_m, i).$$

  Then, for each $0 \leq i < j \leq n$,

  $$p_i \sqcup p_j = p_j \sqcup p_i.$$

We will refer to this property as *commutativity of merging for descendant patches*.

□

That last property might seem weird at the moment, but the motivation should become clearer after discussing a couple of examples. For clarity of exposition we will give a name to sequences of patches mentioned in the property.

Definition 2.2.2. Let $(P, I, O, \delta, p_0, \sqcup)$ be a patch machine. A sequence of patches $p_1, p_2, \ldots, p_n$ satisfying

$$\forall m \in \{0, \ldots, n-1\} : \exists o \in O, i \in I : (p_{m+1}, o) = \delta(p_0 \sqcup p_1 \sqcup \cdots \sqcup p_m, i)$$

is called a **descendant sequence of patches**. □

Hence the last property in the definition of patch machine says that for any two patches in a descendant sequence of patches, $\sqcup$ commutes for these patches.

The set of patches P in the definition of patch machine looks similar to the set of states S in a state machine. Think of a patch as representing "partial information" about the machine's state, with some patches having "full" information (e.g. p_0 has full information about the initial state).

Definition 2.2.3. An **execution** of a patch machine $(P, I, O, \delta, p_0, \sqcup)$ is a sequence

$$s_0, i_1, (s_1, o_1), i_2, (s_2, o_2), i_3, (s_3, o_3), \ldots$$

where:

- $s_j \in P$ for $j = 0, 1, \ldots$ and $s_0 = p_0$,
- $i_j \in I$ and $o_j \in O$ for $j = 1, 2, \ldots$,
- $s_{j+1} = s_j \sqcup p_{j+1}$, where $(p_{j+1}, o_{j+1}) = \delta(s_j, i_{j+1})$ for $j = 0, 1, \ldots$

The sequence might be infinite or finite. If it's finite, it either ends on s_0 or on (s_j, o_j) for some j (it doesn't end on an input). □

A patch machine execution differs from a state machine execution in that δ doesn't provide the next state; it provides a patch that must be applied (using the merge operation) to the previous state to obtain the next state. That is, to obtain the next state from state s_j and input i_{j+1} we first compute $(p_{j+1}, o_{j+1}) = \delta(s_j, i_{j+1})$, and then use the obtained patch to compute $s_{j+1} = s_j \sqcup p_{j+1}$.

Observe that an execution $s_0, i_1, (s_1, o_1), \ldots, i_n, (s_n, o_n)$ natually defines a descendant sequence of patches $p_1, \ldots, p_n$ by $(p_{j+1}, o_{j+1}) = \delta(s_j, i_{j+1})$. And vice-versa: given a descendant sequence of patches $p_1, \ldots, p_n$, define $s_j = p_0 \sqcup \cdots \sqcup p_j$ and let o_{j+1}, i_{j+1} be such that $(p_{j+1}, o_{j+1}) = \delta(s_j, i_{j+1})$ (these o_{j+1}, i_{j+1} exist by the definition of a descendant sequence). Then $s_0, i_1, (s_1, o_1), \ldots, i_n, (s_n, o_n)$ is an execution.

Example 2.2.1 (CAS register)**.** This is a patch machine version of example 2.1.2. A patch is represented by a pair: the first element represents the value of the register, and the second element represents a timestamp, used to discern which value is "newer" when merging two patches.

Let $\mathcal{M} = (P, I, O, p_0, \delta, \sqcup)$, where:

- $P = \mathbb{Z} \times \mathbb{N}$.
- $I = \{cas(x, y) : x, y \in \mathbb{Z}\}$.
- $O = \{ok, fail\}$.
- $p_0 = (0, 0)$.
- $\delta((s, t), cas(x, y)) = \begin{cases} ((y, t+1), ok) & \text{if } s = x, \\ ((s, t+1), fail) & \text{otherwise.} \end{cases}$
- $(x_1, t_1) \sqcup (x_2, t_2) = \begin{cases} (x_1, t_1) & \text{if } t_1 > t_2, \\ (x_2, t_2) & \text{otherwise.} \end{cases}$

It is easy, albeit tedious, to check that the $\sqcup$ axioms hold:

- associativity: let $p_1 = (x_1, t_1), p_2 = (x_2, t_2), p_3 = (x_3, t_3)$. Then it is easy to see that the following implications hold:
  - $t_1 > t_2 \wedge t_1 > t_3 \Rightarrow (p_1 \sqcup p_2) \sqcup p_3 = p_1 = p_1 \sqcup (p_2 \sqcup p_3)$,
  - $t_1 > t_2 \wedge t_1 \leq t_3 \Rightarrow (p_1 \sqcup p_2) \sqcup p_3 = p_3 = p_1 \sqcup (p_2 \sqcup p_3)$,
  - $t_1 \leq t_2 \wedge t_2 > t_3 \Rightarrow (p_1 \sqcup p_2) \sqcup p_3 = p_2 = p_1 \sqcup (p_2 \sqcup p_3)$,
  - $t_1 \leq t_2 \wedge t_2 \leq t_3 \Rightarrow (p_1 \sqcup p_2) \sqcup p_3 = p_3 = p_1 \sqcup (p_2 \sqcup p_3)$.

  Since one of the conditions from the left sides of the above implications must be true, we get $(p_1 \sqcup p_2) \sqcup p_3 = p_1 \sqcup (p_2 \sqcup p_3)$.
- Idempotency is obvious.
- Commutativity for descendant patches: let $s_0, i_1, (s_1, o_1), \ldots, i_n, (s_n, o_n)$ be an execution and let $(p_j, o_j) = \delta(s_{j-1}, i_j)$. Let $(x_j, t_j) = p_j$. Then it is easy to see directly from the definition of δ that for $i < j$, $t_i < t_j$ (in fact: $t_j = t_i + j - i$), so by definition of $\sqcup$, $p_i \sqcup p_j = p_j = p_j \sqcup p_i$.

Therefore $\mathcal{M}$ is a patch machine.

Example execution:

$$(0, 0), cas(1, 2), ((0, 1), fail), cas(0, 42), ((42, 2), ok).$$

We've defined δ so that the timestamp is increased on each operation, but the following definition would also give a valid patch machine:

$$\delta((s, t), cas(x, y)) = \begin{cases} ((y, t+1), ok) & \text{if } s = x, \\ ((s, t), fail) & \text{otherwise.} \end{cases}$$

In this case the proof of commutativity of $\sqcup$ is different, but still easy; simply observe that if two patches in a descendant sequence have the same timestamp, they are equal.

□

Many examples can be produced by considering cases where $(P, \sqcup)$ forms a *semi-lattice*, i.e. where the $\sqcup$ operation is commutative for any two patches, not only descendant ones.

In fact, semilattices are the main motivation between the definition of patch machines, but the author found himself needing something a bit more general: there are valid examples, such as the CAS register example above, where $\sqcup$ doesn't always commute. Indeed, let $i = cas(0,1)$, $i' = cas(0,2)$, $(p, o) = \delta(p_0, i)$, and $(p', o') = \delta(p_0, i')$. Then $p = (1,1)$ and $p' = (2,1)$, hence $p \sqcup p' = p'$, but $p' \sqcup p = p \neq p'$. Intuitively, given two different inputs, starting from the same state, the patch machine can end up in "incompatible" states.

Example 2.2.2 (Compare-and-add grow-set)**.** This artificial example represents a set that can only grow using a "compare-and-add" operation $add(x, y)$: if the element x is not in the set, we add a new element y, and leave the set unmodified otherwise.

- $P = \mathbf{P}(\mathbb{Z})$[1].
- $I = \{add(x, y) : x, y \in \mathbb{Z}\}$.
- $O = \{ok, fail\}$.
- $p_0 = \emptyset$.
- $\delta(s, add(x, y)) = \begin{cases} (\{y\}, ok) & \text{if } x \notin s, \\ (\emptyset, fail) & \text{otherwise,} \end{cases}$
- $\sqcup = \cup$ (the set sum operation).

Clearly $\sqcup = \cup$ is associative, idempotent, and commutative (for any two sets).

Example execution:

$$\emptyset, add(0,0), (\{0\}, ok), add(0,1), (\{0\}, fail), add(1,42), (\{0,42\}, ok).$$

□

The next example is very close to what is used in LattiStore, a key-value store implementation discussed in section 4.3.

Example 2.2.3 (Key-value store)**.** This is a patch machine version of example 2.1.3. Let K be a set of keys and V be a set of values.

- $P = (K \dashrightarrow V \times \mathbb{N}) \times \mathbb{N}$. Thus, each patch $p \in P$ is a pair containing a partial function from K to $V \times \mathbb{N}$ and a natural number.

  For us partial functions from A to B, for any two sets A and B, will be viewed as sets of pairs (a, b), $a \in A$, $b \in B$ such that for every $a \in A$ there is at most one b with (a, b) in the set. Furthermore, a pair with a nested pair $((a, b), c)$ will be simply viewed as a 3-tuple (a, b, c).

  With these considerations, each patch $p \in P$ is a pair (s, N), where $N \in \mathbb{N}$ and s is a set of tuples (k, v, n) with $k \in K$, $v \in V$, and $n \in \mathbb{N}$, such that for each $k \in K$ there is at most one $v \in V$ and $n \in \mathbb{N}$ such that $(k, v, n) \in s$. That is:

  $$(k, v, n) \in s \Leftrightarrow k \in dom(s) \wedge (v, n) = s(k).$$

  Given such a tuple (k, v, n), we will call k the *key*, v the *value*, and n the *version* of this tuple.

[1] **P** denotes the power set operation, so a patch is a subset of $\mathbb{Z}$.

The number $n \in \mathbb{N}$ stored together with each $v \in V$ serves as a "local version" of the value stored under a given key $k \in K$. These local versions will be used to decide which value to take for each key when merging two patches, as we will soon see in the definition of $\sqcup$.

The second element of the pair p is a "global version" of the patch. It will be used to produce the local versions of the next patch calculated by δ.

- $I = (K \dashrightarrow V) \to (K \dashrightarrow V)$.
- $O = K \dashrightarrow V$.
- $p_0 = (\emptyset, 0)$.
- To define δ we'll start with a couple of auxiliary definitions.

  Let $unver : (K \dashrightarrow V \times \mathbb{N}) \to (K \dashrightarrow V)$ be given by

  $$unver(s) = \{(k, v) : (k, v, n) \in s \text{ for some } n \in \mathbb{N}\}.$$

  In other words, *unver* drops the version from each key-value-version tuple.

  Similarily, define $ver : (K \dashrightarrow V) \times \mathbb{N} \to (K \dashrightarrow V \times \mathbb{N})$ by

  $$ver(s, N) = \{(k, v, N) : (k, v) \in s\}.$$

  In other words, *ver* attaches the given version to each key-value pair.

  Then we can define δ:

  $$\delta((s, N), f) = ((ver(f(s'), N + 1), N + 1), s'), \text{ where } s' = unver(s).$$

  Given a patch (s, N), where s is a set of key-value-version tuples and $N \in \mathbb{N}$, and an input f, δ drops all versions from s, applies f to the resulting set of key-value pairs to obtain another set of key-value pairs, and then attaches $N + 1$ to each key-value pair in the new set; the next patch is the resulting set of key-value-version triples with $N+1$ as the global version. To obtain the output we simply drop versions from s.

- The merge operation is given by

  $$(s_1, n_1) \sqcup (s_2, n_2) = (s_1 \sqcup s_2, max(n_1, n_2)),$$

  where $s_1 \sqcup s_2$ for $s_1, s_2 : K \dashrightarrow V \times \mathbb{N}$ is given by

  $$\begin{aligned} s_1 \sqcup s_2 = \{&(k, v, n) \in s_1 \cup s_2 : \\ &((k, v, n) \in s_1 \wedge \forall (k, v', n') \in s_2 : n > n') \\ &\vee ((k, v, n) \in s_2 \wedge \forall (k, v', n') \in s_1 : n \geq n')\}. \end{aligned}$$

  The $max(n_1, n_2)$ should require no explanation, but perhaps the set $s_1 \sqcup s_2$ requires some deciphering. For each $k \in K$:

  - if there is no tuple in s_1 nor s_2 with key k, then there is no such tuple in $s_1 \sqcup s_2$;
  - otherwise, we take the tuple with key k from $s_1 \cup s_2$ with the higher version. If there is a tie (both s_1 and s_2 have a tuple with key k and the same version), we take the tuple from s_2.

As defined above, $s_1 \sqcup s_2$ is indeed a partial function from K to $V \times \mathbb{N}$: for each $t = (k, v, n) \in s_1 \sqcup s_2$, either $t \in s_1$ or $t \in s_2$, but not $t \in s_1 \cap s_2$, so there are no two different tuples with the same key in $s_1 \sqcup s_2$. Thus $(s_1, n_1) \sqcup (s_2, n_2) \in P$.

Proving properties of $\sqcup$ is very similar to what we've seen in example 2.2.1, except that now we're dealing with multiple $(value, version)$ pairs, at most one for each key, while in the referred example we had only one such pair. Furthermore, some keys might have a pair in some of the merged patches but not all of them, which requires a bit of technical handling.

- Let $p_1 = (s_1, m_1), p_2 = (s_2, m_2), p_3 = (s_3, m_3) \in P$.

  We have $(p_1 \sqcup p_2) \sqcup p_3 = ((s_1 \sqcup s_2) \sqcup s_3, max(m_1, m_2, m_3))$ and $p_1 \sqcup (p_2 \sqcup p_3) = (s_1 \sqcup (s_2 \sqcup s_3), max(m_1, m_2, m_3))$.

  Let $L = (s_1 \sqcup s_2) \sqcup s_3$ and $R = s_1 \sqcup (s_2 \sqcup s_3)$. We just have to prove that $L = R$.

  Let $k \in K$. First observe that if none of s_1, s_2, s_3 have a tuple with key k, then neither does $s_1 \sqcup s_2$ nor $s_2 \sqcup s_3$, hence neither does L nor R. Suppose then that some of them have a tuple with key k.

  For $i = 1, 2, 3$, if $k \in dom(s_i)$, then let $(v_i, n_i) = s_i(k)$ (i.e. $(k, v_i, n_i) \in s_i$). Otherwise let $(v_i, n_i) = (\bot, -1)$, where $\bot$ is any value. Let $t_i = (k_i, v_i, n_i)$.

  It is easy to see that the following implications are true:

  - $n_1 > n_2 \wedge n_1 > n_3 \Rightarrow t_1 \in L \wedge t_1 \in R$.
  - $n_1 > n_2 \wedge n_1 \leq n_3 \Rightarrow t_3 \in L \wedge t_3 \in R$.
  - $n_1 \leq n_2 \wedge n_2 > n_3 \Rightarrow t_2 \in L \wedge t_2 \in R$.
  - $n_1 \leq n_2 \wedge n_2 \leq n_3 \Rightarrow t_3 \in L \wedge t_3 \in R$.

  Only one of t_1, t_2, or t_3 can be in L (since the tuples have the same key), and the same holds for R. In each of the above four cases, L and R have the same t_i. One of the cases is true, so L and R have the same t_i.

  Thus, for each $k \in K$, we have shown that either $k \notin dom(L)$ and $k \notin dom(R)$, or $k \in dom(L) \cap dom(R)$ and $L(k) = R(k)$. Hence $L = R$.

- Idempotency is obvious.

- Let $p_1 = (s_1, m_1), \ldots, p_n = (s_n, m_n)$ be a descendant sequence. It can be easily seen from the definition of δ and simple induction that $m_i = i$ for $0 < i \leq n$.

  Let $0 \leq i < j \leq n$. For each $k \in K$ consider the four possible cases:

  1. $(k, v_1, n_1) \in s_i$ for some v_1, n_1 and $(k, v_2, n_2) \in s_j$ for some v_2, n_2.
  2. $(k, v, n) \in s_i$ for some v, n but there is no tuple in s_j with key k.
  3. $(k, v, n) \in s_j$ for some v, n but there is no tuple in s_i with key k.
  4. Neither s_i nor s_j have a tuple with key k.

  For the first case: since $(s_i, i) = \delta(p_0 \sqcup \cdots \sqcup p_{i-1})$, by definition of δ, $n_1 = i$. Similarily, $n_2 = j$. Thus $n_2 > n_1$.

Therefore, by definition of $s_i \sqcup s_j$ and $s_j \sqcup s_i$, (k, v_2, n_2) is a member of both $s_i \sqcup s_j$ and $s_j \sqcup s_i$.

For the second and third case, $(k, v, n) \in s_i \sqcup s_j$ and $(k, v, n) \in s_j \sqcup s_i$ directly from definition.

For the fourth case, neither $s_i \sqcup s_j$ nor $s_j \sqcup s_i$ have a tuple with key k since $s_i \cup s_j = s_j \cup s_i$ doesn't have one.

Hence we've shown that for any $k \in K$, $s_i \sqcup s_j$ and $s_j \sqcup s_i$ have the same tuple with key k (or both don't have one). Thus $s_i \sqcup s_j = s_j \sqcup s_i$. Since $max(m_i, m_j) = max(m_j, m_i)$, we have $p_i \sqcup p_j = p_j \sqcup p_i$.

Suppose $K = V = \mathbb{Z}$. Let $f_1, f_2 \in I$ be defined as:

$$f_1(s) = \begin{cases} 1 \mapsto get(s, 0) + 1 \\ 2 \mapsto get(s, 1) + 2 \end{cases}$$

$$f_2(s) = \begin{cases} 2 \mapsto get(s, 1) + 3 \\ 3 \mapsto get(s, 2) + 4, \end{cases}$$

where get is defined as in example 2.1.3. An example execution of the machine is:

$$(s_0, 0), f_1, ((s_1, 1), o_1), f_2, ((s_2, 2), o_2),$$

where:

$$s_0 = \emptyset$$

$$s_1 = \begin{cases} 1 \mapsto (1, 1) \\ 2 \mapsto (2, 1) \end{cases} \qquad o_1 = \emptyset$$

$$s_2 = \begin{cases} 1 \mapsto (1, 1) \\ 2 \mapsto (4, 2) \\ 3 \mapsto (6, 2) \end{cases} \qquad o_2 = \begin{cases} 1 \mapsto 1 \\ 2 \mapsto 2. \end{cases}$$

Note that $(s_1, 1) = (s_0, 0) \sqcup (p_1, 1)$ and $(s_2, 2) = (s_1, 1) \sqcup (p_2, 2)$, where

$$p_1 = \begin{cases} 1 \mapsto (1, 1) \\ 2 \mapsto (2, 1) \end{cases} \qquad p_2 = \begin{cases} 2 \mapsto (4, 2) \\ 3 \mapsto (6, 2). \end{cases}$$

These are the patches produced by δ: $((p_1, 1), o_1) = \delta((s_0, 0), f_1)$ and $((p_2, 2), o_2) = \delta((s_1, 1), f_2)$. □

From any patch machine $\mathcal{M} = (P, I, O, \delta, p_0, \sqcup)$ we can obtain a state machine $\mathcal{M}' = (S, I, O, \delta', s_0)$, where $S = P$, $s_0 = p_0$, and

$$\delta'(s, i) = (s \sqcup p, o), \text{ where } (p, o) = \delta(s, i).$$

Observe that every execution of $\mathcal{M}$

$$s_0 = p_0, i_1, (s_1, o_1), i_2, (s_2, o_2), \ldots$$

is also an execution of $\mathcal{M}'$.

On the other hand, from a state machine $\mathcal{M} = (S, I, O, \delta, s_0)$ we can obtain a patch machine $\mathcal{M}' = (P, I, O, \delta', p_0, \sqcup)$, where:

$$
\begin{aligned}
&P = S \times \mathbb{N}, \\
&p_0 = (s_0, 0), \\
&\delta'((s, N), i) = ((s', N+1), o), \text{ where } (s', o) = \delta(s, i), \\
&(s_1, N_1) \sqcup (s_2, N_2) = \begin{cases} (s_1, N_1), \text{ if } N_1 > N_2, \\ (s_2, N_2) \text{ otherwise.} \end{cases}
\end{aligned}
$$

This construction was used to obtain example 2.2.1 from example 2.1.2. Observe that from each execution of $\mathcal{M}$

$$s_0, i_1, (s_1, o_1), i_2, (s_2, o_2), \ldots$$

we can obtain an execution of $\mathcal{M}'$:

$$(s_0, 0), i_1, ((s_1, 1), o_1), i_2, ((s_2, 2), o_2), \ldots$$

Observe how the inputs and outputs of the two executions are the same.

Thus, in some sense, state machines and in patch machines are equivalent. Although the execution of $\mathcal{M}'$ has different states, the sequence of inputs and outputs (obtained by erasing the states) is the same; if we view the inputs and outputs as the "external interface" of the machine, and the states as an "implementation detail", then $\mathcal{M}$ and $\mathcal{M}'$ appear to behave identically from the point of view of an external observer.

The general method of obtaining patch machines from state machines above is shown only to argue that the two abstractions have the same power of expression; it shouldn't necessarily be used in practice. The replication algorithm presented in the next chapter works best with patch machines for which the patches produced by δ are "small", since it needs to send these patches through the network.

For example, consider the state machine from example 2.1.3. Suppose that $f \in I$ overwrites the values of two keys. The size of $(s', o) = \delta(s, f)$ is proportional to the size of s, because $o = s$ and

$$|dom(s)| \leq |dom(s')| \leq |dom(s)| + 2.$$

If we directly apply the above construction to obtain a patch machine, the size of each patch produced by it would also be proportional to the size of the used state (even if the command modifies only a constant number of keys). But as we've seen in example 2.2.3, there is a better way to express the same idea: in this example, the sizes of produced patches are proportional to the number of keys modified by the commands.

Chapter 3

Replication

In the introduction we have outlined a method of implementing distributed services using replicated state machines. A replication protocol maintains copies of a deterministic state machine running on multiple servers and ensures that each copy executes the same sequence of commands.

In *Part-Time Parliament* [13], Lamport presented an algorithm for a set of unreliable actors to agree on a sequence of values. The actors were legislators in a parliament, and the values were decrees that modified the law in force on the Greek island of Paxos. The unreliability of the legislators manifested itself in a tendency to leave the legislative chamber in the middle of parliamentary proceedings (they would sometimes come back after years of absence). However, in order to function and choose new decrees, the parliament required only a majority of the legislators to be present. After a decree was agreed upon, it could never change, and there had to be a way for the legislators and citizens of Paxos to learn what the decree was. The decrees were numbered: the parliament would first attempt to choose the first decree, then the second, third, and so on. Due to their unreliability it would often happen that decrees were chosen out of order — the 5th decree could have been chosen before the 3rd decree — but to determine the law in force after the 5th decree was passed, one would have to wait for all previous decrees to be chosen and learn all of them.

The paper also briefly mentioned how the algorithm could be used to solve the state machine replication problem. Substitute servers in a distributed system for legislators. The algorithm tolerates the servers crashing in the middle of processing a request as it tolerated the whimsical legislators — it runs as long as a majority of servers is available. Substitute decrees that modified the law for commands that modify the state machine. The consistent numbering of commands and the properties of agreement (after a command is chosen, it never changes) ensure that the different copies of the state machine apply the commands in the same order.

Today the algorithm is commonly referred to by the name *Paxos*. It is being successfully used to replicate state machines in systems such as Google Chubby [2] or Spanner [5], by agreeing on a sequence of commands, as Lamport suggested in his famous paper. Underneath, Paxos uses a consensus protocol sometimes referred to as the *multi-decree Synod* [22].

In this thesis we're also going to use the multi-decree Synod, but instead of choosing commands of a state machine we will be choosing patches of a patch machine. As

we will see, this leads to a simple replication algorithm that does not require keeping a log of commands.

3.1. Getting rid of the log

A "by-product" of using Paxos for replicating a state machine is the log of commands which has to be carefully maintained; while the consensus algorithm itself (the part of Paxos which agrees on a command) is rather simple (after one spends enough time trying to understand why it works), the log bookkeeping adds a lot of overhead — both mental overhead for the implementor, and resource usage overhead for the system — and takes a central part in many Paxos implementations.

The main reason of the necessity of keeping the command log is to be able to process and respond to client requests even if some servers are currently unavailable. Indeed: when a new command is chosen, it doesn't have to be applied to *every* copy of the state machine right away. It just needs to be applied to a single copy (one which applied all previous commands) in order to obtain an output which can be sent back to the client. The command is then kept so it can be applied to other copies as well some time in the future. All of this is still rather simple, but then reality tells us that we cannot keep a command forever due to lack of unlimited space (and other reasons) and we are forced to incorporate a complex log compaction mechanism into our replication algorithm.

A lot of recent research in distributed algorithms focuses on Conflict-Free Replicated Data Types (CRDTs) [21]. Roughly speaking, the idea is that different replicas may contain partial information about the state of a data structure. These different parts can then be merged together to obtain the whole picture.

CRDTs (with the underlying mathematical theory of semilattices) are the inspiration behind the idea of choosing a sequence of patches for a patch machine instead of a sequence of inputs. As we will see, the patches of a patch machine will not have to be applied on every replica in the same order. Each patch will only have to be applied to a majority of replicas, not all of them, causing replicas to miss some of the patches.

This will allow us to forget a patch as soon as it's applied to a majority; we can then proceed to choosing another patch. The need to maintain a complex distributed log of commands is gone.

But to choose the next patch, the algorithm first needs to obtain the current state s in order to calculate $\delta(s, i)$, where i is the handled input[1]. This can be done by contacting a majority of replicas and merging the obtained pieces of information. The properties of patch machines and the fact that every previous patch was applied to some majority implies that the merged pieces give the "correct" state, i.e. the state that we would obtain by applying all of the previous patches in-order.

This naturally leads to the following question: does the requirement of applying a patch to a majority of replicas before the next patch is chosen and the requirement of contacting a majority to retrieve the current state reduce the fault-tolerance of

[1] Fetching the entire state may not be necessary to handle the given input, only a part of it; we discuss this idea in section 4.1. For now, to make the exposition clearer, we assume that the whole state is required.

the algorithm, compared to the usual way Paxos is applied to solve state machine replication?

The answer to the question is *no*. Paxos needs a majority of participants to be available anyway in order to reach agreement on the nth decree. The requirement that a majority of participants is available in order to apply a chosen patch is thus not stronger. The algorithm will be able to proceed as soon as a majority of servers are non-faulty and can communicate.

3.2. System model

A *process* represents a sequential program running on one of the servers in the system.

The processes communicate through messages using a network. A process might receive a message, which may cause it to make a state transition (change its internal state) and/or send a message. It may also decide to spontaneously perform a state transition and/or send a message.

We assume an asynchronous distributed system model with non-byzantine failures:

- the processes operate at arbitrary speeds. The processing of a message and producing a response may take an arbitrarily long time.

  A process that crashes but some time later restarts and recovers its previous state using persistent storage, from the point of view of the model is no different from a process that was just slow (*very* slow) during a period of time when it was down; there is no way for other processes to distinguish between these two cases.

- A process may encounter a *crash failure*, meaning that it can't make progress (it doesn't perform any more state transitions and doesn't send messages). This is different from the "crash and then recover" scenario described above, which is not a crash from the model's point of view; here we're dealing with an irrecoverable crash. This can represent scenarios in which processes require persistent storage in order to safely recover, but they cannot because the storage is gone for some reason (e.g. the disk got corrupted) or wasn't even used in the first place.

- Messages may take arbitrarily long to deliver. They may be lost, reordered, or duplicated.

- Processes don't lie. They don't attempt to subvert the protocol. They perform according to the specification, i.e. every state transition made by a process corresponds to a state transition of the specification.

  This (together with the next item) is what it means for the model of failures to be non-byzantine. An example byzantine failure would be a process that crashes and then restarts from the initial state (because e.g. it lost its persistent storage) while the specification doesn't allow going back to the initial state. From the model's point of view the process performed an illegal state transition. Now the process can start sending messages which should never be sent and the algorithm can give incorrect results.

There are algorithms which tolerate byzantine failures (to some degree) but the algorithm presented here is not one of them. The original Paxos and Raft also assume non-byzantine failures only.

- Messages are not corrupted (or modified in any way) while in-flight, and new messages only appear in the network according to the specification. This is the second part of the "non-byzantiness" of the model. An example of how this assumption might be false is an external process pretending to be one of the processes in the system and sending messages that the specification doesn't allow.

Particular types of messages may spontaneously appear in the environment that the processes live in that are not sent by any of the system's processes, and be received by them (but the appearance of such messages must be explicitly allowed by the specification). The processes may also send messages "to the environment", not aimed at any other process in the system. Such messages represent communication with "the external world", e.g. with clients that are using the service.

3.3. Outline of the algorithm

We will later specify the algorithm using a formal language. But to gain some initial intuition, we first present an informal outline of how the algorithm runs. This outline is longer than the specification for the following reasons:

- in the outline we use prose, which is less concise than the formal language,
- we give some implementation tips for easier understandability of how the algorithm could run in a real system; some of these details are not enforced by the specification,
- we describe the meaning and purpose of some of the variables appearing in the specification. Just reading the specification does not make it immediately obvious what the variables are supposed to represent; only the theorems we prove later establish semantics and relationships between them. In the outline, we describe what the variables "do", sometimes stating facts that require a proof (the proofs are shown in section 3.5, after presenting the algorithm formally).

Let $\mathcal{M} = (P, I, O, \delta, p_0, \sqcup)$ be a patch machine. While the algorithm executes, the processes will reach agreement on a sequence of patches for the machine $\mathcal{M}$. The position of each patch in the sequence is determined by the *slot* that the patch was chosen in.

A slot represents a container that can hold a single patch. There is a single slot for every natural number. Initially, slots $1, 2, \ldots$ are empty, and slot 0 contains the initial patch p_0. The algorithm will attempt to decide what the patch p_1 in slot 1 is, then decide the patch p_2 in slot 2, and so on. After a slot is filled with a patch, it can never be changed.

The patches p_n will be refered to as *the chosen patches*, with p_n being the *nth chosen patch*. It will be shown later that the sequence $p_1, p_2, \ldots$ forms a descendant sequence of patches.

The state of the replicated patch machine is stored by a set of processes called **replicas**. As the algorithm executes, replicas learn that new patches are decided on and update their state. Every replica *rep* remembers two things:

- $last[rep]$ is the highest slot among the slots of all chosen patches learned by *rep*.
- $store[rep]$ is a patch. It is the result of merging a sequence of patches learned by *rep* from the set $\{p_0, p_1, \ldots, p_{last[rep]}\}$ using the $\sqcup$ function. That is,

  $$store[rep] = p_{i_0} \sqcup p_{i_1} \sqcup \cdots \sqcup p_{i_m},$$

  where $i_k \in \{0, \ldots, last[rep]\}$ for every $0 \leq k \leq m$. The length of the sequence, m, may be any natural number. Furthermore, the patches may appear multiple times and in arbitrary order. By the previous point, $last[rep] = i_k$ for some k, i.e. $p_{last[rep]}$ appears at least once somewhere in the sequence. It is also true that $i_0 = 0$, i.e. the first patch in the sequence is always p_0.

  We will say that *rep* has *applied* the patches $p_{i_0}, \ldots, p_{i_m}$.

Below we will use the term "a quorum of replicas". A quorum of replicas is a subset of the set of replicas which contains more than half of all the replicas. For example, if there are 5 replicas, then a quorum is any set containing 3, 4, or 5 replicas. If there are 4 replicas, then a quorum is any set containing 3 or 4 replicas. A quorum may itself contain multiple quorums; for example, a quorum of 4 replicas (with 5 replicas in total) contains $\binom{4}{3} + 1 = 5$ quorums (including itself). Note that any two quorums have a nonempty intersection.

Initially, for each *rep*, $last[rep] = 0$ and $store[rep] = p_0$. The algorithm proceeds roughly as follows:

1. a client request containing $i \in Input$ arrives.
2. A quorum of replicas is contacted and their stored patches are retrieved.
3. The patches are merged in order to obtain the current state, say s_n.
4. A patch and an output is calculated: $(p, o) = \delta(s_n, i)$.
5. A decision is made that the $(n+1)$th patch will be p, i.e. $p_{n+1} = p$.
6. The output o is then returned to the client which sent the request.
7. The patch p_{n+1} is applied onto at least a quorum of replicas, i.e. each member *rep* of some quorum performs the following:
   - updates $last[rep]$ to $max(last[rep], n+1)$,
   - updates $store[rep]$ to $store[rep] \sqcup p_{n+1}$.

   This will make it possible to later obtain s_{n+1} by contacting a (possibly different) quorum of replicas and merging their patches.
8. The algorithm proceeds to handling another client request.

This is a sequential operational description given for easier understanding, but many of these steps can be done in parallel. For example:

- steps 2 and 3 can be done without waiting for step 1. When the client request arrives, s_n can be already calculated so we can immediatelly proceed to 4.
- Step 2 can be merged with step 7. That is, a single message can be used to inform replicas about p_{n+1} and ask them to return their current patch (after applying p_{n+1}) in order to calculate s_{n+1}.
- Steps 6 and 7 can be done in parallel.

The formal specification which we'll present later enables these (any more) kinds of parallelism.

It is not obvious that step 3 allows obtaining s_n by merging the states stored on some quorum of replicas, based only on the fact that each previously chosen patch was applied by some quorum (and for each patch, that quorum may be different). Showing this will be one part of proving the algorithm's correctness.

The most complex part of the algorithm is step 5 — making a decision on the $(n+1)$th patch. The complexity follows from the fact that there may be multiple processes that handle a client request and try to get a new patch chosen in the $(n+1)$th slot; their candidate patches may be different. They need to achieve *consensus* on the chosen patch. This is done using the *multi-decree Synod* protocol which we describe next.

In order to choose a patch, a *proposal* must be issued. Each proposal has a *decree* — the value being proposed (in our case, a patch), and a slot number, designating the slot in which we want to choose the decree.

There is a set of **acceptor** processes which can *accept* issued proposals. A *quorum of acceptors* is a set of acceptors containing more than a half of them. A proposal is *chosen* when a quorum of acceptors accept it. After a proposal is chosen, it remains chosen (a proposal cannot be "unaccepted"). A decree is chosen in slot n if there is a proposal with slot number n containing this decree that is chosen. The algorithm must ensure that at most one decree is chosen in a given slot; thus, if two proposals with the same slot number are chosen, the algorithm must ensure that they have the same decree.

Proposals are sorted into *ballots*. In a single ballot, there can be at most one proposal issued for each slot. Multiple proposals can be issued for the same slot, but then they have to be issued in different ballots.

The set of ballots is given a-priori (unlike proposals, which are created as the algorithm executes). Ballots are uniquely identified by *ballot numbers*, which we'll shortly call *balnums*. Balnums are totally ordered, and the order is a well-order, i.e. every subset of balnums has a smallest element. In particular, there exists a smallest balnum, which we'll denote 0. The set of balnums is unbounded; for any balnum, we can find a higher one. A good example for a set of balnums is the set of natural numbers. Ballots are ordered by their numbers.

Each proposal has a balnum, additionally to the decree and the slot number. The balnum defines which ballot the proposal belongs to.

Proposals within the same slot are ordered by their balnums.

For a proposal $prop$, we will use $num(prop)$ to denote its balnum, $slot(prop)$ to denote its slot, and $dec(prop)$ to denote its decree.

The algorithm maintains the following property:

(P1) for each proposal $prop$ there exists a quorum of acceptors Q such that either:

- no member of Q has accepted or will ever accept a proposal $prop_0$ with the same slot in a lower ballot (i.e. $slot(prop_0) = slot(prop)$ and $num(prop_0) < num(prop)$),
- or $dec(prop)$ is equal to $dec(prop_0)$, where $slot(prop_0) = slot(prop)$, $num(prop_0) < num(prop)$, and:
  - a member of Q has accepted $prop_0$,
  - for every $prop_1$ with $slot(prop_1) = slot(prop)$ and $num(prop_1) < num(prop)$ that a member of Q has or will ever accept, $num(prop_1) \leq num(prop_0)$.

  In other words: $dec(prop) = dec(prop_0)$, where $prop_0$ is the greatest proposal (w.r.t balnum) before $num(prop)$ among all proposals with slot $slot(prop)$ that a member of Q has or will ever accept.

It turns out that maintaining P1 is enough to ensure that if two proposals with the same slot are chosen, then both have the same decree, assuming that different proposals with the same slot are issued in different ballots (lemma 3.5.13 and theorem 3.5.1). This prevents two different decrees from being decided for the same slot.

The algorithm also maintains the following property:

(P2) for slots $n > 1$, a proposal is issued with slot number n only if a decree has been decided in slot $n - 1$.

P2 implies that decrees are chosen consecutively: first in slot 1, then in slot 2, and so on.

We will now informally describe how P1 and P2 are enforced by the algorithm.

Each acceptor accepts only proposals with increasing slot numbers and balnums. That is, if an acceptor accepts $prop_1$ and later accepts $prop_2$, then $slot(prop_2) \geq slot(prop_1)$ and $num(prop_2) \geq num(prop_1)$. The acceptor always remembers the *greatest* proposal (w.r.t both the slot number and the balnum) among all proposals previously accepted. For acceptor acc, let $maxProp[acc]$ denote the greatest proposal previously accepted by acc. If acc didn't accept any proposals yet, then let $maxProp[acc]$ be a special *null proposal*, denoted $\perp$, with $slot(\perp) = 0$, $num(\perp) = 0$, and no decree. "Real" proposals are only issued in slots ≥ 1, making it easy to distinguish them from $\perp$.

Before any proposal is issued in ballot num, a quorum of acceptors Q must do the following. Each member $acc \in Q$:

- checks that $num > num(maxProp[acc])$,
- remembers not to accept any more proposals in ballots lower than num,
- sends a message m with the following fields:
  - $m.acc = acc$,

– $m.num = num$,

– $m.mprop = maxProp[acc]$.

A message m like above with $m.num = num$ is called a *num-promise*. It says that the greatest proposal that $m.acc$ has accepted or will ever accept in ballots lower than $m.num$ is $m.mprop$. The "has accepted" follows from the fact that $m.mprop$ was equal to $maxProp[acc]$ when m was sent, and the "will ever accept" follows from the fact that $m.acc$ remembered not to accept any more proposals in ballots lower than $m.num$.

After obtaining num-promises from each member of Q, the greatest slot among the slots of returned proposals is calculated:

$$n = max(\{slot(m.mprop) : m \in \text{obtained promises}\}).$$

By definition of the promises we know the following:

- for all slots greater than n, no member of Q has accepted or will ever accept a proposal in any of these slots in ballot lower than num.

  This means that a proposal with *any decree* can be issued in any of these slots in ballot num, and property P1 will be preserved (the new proposal will satisfy the first part of P1).

- For all slots lower than n, a decree has already been decided in these slots (assuming that P2 was preserved by all proposals issued previously). Indeed: the definition of n implies that there has been at least one proposal issued for slot n. An exception is $n = 0$; in that case the statement is vacuously true (there are no slots lower than n).

The only possible "unknown" is slot n: a decree may or may not have been decided in slot n; in order to preserve P2, before issuing a proposal in slot $n + 1$, we have to ensure that a decision is made for slot n. The exception is $n = 0$, which means that no member of Q has accepted any proposal in ballot lower than num *in any slot* (since $m.mprop = \perp$ for every obtained promise m), and we can immediately proceed to issuing a proposal for slot 1.

Suppose then that $n \geq 1$. This means that some member of Q has accepted a proposal in ballot $< num$, and the greatest slot among the slots of all accepted proposals by Q is n.

Among all proposals accepted by members of Q in ballots lower than num in slot n, the one in the largest ballot is taken. That is, let $prop_0$ be such that:

- $prop_0 = m.mprop$ where m is one of the obtained promises,
- $slot(prop_0) = n$,
- $\forall m \in$ obtained promises : $slot(m.mprop) = n \Rightarrow num(m.mprop) \leq num(prop_0)$.

Indeed $prop_0$ is the desired proposal: suppose that $acc \in Q$ accepted a proposal $prop_1$ with $slot(prop_1) = n$ and $num(prop_1) < num$. We have a promise m with $m.acc = acc$ and $m.num = num$, which means that $num(prop_1) \leq num(m.mprop)$ by definition of $m.mprop$. But $num(m.mprop) \leq num(prop_0)$.

We now know that it is safe to issue a proposal $prop$ in ballot num with slot n, if $dec(prop) = dec(prop_0)$. This proposal will satisfy the second part of P1.

If the proposal is issued and gets accepted by some quorum of acceptors, it is then safe to issue a proposal in slot $n + 1$, ballot num, and a new decree, preserving P1 and P2. After this proposal gets accepted, a proposal can be issued in slot $n + 2$ in ballot num with a new decree, and so on.

There is a small optimization that sometimes allows us to avoid issuing a proposal in slot n and instead proceed directly to slot $n + 1$, even for $n \geq 1$. Namely, it might turn out that $prop_0 = m.mprop$ for all promises m obtained from a quorum of acceptors. This means that $prop_0$ was accepted by a quorum, so it is chosen in slot n, hence $dec(prop_0)$ is already decided for slot n. In that case it is safe to issue a proposal in slot $n + 1$ without breaking P2.

We can now fill in the missing piece: who is issuing the proposals? Who are the acceptors sending their promises to? Who is handling client requests? Who is reading the state from replicas and using it to calculate new patches?

Additionally to replicas and acceptors, there is a set of **proposers**. They are the "drivers" of the algorithm: they handle client requests and propose new patches. The proposers may compete with each other, trying to handle different requests in parallel; they use the multi-decree Synod protocol described above to reach consensus.

In order to issue proposals, a proposer must first pick a ballot. It must then obtain promises from a quorum of acceptors for this ballot. If it succeeds, it is ready to issue proposals in the picked ballot.

However, we must ensure that no two proposers pick the same ballot. Otherwise we might risk that two proposals are issued in the same ballot for the same slot but with different decrees, which may lead to two proposers thinking that different decisions were made in the same slot, giving inconsistent results in the end.

This is usually done by having a fixed mapping from balnums to proposers. Each balnum has a single proposer, its *owner*. Proposers pick only ballots identified by balnums that they own. Therefore two proposers can never pick the same ballot.

A standard way of getting such a mapping is to define balnums to be pairs (n, id), where $n \in \mathbb{N}_+$ and id uniquely identifies a proposer; the proposer ids must be ordered. Then the owner of (n, id) is simply the proposer identified by id. Furthermore, with this definition, for each balnum it is easy to find a greater balnum owned by a given proposer.

Assuming we found a way to ensure that each ballot is picked by at most one proposer, the proposer must then take care not to issue two different proposals for the same slot (in the same ballot), which is easy to achieve since it is the only process that may issue proposals in this ballot.

A proposer's operation looks roughly as follows. Let P denote the proposer.

1. P picks a ballot num that cannot be picked by any other proposer (in particular, P ensures that the ballot was not earlier picked by P itself).

Preferably num should be greater than any ballot picked by other proposers in the past; otherwise, if there exists a $num' > num$ that was used to choose proposals, P won't be able to obtain a quorum of accepts using num. The proposer can base its pick on information obtained from acceptors, for example — it can first contact some of the acceptors to learn the greatest ballot that they have previously observed. The details are left to the implementation.

2. P tries to obtain promises from a quorum of acceptors for ballot num. If it succeeds, it proceeds to the next step. If it fails — for example, because all contacted acceptors already accepted a proposal in a higher ballot — it can restart from step 1 but with a higher ballot.

3. The proposer calculates n, the greatest slot among the slots of proposals returned in the promises. If $n = 0$ (meaning that every acceptor in the quorum returned $m.mprop = \bot$), it skips to step 8. Otherwise it proceeds to the next step.

4. P calculates $prop_0$, the proposal with the greatest balnum among proposals returned in the promises with slot n.

   It then checks if $prop_0$ was returned by each member of the quorum (i.e. $m.mprop = prop_0$ for each obtained promise m), which means that $prop_0$ is chosen. If so, it may skip to step 7. Otherwise it proceeds to the next step.

   Note that even if $prop_0$ is chosen, skipping to step 7 is an optional optimization. The proposer can always safely proceed to step 5.

5. P issues a proposal $prop$ with $num(prop) = num$, $slot(prop) = n$, and $dec(prop) = dec(prop_0)$, and sends it to the acceptors.

6. It waits until some quorum of acceptors answers, confirming that they have accepted the proposal. This may of course fail, e.g. because every possible quorum has an acceptor that already made a promise with a higher balnum. In case of failure the proposer can restart from step 1 with a higher ballot. Otherwise it proceeds to the next step.

7. At this point P knows that $prop_0$ is chosen, meaning that a patch is decided in slot n; it is given by $dec(prop_0)$. The proposer sends the patch $p_n = dec(prop_0)$ and the slot number n to replicas for application.

   There is an alternative way for replicas to learn about a chosen patch (besides receiving a message from a proposer): when an acceptor accepts a proposal, instead of sending a message to the proposer who requested the accept, it broadcasts a message to all replicas, telling them that it accepted the proposal. When a replica receives accept messages for the same proposal from a quorum of acceptors, it knows that the proposal is chosen, so it can apply it. This method results in fewer message delays but it requires sending $O(A \times R)$ messages where A is the number of acceptors and R is the number of replicas. The previous method, where accepts are first sent to a proposer and the proposer then informs replicas, requires sending only $O(A + R)$ messages.

   With this alternative method we may also skip step 6. After issuing a proposal, P would not wait for acceptors to answer; instead, it would wait for messages that come directly from replicas, confirming that they have applied a new

patch, which implies that the patch was chosen. The path of communication would look as follows: $P \rightarrow$ acceptors $\rightarrow$ replicas $\rightarrow P$. In the standard approach the path is $P \rightarrow$ acceptors $\rightarrow P \rightarrow$ replicas $\rightarrow P$.

How the proposer and the replicas learn about chosen patches is an implementation choice; the specification presented later does not enforce either method.

8. The proposer is now ready to handle client requests. In order to handle a request, P must first retrieve the current state. For that it contacts the replicas; each replica *rep* sends a message with $store[rep]$ and $last[rep]$ to the proposer. The proposer needs answers from some quorum in order to proceed. For each received message, it looks at $last[rep]$, expecting it to be equal to n — this is used by the algorithm to confirm that a quorum of replicas has applied the previous patch p_n. If the comparison succeds, P merges the obtained patches into s_n and proceeds to the next step. Otherwise:

   - if some replica returned $last[rep] > n$, it means that another proposer already managed to choose a patch in slot $n + 1$, so P is no longer able to choose new patches using ballot *num*. It can restart from step 1 with a higher ballot.

   - if all responses satisfy $last[rep] \leq n$ and there is no quorum satisfying $last[rep] = n$, the proposer can simply resend the previous patch (p_n) back to the replicas and retry the read.

   It may be convenient to merge the "read" step with the "apply chosen patch" step, as described before when we first introduced replicas. The details are left to the implementation.

9. Given $i \in Input$ (from a client request), P calculates $(p, o) = \delta(s_n, i)$ and issues a proposal *prop* with $slot(prop) = n + 1$, $num(prop) = num$, and $dec(prop) = p$.

   As in step 6, the proposal may fail to become chosen due to another proposer, in which case P can restart with a higher balnum.

10. After p is chosen as p_{n+1}, the proposer returns o as the response to the client and sends p with slot number $n + 1$ for replicas to apply. It can then loop back to step 8 to handle another client request, with $n + 1$ substituted for n.

Steps 1 to 7 form the *recovery phase*, and steps 8 to 10 are *normal operation*. After recovery, as long as no other proposers block P's ballot (by having a quorum of acceptors perform a num'-promise for some $num' > num$), P can run steps 8 to 10 in a loop, quickly handling new client requests. It goes back into recovery when it finds that its ballot is no longer useful.

It would be desirable if there was only a single proposer running at a time. Indeed, we can easily find a scenario where contending proposers block each other from getting any proposals chosen, for example:

1. P_1 obtains promises for balnum 1 from a quorum of acceptors in step 2. A moment later it issues a proposal in step 5.

2. In the meantime, P_2 obtains promises for balnum 2 from a quorum of acceptors in step 2. This prevents the proposal from P_1 from being accepted by a quorum. P_2 sends its own proposal.

3. P_1 notices that its ballot got blocked, so it restarts with balnum 3. It obtains the promises for 3, blocking P_2's proposal.

4. P_2 restarts with balnum 4. And so on.

In effect, the system appears unavailable to the clients.

Unfortunately, having a single proposer is also not an option if we aim for fault tolerance. The failure of this single proposer would effectively block the entire system. The entire purpose of having multiple ballots is to allow multiple proposers to send proposals while maintaining consensus.

A common method of dealing with this problem in many consensus-based systems is to use an *unreliable failure detector* [4], specifically the Ω failure detector [3]. The purpose of Ω is to provide a local "oracle" to each process that the process can query; the oracle points to one of the other processes that the oracle thinks to be currently non-faulty. The theoretical property of Ω is that eventually the oracles of all non-faulty processes point to a single non-faulty process (assuming there is one), often referred to as the *leader* (Ω is said to provide *weak leader election* functionality). While unreliable failure detectors are theoretical constructs and cannot be implemented in asynchronous environments [4], one can build approximations of them with good practical properties using mechanisms such as timeouts and periodic heartbeats exchanged between processes.

We can apply Ω in order to solve the problem of contending proposers. Before a proposer P starts the recovery phase, it queries the failure detector. If the failure detector returns P, then P proceeds to picking a balnum and obtaining promises. Otherwise, P remains dormant. P will continue to periodically query the failure detector in order to decide if it should start recovery. Until then, all client requests can either be rejected by P or redirected to whomever P thinks to be the current leader (based on the output of Ω). The details are left to the implementation.

Note that the failure detector is only used to improve the availability of the system; the algorithm preserves safety (meaning that it won't return incorrect results) in completely asynchronous conditions. The oracles may return random results, but it will only affect liveness, not correctness.

Before we present the formal specification, let us go back to acceptors for a moment. We now know that acceptors can perform two actions: *accept* a proposal, and *promise* to not accept proposals in ballots lower than some given balnum. To perform these actions, in addition to storing a $maxProp[acc]$ by each acceptor acc, which is the greatest proposal accepted by acc with respect to the slot and balnum, acc also stores $maxNum[acc]$, which is defined to be the maximum of two values:

- $num(maxProp[acc])$,
- $max(\{num : acc \text{ gave a } num\text{-promise}\})$.

For simplicity, we specify that acc can only give a num promise if $num > maxNum[acc]$.

Summarizing, each acceptor acc stores $maxNum[acc]$ and $maxProp[acc]$, where:

- initially $maxNum[acc] = 0$ and $maxProp[acc] = \perp$, where $slot(\perp) = 0$ and $num(\perp) = 0$.
- $maxNum[acc] \geq num(maxProp[acc])$.
- acc gives a num-promise only if $num > maxNum[acc]$; it then updates $maxNum[acc]$ to num.
- acc accepts a proposal $prop$ only if $num(prop) \geq maxNum[acc]$ and $slot(prop) \geq slot(maxProp[acc])$. It then updates $maxNum[acc]$ to $num(prop)$ and $maxProp[acc]$ to $prop$.

3.4. The specification

In order to precisely define the algorithm and formally reason about it we will use TLA+ [14], a formal specification language that has been widely applied to specify systems and algorithms both in science [17] and industry [16]. If the reader is not familiar with TLA+, a good and short introductory course in TLA+ which the author of this thesis highly recommends is the *TLA+ Video Course* [15].

It is not immediately obvious how a specification can be translated to an implementation, and how the specification represents concepts such as sending a message, for example. The specification also contains elements such as the *history* variable which are used only for reasoning about the algorithm's correctness and do not necessarily have an implementation counterpart. We first present the full spec and then describe how it links to the previously discussed informal outline. In section 4.3 we discuss an actual implementation of the algorithm written in the Rust programming language.

The specification is given below.

MODULE *LPaxos*

EXTENDS *Integers*, *FiniteSets*, *Sequences*

CONSTANT

Patch, *Input*, *Output*, *Delta*(_, _), *p*0, *Merge*(_, _),

Acceptor, *Replica*,

Balnum,

ReqId

$$AQuorum \triangleq \{q \in \text{SUBSET } Acceptor : Cardinality(q) > Cardinality(Acceptor) \div 2\}$$

$$RQuorum \triangleq \{q \in \text{SUBSET } Replica : Cardinality(q) > Cardinality(Replica) \div 2\}$$

$$Max(a, b) \triangleq \text{IF } a > b \text{ THEN } a \text{ ELSE } b$$

RECURSIVE $MergeSet(\_)$

$MergeSet(ps) \triangleq$

LET $p \triangleq$ CHOOSE $p \in ps$: TRUE

IN IF $ps = \{p\}$ THEN p

ELSE $Merge(p,\ MergeSet(ps \setminus \{p\}))$

RECURSIVE $MergeSeq(\_)$

$MergeSeq(ps) \triangleq$ IF $Len(ps) = 1$ THEN $Head(ps)$

ELSE $Merge(Head(ps),\ MergeSeq(Tail(ps)))$

ASSUME

$\land\ Balnum \subseteq Nat$

$\land\ p0 \in Patch$

$\land\ \forall\, p \in Patch,\ i \in Input : Delta(p,\ i) \in Patch \times Output$

$\land\ \forall\, p1,\ p2 \in Patch : Merge(p1,\ p2) \in Patch$

$\land\ \forall\, p1,\ p2,\ p3 \in Patch :$

$Merge(p1,\ Merge(p2,\ p3)) = Merge(Merge(p1,\ p2),\ p3)$

$\land\ \forall\, p \in Patch : Merge(p,\ p) = p$

$\land\ \forall\, p \in Seq(Patch) :$

LET $pp \triangleq \langle p0 \rangle \circ p$

IN $(\forall\, n \in$ DOMAIN $pp \setminus \{1\} :$

$\exists\, i \in Input,\ o \in Output :$

$\langle pp[n],\ o \rangle = Delta(MergeSeq(SubSeq(pp,\ 1,\ n-1)),\ i))$

$\Rightarrow \forall\, n1,\ n2 \in$ DOMAIN $pp :$

$Merge(pp[n1],\ pp[n2]) = Merge(pp[n2],\ pp[n1])$

VARIABLE

$history,$

$requests,\ reads,\ promises,\ proposals,\ accepts,$

$maxNum,\ maxProp,$

$store,\ last$

$Request(i) \triangleq$

$$
\begin{array}{l}
\quad \land \quad \exists\, id \in ReqId : \\
\qquad \land \neg id \in \{req.id : req \in requests\} \\
\qquad \land requests' = requests \cup \{[id \mapsto id,\ input \mapsto i]\} \\
\qquad \land history' = Append(history,\ [type \mapsto \text{“request”},\ id \mapsto id,\ input \mapsto i]) \\
\quad \land \quad \text{UNCHANGED } \langle maxNum,\ maxProp,\ store,\ last, \\
\qquad\qquad\qquad\qquad reads,\ promises,\ proposals,\ accepts\rangle
\end{array}
$$

$$
\begin{array}{l}
Promise(acc,\ num) \triangleq \\
\quad \land \quad maxNum[acc] < num \\
\quad \land \quad maxNum' = [maxNum \text{ EXCEPT } ![acc] = num] \\
\quad \land \quad promises' = promises \cup \\
\qquad \{[acc \mapsto acc,\ num \mapsto num,\ mprop \mapsto maxProp[acc]]\} \\
\quad \land \quad \text{UNCHANGED } \langle history,\ maxProp,\ store,\ last, \\
\qquad\qquad\qquad\qquad requests,\ reads,\ proposals,\ accepts\rangle
\end{array}
$$

$$
\begin{array}{l}
Read(rep) \triangleq \\
\qquad \land reads' = reads \cup \\
\qquad\quad \{[rep \mapsto rep,\ p \mapsto store[rep],\ last \mapsto last[rep]]\} \\
\qquad \land \text{UNCHANGED } \langle history,\ maxNum,\ maxProp,\ store,\ last, \\
\qquad\qquad\qquad\qquad requests,\ promises,\ proposals,\ accepts\rangle
\end{array}
$$

$$
\begin{array}{l}
Repropose(num) \triangleq \\
\quad \land \neg \exists\, p \in proposals : p.num = num \\
\quad \land \exists\, Q \in AQuorum : \\
\qquad \exists\, ms \in \text{SUBSET } \{m \in promises : m.acc \in Q \land m.num = num\} : \\
\qquad\quad \land Q \subseteq \{m.acc : m \in ms\} \\
\qquad\quad \land \exists\, maxp \in ms : \\
\qquad\qquad \land \forall\, m \in ms : \\
\qquad\qquad\quad \lor maxp.mprop.slot > m.mprop.slot \\
\qquad\qquad\quad \lor \land maxp.mprop.slot = m.mprop.slot \\
\qquad\qquad\qquad\ \land maxp.mprop.num \geq m.mprop.num \\
\qquad\qquad \land maxp.mprop.slot \neq 0 \\
\qquad\qquad \land proposals' = proposals \cup \\
\qquad\qquad\quad \{[num \mapsto num, \\
\qquad\qquad\qquad dec \ \mapsto maxp.mprop.dec,
\end{array}
$$

$$slot \mapsto maxp.mprop.slot]\}$$

$\land$ UNCHANGED $\langle history,\ maxNum,\ maxProp,\ store,\ last,$

$$requests,\ reads,\ promises,\ accepts\rangle$$

$ProposeNew(num,\ slot)\ \triangleq$

$\land \neg\exists\, p \in proposals : p.num = num \land p.slot = slot$

$\land \exists\, Q \in AQuorum :$

$\exists\, ms \in$ SUBSET $\{m \in promises : m.acc \in Q \land m.num = num\} :$

$\land\ Q \subseteq \{m.acc : m \in ms\}$

$\land \forall\, m \in ms : m.mprop.slot < slot$

$\land \exists\, Q \in RQuorum :$

$\exists\, rs \in$ SUBSET $\{r \in reads : r.rep \in Q \land r.last = slot - 1\} :$

$\land\ Q \subseteq \{r.rep : r \in rs\}$

$\land \exists\, req \in requests :$

LET $res\ \triangleq\ Delta(MergeSet(\{r.p : r \in rs\}),\ req.input)$

IN $proposals' = proposals \cup$

$\{[num \mapsto num,$

$dec\ \mapsto [id \mapsto req.id,\ p \mapsto res[1],\ o \mapsto res[2]],$

$slot \mapsto slot]\}$

$\land$ UNCHANGED $\langle history,\ maxNum,\ maxProp,\ store,\ last,$

$$requests,\ reads,\ promises,\ accepts\rangle$$

$Accept(acc)\ \triangleq$

$\land \exists\, p \in proposals :$

$\land\ maxNum[acc] \leq p.num$

$\land\ maxProp[acc].slot \leq p.slot$

$\land\ maxNum' = [maxNum$ EXCEPT $![acc] = p.num]$

$\land\ maxProp' = [maxProp$ EXCEPT $![acc] = p]$

$\land\ accepts'\ = accepts \cup \{[acc \mapsto acc,\ p \mapsto p]\}$

$\land$ UNCHANGED $\langle history,\ store,\ last,$

$$requests,\ reads,\ promises,\ proposals\rangle$$

$Apply(rep)\ \triangleq$

$\land \exists\, Q \in AQuorum : \exists\, p \in proposals :$

$\land \forall\, acc\ \in Q : [acc \mapsto acc,\ p \mapsto p] \in accepts$

$\land\ store' = [store \text{ EXCEPT } ![rep] = Merge(store[rep],\ p.dec.p)]$

$\land\ last' = [last \text{ EXCEPT } ![rep] = Max(last[rep],\ p.slot)]$

$\land$ UNCHANGED $\langle history,\ maxNum,\ maxProp,$

$requests,\ reads,\ promises,\ proposals,\ accepts\rangle$

$Respond \triangleq$

$\land \exists\, p \in proposals : \exists\, Q \in AQuorum :$

$\land \forall\, acc \in Q : [acc \mapsto acc,\ p \mapsto p] \in accepts$

$\land \forall\, n \in \text{DOMAIN } history :$

$history[n].type = \text{“response”} \Rightarrow history[n].id \neq p.dec.id$

$\land\ history' = Append(history,$

$[type \mapsto \text{“response”},\ id \mapsto p.dec.id,\ output \mapsto p.dec.o])$

$\land$ UNCHANGED $\langle maxNum,\ maxProp,\ store,\ last,$

$requests,\ reads,\ promises,\ proposals,\ accepts\rangle$

$Init \triangleq$

$\land\ history = \langle\rangle$

$\land\ maxNum = [acc \in Acceptor \mapsto 0]$

$\land\ maxProp = [acc \in Acceptor \mapsto [num \mapsto 0,\ slot \mapsto 0]]$

$\land\ store = [rep \in Replica \mapsto p0]$

$\land\ last = [rep \in Replica \mapsto 0]$

$\land\ requests = \{\}$

$\land\ reads = \{\}$

$\land\ promises = \{\}$

$\land\ proposals = \{\}$

$\land\ accepts = \{\}$

$Next \triangleq$

$\lor \exists\, i \in Input : Request(i)$

$\lor \exists\, acc \in Acceptor,\ num \in Balnum \setminus \{0\} : Promise(acc,\ num)$

$\lor \exists\, rep \in Replica : Read(rep)$

$\lor \exists\, num \in Balnum \setminus \{0\},\ slot \in Nat \setminus \{0\} : ProposeNew(num,\ slot)$

$\lor \exists\, num \in Balnum \setminus \{0\} : Repropose(num)$

$\lor \exists\, acc \quad \in Acceptor : Accept(acc)$

$\lor \exists\, rep \quad \in Replica : Apply(rep)$

$\lor Respond$

The specification's parameters are given by the *CONSTANT* statement. The patch machine being replicated is defined by the first line in the parameter list:

$$\mathcal{M} = (Patch, Input, Output, Delta, p0, Merge).$$

We model the transition and merge functions as TLA+ operators that take two arguments.

The acceptor processes are given by the *Acceptor* set, and the replica processes are given by the *Replica* set. In order to reduce the size of the specification, we do not explicitly define the set of proposers. Instead we operate directly on the set of ballot numbers, given by the *Balnum* set, and use the previously stated assumption that each balnum has a unique proposer owning the balnum to reason about proposers.

The *ReqId* set is a set of *request identifiers*; we use it to uniquely identify all client requests. These identifiers will be used in the proof of correctness and might, but don't necessarily have to have a direct counterpart in an implementation.

After parameters we introduce a couple of auxiliary definitions.

AQuorum is the set of all acceptor quorums: sets of acceptors containing more than a half of them. Similarily, *RQuorum* is the set of all replica quorums.

The *MergeSeq* operator merges a sequence of patches. If the sequence is empty, or if we pass anything else than a sequence of patches to *MergeSeq*, the result is unspecified.

We use standard TLA+ finite sequences defined in the *Sequences* module; there, a finite sequence with elements from some set S is a function from a prefix of the set of positive natural numbers to S:

$$seq : \{1, \ldots, n\} \to S, \text{ where } n \in \mathbb{N}_+.$$

Then $seq[1]$ is the first element of the sequence, $seq[2]$ the second, and so on.[2] For a sequence seq, $Len(seq)$ is the length of seq, $Head(seq)$ denotes $seq[1]$, and $Tail(seq)$ denotes the sequence $seq[2], seq[3], \ldots, seq[Len(seq)]$.

MergeSet merges a set of patches using the *Merge* operator. We use the TLA+ *CHOOSE* operator to select a patch from the set in order to define *MergeSet* recursively. Which patch is chosen is not specified, thus the resulting order of merging is not specified. If the passed value is anything else than a non-empty set of patches, the result is unspecified.

The specification then states the assumptions made about the parameters. The first conjuct of the *ASSUME* statement says that *Balnum* is a subset of $\mathbb{N}$ (written as *Nat* in TLA+). This assumption is made for simplicitly, so we can use operators such as

[2] In TLA+, $f[x]$ denotes the evaluation of a function f on the element $x \in DOMAIN\ f$.

$<$ imported from the *Integers* module, and use the number 0 to denote the smallest balnum. Later in the proof we only use the fact that *Balnum* is well-ordered; the comparison symbols then refer to that order and 0 refers to the smallest balnum.

The other conjucts of *ASSUME* are direct translations of statements from the definition of a patch machine. Perhaps the last assumption requires some commentary. This is the statement saying that *Merge* commutes for a set of patches that form a descendant sequence. The $\circ$ operator concatenates two sequences. $Seq(S)$ for a set S is the set of all finite sequences with elements from S.

The formula says that for all sequences of patches that start with $p0$ — denoted as pp in the formula — if each patch in the sequence ($pp[n]$) is calculated from the previous patches ($MergeSeq(SubSeq(pp, 1, n-1))$) using the *Delta* function, then the order of merging any two patches in that sequence does not matter. The *MergeSeq* operator is defined later in the specification. The *SubSeq* operator comes from the *Sequences* module; for a given sequence seq, $SubSeq(seq, 1, n)$ is the sequence $seq[1], \ldots, seq[n]$.

Next come the variables of the specification. The *history* variable represents the sequence of events observed by the clients of the system. Each event is either a *request* or a *response*. A *request* event contains a request identifier and an input; a *response* event contains a request identifier and an output. We represent events as TLA+ records with a *type* field, whose value is either "*request*" or "*response*", an *id* field whose value belongs to the *ReqId* set, and either an *input* field with a value from *Input* (for requests) or an *output* field with a value from *Output*. For example, if $\mathcal{M}$ is the machine from example 2.2.1, one history could be:

$\langle [type \mapsto$ "request", $id \mapsto 0,\ input \mapsto cas(0,\ 1)]$,

$[type \mapsto$ "request", $id \mapsto 1,\ input \mapsto cas(1,\ 2)]$,

$[type \mapsto$ "response", $id \mapsto 1,\ output \mapsto ok]$,

$[type \mapsto$ "request", $id \mapsto 2,\ input \mapsto cas(2,\ 0)]$,

$[type \mapsto$ "response", $id \mapsto 2,\ output \mapsto ok] \rangle$.

Note that requests can happen concurrently, i.e. a new request can be issued by a client (represented as a *request* event in the history) before the previous request finishes. The goal of the proof presented later is to prove that all histories observed by the clients appear as if a single patch machine was executing them. In particular, for *idempotent* patch machines (which we'll define later), we will prove that all observed histories are *linearizable* [7].

The *maxNum*, *maxProp*, *store* and *last* variables were already described in the outline. We use TLA+ functions to represent the states of acceptors (*maxNum*, *maxProp*) and replicas (*store*, *last*). For example, *maxNum* is a function from the *Acceptor* set to the *Balnum* set, meaning that $maxNum[acc]$ is a balnum for each acceptor acc; in TLA+ notation,

$$maxNum \in [Acceptor \to Balnum].$$

That last formula is actually not immediate from the specification as TLA+ does not have a type system. We will have to prove it, which we do in section 3.5.1.

The rest of the variables are used to represent messages in the network: *requests* is the set of client requests, *reads* is the set of messages that replicas send to proposers

(containing their state), *promises* is the set of *num*-promises made by acceptors, *proposals* are the proposals issued by proposers, and *accepts* are sent by acceptors when they accept a proposal.

TLA+ specifications of algorithms often use a single *messages* variable to represent the set of all messages; different message types are then distinguished using a *type* field or similar, like we do for events in the *history* variable. The author of this thesis decided that having separate sets for different message types makes the specification a bit easier to work with.

The reader may wonder why there are no other message types, for example *promise requests* that proposers send to acceptors asking them to make a *num*-promise, or *read requests* that proposers send to replicas asking them to send their current state. The reasons are as follows:

- it is always safe for a proposer to send a promise request. That is, sending a promise request has no preconditions; a proposer can send a new promise request at any point during the algorithm's execution and it won't break any properties. The only effect of sending a promise request would be adding a new message to the set of messages (it wouldn't modify any other variable of the specification), and the only purpose of a promise request would be to enable the action of sending a promise response.

  Instead, we have decided to completely omit the action of sending a promise request; acceptors make *num*-promises spontaneously. In implementations of this specification that have explicit promise requests, the action of sending a promise request corresponds to a stuttering step of the specification, i.e. a step in which values of no variables change.

  This also makes the specification more general: it allows implementations of the algorithm in which an acceptor can decide to make a promise without a request from a proposer, e.g. based on the output of a failure detector.

- Similarily, there is no reason for the specification to have a "read request" action. Instead, replicas send their state spontaneously, and the action of performing a read request by a proposer in an implementation corresponds to a stuttering step.

After variables come the actions of the algorithm.

The $Request(i)$ action, for $i \in Input$, represents a client sending a request to the system. The request is given a new identifier and added to the *requests* set; this represents the request appearing in the network, ready for proposers to receive it. The request event is also appended to *history*.

The $Promise(acc, num)$ action, for $acc \in Acceptor$ and $num \in Balnum$, denotes the acceptor acc making a *num*-promise; we already described what it means in the outline. The acceptor sends a message m that identifies the acceptor ($m.acc = acc$), contains the balnum ($m.num = num$), and the greatest accepted proposal ($m.mprop = maxProp[acc]$). The acceptor remembers not to accept any more proposals in ballots lower than num by updating $maxNum[acc]$.

$Read(rep)$ for $rep \in Replica$ denotes a replica sending its current state. In a real implementation this action would be probably performed after receiving a read request

from a proposer. However, it is always safe for a replica to just send its current state out, even if it may be redundant; in this high level specification we don't care about the reason.

Note that this action says that the message sent contains the whole state $store[rep]$; this allows the specification to remain simple, but for large states it is impractical. In section 4.1 we discuss how a real system can implement this action efficiently.

The $Repropose(num)$ action corresponds to step 5 in the proposer's operation outline. The preconditions of this action map to steps 1 to 4 in the outline:

- the first precondition, $\neg\exists p \in proposals : p.num = num$, says that num was never used by any proposer to issue proposals. It maps to step 1 of the outline.
- The action then specifies that each member of some quorum of acceptors returned a promise:

  $$\begin{array}{l}\exists Q \in AQuorum : \\ \quad \exists ms \in \text{SUBSET } \{m \in promises : m.acc \in Q \land m.num = num\} : \\ \quad\quad \land\ Q \subseteq \{m.acc : m \in ms\}\end{array}$$

  The quorum is called Q. The $\exists ms \in \ldots$ line defines the set of promise messages; each promise comes from a member of Q, and it is a num-promise. The $Q \subseteq \{m.acc : m \in ms\}$ line says ms has messages from all members of Q.
- Next follows the definition of $maxp$, the proposal accepted by members of Q with the greatest slot, and with the greatest balnum among proposals accepted in this slot:

  $$\begin{array}{l}\exists maxp \in ms : \\ \quad \land \forall m \in ms : \\ \quad\quad \lor\ maxp.mprop.slot > m.mprop.slot \\ \quad\quad \lor \land\ maxp.mprop.slot = m.mprop.slot \\ \quad\quad\quad \land\ maxp.mprop.num \geq m.mprop.num\end{array}$$

  This maps to step 4 in the outline; $maxp$ maps to $prop_0$.
- The precondition $maxp.mprop.slot \neq 0$ maps to the check in step 3 of the outline. Here $n = maxp.mprop.slot$; $n = 0$ means that none of the members of Q has accepted a proposal yet, in which case the proposer does not perform recovery, but proceeds directly to issuing new proposals, specified by action $ProposeNew$.
- Finally, if all the preconditions hold, a state transition can be performed; the effect is issuing a new proposal, represented by adding a new element to the $proposals$ set:

  $$\begin{array}{l}proposals' = proposals \cup \\ \quad \{[num \mapsto num, \\ \quad\ dec \mapsto maxp.mprop.dec, \\ \quad\ slot \mapsto maxp.mprop.slot]\}.\end{array}$$

  As described in the outline, the decree of the new proposal is equal to $dec(maxp) = dec(prop_0)$.

The $ProposeNew(num, slot)$ action corresponds to step 9 of the outline. It denotes the owner of ballot num sending a proposal in that ballot with slot $slot$ and a new decree; $slot$ maps to $n + 1$ in the outline. The decree contains a patch which is calculated from the state obtained by reading from a quorum of replicas and the input from a client request using the transition function $Delta$.

A slight difference from what we described in the outline is that the decree, additionally to the patch, contains the identifier of the request from which the input used to calculate the patch was taken, and the calculated output. This request identifier will be used in the correctness proof.

Implementations of this algorithm don't necessarily have to send anything besides the patch as part of the proposed decree. For the output, for example, the proposer can simply store the output in memory until it learns that the proposal was chosen, then send the output back to the client. Encoding this in the specification would require introducing additional proposer state which the author preferred to avoid. As for the identifiers, the implementation may completely omit them; here we use them only for reasoning about the algorithm (the identifiers are used to match responses to requests in the *history* variable). However, we will find an actual "real" use for request identifiers later, in section 4.2.

The action is constructed as follows:

- the first precondition says that no proposal was yet sent in ballot num and slot $slot$. Recall that at most one proposer sends proposals in each ballot, making it easy for the proposer to enforce this precondition.
- The next precondition checks that it is safe for the proposal to have any decree by assuring that no member of a quorum of acceptors has accepted a proposal in slot $slot$ in lower ballots:

$$\begin{aligned}
&\exists Q \in AQuorum : \\
&\quad \exists ms \in \text{SUBSET } \{m \in promises : m.acc \in Q \land m.num = num\} : \\
&\qquad \land\ Q \subseteq \{m.acc : m \in ms\} \\
&\qquad \land\ \forall m \in ms : m.mprop.slot < slot
\end{aligned}$$

  From the proposer's operational perspective, these are the promises obtained during the recovery phase in step 2. Note that the proposer doesn't have to remember the promises or receive them again from the acceptors (they are not used when sending new proposals); it only relies on the fact that they were once sent by the acceptors to deduce that it's safe to send new proposals in subsequent slots using ballot num.

- The action then ensures that we have read messages from a quorum of replicas. It requires that for each such message r, the patch $r.p$ that it contains was sent by replica $r.rep$ when it satisfied $last[r.rep] = slot - 1$, by checking that $r.last = slot - 1$:

$$\begin{aligned}
&\exists Q \in RQuorum : \\
&\quad \exists rs \in \text{SUBSET } \{r \in reads : r.rep \in Q \land r.last = slot - 1\} : \\
&\qquad Q \subseteq \{r.rep : r \in rs\}
\end{aligned}$$

  This implies the following:

  - a patch must have already been chosen in slot $slot - 1$, since otherwise the replicas wouldn't apply it and would not send $r.last = slot - 1$. Let p_{slot-1} be the patch.
  - A quorum of replicas has applied the patch p_{slot-1}.
  - The patch $r.p$ for each $r \in rs$ is "recent enough", i.e. it was produced after the replica applied p_{slot-1}.
  - If we merge the patches $r.p$, we'll obtain s_{slot-1} — this is a nontrivial fact and will be shown as part of the correctness proof in the next section.

- Finally, if all the preconditions hold, the proposer can issue a new proposal. The decree is calculated using the input from a client request and the state obtained by merging the received patches:

$$
\begin{array}{l}
\exists req \in requests : \\
\quad LET\ res \triangleq Delta(MergeSet(\{r.p : r \in rs\}), req.input) \\
\quad IN\ proposals' = proposals \cup \\
\qquad \{[num \mapsto num, \\
\qquad\ dec \mapsto [id \mapsto req.id, p \mapsto res[1], o \mapsto res[2]], \\
\qquad\ slot \mapsto slot]\}.
\end{array}
$$

The $Accept(acc)$ action denotes the acceptor acc accepting a proposal. This is exactly what we've previously described in the outline.

$Apply(rep)$ denotes the replica rep applying a patch $p.dec.p$, where p is a chosen proposal (meaning that the patch $p.dec.p$ is chosen in slot $p.slot$). The precondition of this action is exactly the definition of a proposal being chosen — it was accepted by a quorum of acceptors.

A direct interpretation of this precondition would be that rep receives accept messages from a quorum of acceptors. However, nothing prevents an implementation to have another process, like a proposer, receiving those messages, and then forwarding the information about the chosen proposal to the replica. The replica only requires some proof that the precondition is true and needs to know what the chosen patch is; this information can be obtained directly from acceptors or indirectly through a middle-man such as the proposer. Implementations are free to choose either method, but they differ in the number of messages that need to be sent, as we discussed in the outline (step 7).

Respond is the action of sending a response to the client. As in *Apply*, the precondion is that a decree was chosen; the output stored in the decree is the client response. The only effect of *Respond* is updating the *history* variable which we'll later use in the proof. It doesn't affect the execution of the algorithm otherwise (it doesn't affect the preconditions of other actions).

There is an additional precondition which says that the history doesn't already have a response event for the request identifier stored in the chosen decree. We included it only so that the history has at most one response for a single request. Note however that the specification does not prevent a request to be executed multiple times; that is, one request may be used multiple times by the *ProposeNew* action to

calculate a decree, and each of these decrees may become chosen (in different slots). In particular, the identifier of this request may appear in multiple decrees chosen in different slots. In effect clients of the system may potentially receive multiple responses to a single request. We record only the first reponse in *history* since it will later make stating and proving the correctness condition easier. Clients are free to ignore all but the first response to each request.

The problem of a request being executed multiple times is further discussed in section 4.2.

The specification finishes by defining the initial state (*Init*) and the state transition relation (*Next*); both should be clear. Observe that we don't use the balnum 0 for anything except defining the initial state, and slot 0 is used only in *Init* and in the *Repropose* action to check that we're not trying to use the $\perp$ proposal.

Note that we never remove messages (the sets *requests*, *proposals*, etc. can only grow with each step). The reader may ask if it contradicts what we've said in the system model section, that messages may be dropped (or duplicated, or reordered). In fact it does not. Dropping a message can be represented by an execution in which the message is simply never received — even though the message is present in the set, nothing in the specification forces any of the actions which would use that message to be performed. Using a set of messages is also sufficient to model duplication (because upon receiving a message it is not removed from the set, so it can be received again) and reordering (messages present in the set can be received in any order). Using sets to represent messages in the network is a standard pattern in TLA+ specifications.

3.5. Proof of correctness

In this section we prove the correctness of LPaxos.

But what does it mean for the algorithm to be correct? We will show that the algorithm can be used to implement a *linearizable service*. Informally it means that all possibly observed sequences of requests and responses appear as if the requests were executed by a single copy of the given patch machine, exactly once sometime after sending the request but before receiving the response (if there is any). The requests can happen concurrently (one can start before another is answered to) but they appear to be executed in some total order from the clients' perspective.

The above intuition is formalized as theorem 3.5.3. But before we can prove that theorem, we must go through a long path of showing a bunch of "smaller" statements about the algorithm. Each statement is formed as an *invariant*: a formula that should be true in every state of every execution of the algorithm. We take a *bottom-up* approach: we start by stating and proving very simple invariants (which may sometimes even seem obvious) and use them as building blocks to prove more complex statements. This approach should help the reader build some intuition and understand the algorithm better as we go.

3.5.1. Type correctness

The TLA+ language is untyped, or rather every value has the same type: it is a set (TLA+ is based on the ZF set theory). By default that's all we know. Therefore the meaning of expressions like $a < b$ is not immediately specified. Only after we prove that $a, b \in \mathbb{Z}$ (for example) we are able to say that this is a comparison of integers

and the result is either $TRUE$ or $FALSE$. Knowledge about types is necessary before we make any meaningful non-trivial statements.

Thus we will start by proving a *type correctness invariant*, a formula saying that each variable belongs to a specific set which we'll call its *type*.

Type correctness will be implicitly used all the time in further steps of the proof. For example, if we know that

$$maxNum \in [Acceptor \to Balnum],$$

it will be implicitly understood what an expression like "$maxNum[acc] < num$" for acceptor acc and ballot number num means: $maxNum$ is a function that takes an acceptor and returns a balnum, hence $maxNum[acc]$ is a balnum; $Balnum$ is assumed to be a subset of Nat by the specification, so $maxNum[acc] < num$ is the result of integer comparison, hence the value of this expression is either $TRUE$ or $FALSE$.

First we need a couple of auxiliary definitions:

$$\begin{aligned}
&Decree \triangleq [id : ReqId,\ p : Patch,\ o : Output] \\
&NullProp \triangleq \{[num \mapsto 0,\ slot \mapsto 0]\} \\
&Proposal \triangleq [num : Balnum,\ slot : Nat \setminus \{0\}, dec : Decree]
\end{aligned}$$

$Decree$ is the set of *decrees*; each decree has a request identifier, a patch, and an output.

$NullProp$ is a set with a single record, $[num \mapsto 0,\ slot \mapsto 0]$, which is the null proposal $\perp$ that we've already defined in the outline. Recall that this is the value of $maxProp[acc]$ for each $acc \in Acceptor$ in the initial state.

$Proposal$ is the set of actual proposals that proposers send during the algorithm. A proposal has a balnum, a positive slot, and a decree.

The type correctness formula follows:

$$\begin{aligned}
&TypeOK \triangleq \\
&\quad \land history \in Seq([type : \{\text{"}response\text{"}\},\ id : ReqId,\ output : Output] \\
&\qquad\qquad\qquad \cup [type : \{\text{"}request\text{"}\},\ id : ReqId,\ input : Input]) \\
&\quad \land maxNum \in [Acceptor \to Balnum] \\
&\quad \land maxProp \in [Acceptor \to NullProp \cup Proposal] \\
&\quad \land store \in [Replica \to Patch] \\
&\quad \land last \in [Replica \to Nat] \\
&\quad \land requests \subseteq [id : ReqId,\ input : Input] \\
&\quad \land reads \subseteq [rep : Replica,\ p : Patch,\ last : Nat] \\
&\quad \land promises \subseteq [acc : Acceptor,\ num : Balnum,\ mprop : NullProp \cup Proposal] \\
&\quad \land proposals \subseteq Proposal \\
&\quad \land accepts \subseteq [acc : Acceptor,\ p : Proposal]
\end{aligned}$$

$TypeOK$ is a standard name for type correctness formulas in TLA+ specifications. It is a simple conjuction of independent statements, each statement specifying the type

of one of the algorithm's variables. The first conjuct says that *history* is a sequence, where each element of the sequence is either a response with request identifier and output, or a request with request identifier and input; *maxNum* is a function from *Acceptor* to *Balnum*; and so on.

We want to prove that *TypeOK* is an *invariant* of the specification. This means that *TypeOK* is true *in every state of every execution* of the algorithm. An *execution* (also called a *behavior*) is a sequence of states such that the initial state satisfies the *Init* predicate, and every consecutive pair of states in the sequence (called a *step*) satisfies the *Next* relation. Thus, to prove invariance, we can use induction; we prove two statements:

1. Base case: *TypeOK* is true in all states satisfying the *Init* predicate (in the case of our algorithm there is exactly one such state).
2. Inductive step: for every *step* of the algorithm, i.e. two states s and s' that satisfy the *Next* relation, if *TypeOK* is true in s (inductive hypothesis), then *TypeOK* is true in s'.

Proving the base case amounts to mechanically checking each conjuct of the *Init* predicate. For example, $maxNum = [a \in Acceptor \mapsto 0]$ obviously satisfies $maxNum \in [Acceptor \to Balnum]$, etc. We leave the rest of the checks to the reader.

To prove the inductive step we consider any step of the algorithm (s, s') and analyze the *Next* relation. It is a disjunction of statements which we refer to as *actions*. Thus we can analyze it on a case-by-case basis: we assume that *TypeOK* is true in s, and

- prove that it is true in s', given that $\exists i \in Input : Request(i)$ is true for (s, s'),
- prove that it is true in s', given that $\exists acc \in Acceptor, num \in Balnum \setminus \{0\} : Promise(acc, num)$ is true for (s, s'),

and so on, for each action. By checking all actions we cover all cases, proving that *TypeOK* is true in s' for every step (s, s') such that it is true in s. This is a standard pattern that we use in many proofs further in the work.

In the context of a step (s, s') of the algorithm, we will use the term *current state* to refer to s and the term *next state* to refer to s'.

We will be extensively using the standard TLA+ *priming* notation in this and following proofs. Suppose x is a variable. Given a step, when we write "x", this is a shortcut for "the value of x in the currrent state". When we write "x'", this is a shortcut for "the value of x in in the next state". When we want to refer to the variable itself (and not to the value of this variable in any state), we will say "the variable x".

For example, the following argument that doesn't use the priming notation:

> "Consider a step. Let x_1 be the value of x in the current state and x_2 be the value of x in the next state. By definition of *Next*, $x_2 = x_1 + 1$, therefore $x_2 > x_1$. In other words, x increases in each step."

would be written with the priming notation as:

> "Consider a step. By definition of *Next*, $x' = x + 1$, therefore $x' > x$. In other words, the value of (the variable) x increases in each step."

The priming notation extends to any TLA+ expression that contains variables. If e is an expression, then e' is the expression e with all variables replaced by their primed versions. For example, $(x + y)'$ denotes $x' + y'$.

Using this notation, the inductive step in the proof of invariance of *TypeOK* can be briefly stated as follows:

$$\text{for every step of the algorithm, if } \mathit{TypeOK} \text{ then } \mathit{TypeOK}'.$$

We will be often defining and using TLA+ operators that refer to the variables of the specification. If *Op* is an operator, then $Op'(e)$ for any expression e is $Op(e)$ but with all variables from the definition of *Op* replaced by their primed versions; this extends to operators with any number of parameters. For example, consider the following operator definition:

$$\mathit{AddX}(\mathit{arg}) \triangleq \mathit{arg} + x,$$

where x is a variable. Then

$$\mathit{AddX}'(\mathit{arg}) = \mathit{arg} + x'.$$

The argument of the operator itself may be an expression with primed or non-primed variables. The "priming" of the operator is independent from the "priming" of the expression. For example, if y is a variable, then:

$$\begin{aligned}
&\mathit{AddX}(y) = y + x \\
&\mathit{AddX}'(y) = y + x' \\
&\mathit{AddX}(y') = y' + x \\
&\mathit{AddX}'(y') = y' + x'.
\end{aligned}$$

In the proof of *TypeOK* below, when we say that "x is *type correct*", where x is one of our algorithm's variables, we mean that its corresponding conjuct in *TypeOK* is true in the current state. Similarily, when we say that "x' is type correct", we mean that its corresponding conjuct in *TypeOK* is true in the next state. Thus, for example, if we say that "$\mathit{proposals}'$ is type correct", we mean that $\mathit{proposals}' \subseteq \mathit{Proposal}$. Thus, to prove that TypeOK' is true, we need to show that x' is type correct for every variable x.

We are ready to show the inductive step. We analyze the actions in the order they appear in the specification, starting with *Request*.

Suppose that a step of the algorithm satisfies $\mathit{Request}(i)$ for $i \in \mathit{Input}$ and *TypeOK* holds; we need to show TypeOK'. By definition of *Request*, we have

$$\mathit{requests}' = \mathit{requests} \cup \{[\mathit{id} \to \mathit{id},\ \mathit{input} \to i]\},$$

where

- $\mathit{id} \in \mathit{ReqId}$,
- $\mathit{requests} \subseteq [\mathit{id} : \mathit{ReqId},\ \mathit{input} : \mathit{Input}]$,
- $[\mathit{id} \to \mathit{id},\ \mathit{input} \to i] \in [\mathit{id} : \mathit{ReqId},\ \mathit{input} : \mathit{Input}]$,

therefore $requests' \subseteq [id : ReqId,\ input : Input]$, i.e. $requests'$ is type correct.

Furthermore, by definition,

$$history' = Append(history, [type \mapsto \text{"}request\text{"},\ id \mapsto id,\ input \mapsto i])$$

where

- $history \in Seq([type : \{\text{"}response\text{"}\},\ id : ReqId,\ output : Output] \cup [type : \{\text{"}request\text{"}\},\ id : ReqId,\ input : Input])$,
- $[type \mapsto \text{"}request\text{"},\ id \mapsto id,\ input \mapsto i] \in [type : \{\text{"}request\text{"}\},\ id : ReqId,\ input : Input] \subseteq ([type : \{\text{"}response\text{"}\},\ id : ReqId,\ output : Output] \cup [type : \{\text{"}request\text{"}\},\ id : ReqId,\ input : Input])$,

  since $id \in ReqId$ and $i \in Input$,
- for a set S, if $x \in Seq(S)$ and $y \in S$ then $Append(x, y) \in Seq(S)$ ($Append$ is imported from the *Sequences* standard module),

therefore

$$history' \in Seq([type : \{\text{"}response\text{"}\},\ id : ReqId,\ output : Output] \cup [type : \{\text{"}request\text{"}\},\ id : ReqId,\ input : Input])$$

i.e. $history'$ is type correct.

The other variables ($maxNum$, $maxProp$, etc.) remain unchanged, so their type correctness in the next state follows immediately from type correctness in the current state. This covers the case of the *Request* action.

We did a lot of work just to deal with *Request*. But the truth is the work wasn't hard, it was mechanical, however tedious. The same is true for the other actions. When dealing with the rest of the actions below we only point out the core facts needed to prove the inductive step and leave the gaps to be filled by the reader.

When analyzing other actions, we will be often referring to symbols that the actions quantify over without explicitly introducing them ourselves in order to shorten the proofs. For example, the *Repropose* action introduces a $maxp$ symbol in the "$\land\ \exists maxp \in ms$:" line; we will simply write $maxp$ and assume that the reader understands the intent. The reader will find it useful to have the specification opened on the side when reading the arguments.

- $Promise(acc, num)$ where $acc \in Accept$ and $num \in Balnum$:

  by $TypeOK$ we have $maxProp[acc] \in NullProp \cup Proposal$, so

  $$m = [acc \mapsto acc, num \mapsto num, mprop \mapsto maxProp[acc]] \in [acc : Acceptor, num : Balnum, mprop : NullProp \cup Proposal],$$

  hence $promises' = promises \cup \{m\}$ is type correct.

  By $maxNum \in [Acceptor \to Balnum]$, $acc \in Acceptor$ and $num \in Balnum$ we get type correctness of $maxNum'$.

  Other variables remain unchanged.

- $Read(rep)$ for $rep \in Replica$: $store[rep] \in Patch$ and $last[rep] \in Nat$ by $TypeOK$, which implies that $reads'$ is type correct. Other variables are unchanged.
- $ProposeNew(num, slot)$ for $num \in Balnum$, $slot \in Nat \setminus \{0\}$: to prove that $proposals'$ is type correct we need to prove that the new proposal is an element of $Proposal$. For this the only nonobvious fact is that

  $$[id \mapsto req.id,\ p \mapsto res[1],\ o \mapsto res[2]] \in Decree,$$

  where $req \in requests$, $res = Delta(MergeSet(\{r.p : r \in rs\}), req.input)$, and $rs \subseteq reads$. We need to show that $req.id \in ReqId$, $res[1] \in Patch$, and $res[2] \in Output$.

  Since $req \in requests$ and $requests$ is type correct, $req.id \in ReqId$ and $req.input \in Input$. Since $rs \subseteq reads$ and $reads$ is type correct, $\{r.p : r \in rs\} \subseteq Patch$; by definition of $MergeSet$, nonemptiness of the set $\{r.p : r \in rs\}$ (by the $Q \subseteq \{r.rep : r \in rs\}$ condition) and the assumptions about $Merge$, $MergeSet(\{r.p : r \in rs\}) \in Patch$, hence, by the assumption on $Delta$, $res \in Patch \times Output$. Thus, $res[1] \in Patch$ and $res[2] \in Output$.
- $Repropose(num)$ for $num \in Balnum$: since $maxp \in ms \subseteq promises$ and $promises$ is type correct, we have $maxp.mprop \in NullProp \cup Proposal$. Furthermore, $maxp.mprop.slot \neq 0$, so $maxp.mprop \notin NullProp$, hence $maxp.mprop \in Proposal$. Therefore $maxp.mprop.dec \in Decree$ and $maxp.mprop.slot \in Nat$, so the new proposal belongs to $Proposal$, therefore $proposals'$ is type correct.
- $Accept(acc)$, $acc \in Acceptor$: since $p \in proposals$ and $proposals$ is type correct, $p \in Proposal$ and $p.num \in Balnum$. Type correctness of $maxNum'$, $maxProp'$, and $accepts'$ is then obvious.
- $Apply(rep)$, $rep \in Replica$: by $TypeOK$, $store[rep] \in Patch$ and $p.dec.p \in Patch$, therefore $Merge(store[rep], p.dec.p) \in Patch$ by the assumptions on $Merge$. Type correctness of $store'$ follows. Similarily, $last[rep] \in Nat$ and $p.slot \in Nat$, so $Max(last[rep], p.slot) \in Nat$, hence $last'$ is type correct.
- $Respond$: since $p \in proposals$ and $proposals$ is type correct, $p.dec.id \in ReqId$ and $p.dec.o \in Output$. Hence $history'$ is type correct.

□

We have shown that $TypeOK$ is true in every state in every execution of the algorithm. We are now allowed to implicitly assume $TypeOK$ in all our further arguments: when we write expressions like $p.dec.id$ where $p \in proposals$, they automatically "make sense" — e.g. we know that $p \in Proposal$, hence $p.dec \in Decree$, hence $p.dec.id \in ReqId$.

3.5.2. Consensus

The goal of this subsection is to prove that consensus is achieved on the decree chosen in each slot. This is formalized as corollary 3.5.2.

For now we view decrees abstractly; we don't care what's inside them (in this subsection it doesn't matter that they have patches or request identifiers). The proofs don't assume the types of the decrees. The contents of the chosen decrees will be inspected in the next subsection.

Lemma 3.5.1. *The following is an invariant.*

$$AcceptingProposals \triangleq$$
$$\forall a \in accepts : a.p \in proposals$$

Proof. The initial state satisfies *AcceptingProposals* because $accepts = \emptyset$.

Consider a step. We assume *AcceptingProposals* and need to show *AcceptingProposals*$'$.

Note that $proposals \subseteq proposals'$ for every action — we never remove proposals. Thus, if $a \in accepts$, $a.p \in proposals'$ because $a.p \in proposals$ by *AcceptingProposals*. Hence we only need to look at the $Accept(acc)$ action, which is the only action that may modify *accepts*.

By definition of $Accept(acc)$, if $a \in accepts'$, then either $a \in accepts$, which we covered above, or $\exists p \in proposals : a = [acc \mapsto acc,\ p \mapsto p]$, so $a.p \in proposals$ by definition, hence $a.p \in proposals'$. □

Lemma 3.5.2. *For every step of the algorithm,*

$$\forall acc \in Acceptor : maxNum[acc] \leq maxNum'[acc].$$

Proof. The only two actions that modify *maxNum* are *Promise* and *Accept*.

For $Promise(acc, num)$ we have

$$maxNum'[acc] = num > maxNum[acc]$$

by definition of *maxNum*$'$ and the precondition on $maxNum[acc]$. For acceptors other than *acc*, *maxNum* is equal to *maxNum*$'$.

For $Accept(acc)$, by definition we have $p \in proposals$ such that

$$maxNum[acc] \geq p.num = maxNum'[acc].$$

For other acceptors *maxNum*$'$ is equal to *maxNum*. □

The next two invariants state that $maxNum[acc]$ for $acc \in Acceptor$ is the highest balnum the acceptor has ever seen (either in an accepted proposal or by making a *num*-promise).

Lemma 3.5.3. *The following is an invariant.*

$$MaxNumPromises \triangleq$$
$$\forall m \in promises : m.num \leq maxNum[m.acc]$$

Proof. The statement is true in the initial state since $promises = \emptyset$.

For every step, if $m \in promises$, then $m.num \leq maxNum[m.acc] \leq maxNum'[m.acc]$ by lemma 3.5.2. If $m \in promises' \setminus promises$, then $Promise(acc, num)$ must be true for some $acc \in Acceptor$ and $num \in Balnum$, since no other actions modify *promises*. By definition of this action,

$$m = [acc \mapsto acc,\ num \mapsto num,\ mprop \mapsto maxProp[acc]]$$

and $num = maxNum'[acc]$. Since $num = m.num$ and $acc = m.acc$, $m.num = maxNum'[m.acc]$, which gives *MaxNumPromises*$'$. □

Lemma 3.5.4.

$$MaxNumAccepts \triangleq \\ \forall a \in accepts : a.p.num \leq maxNum[a.acc]$$

is an invariant.

Proof. True in the initial state since $accepts = \emptyset$.

For every step, if $a \in accepts$, then $a.p.num \leq maxNum[a.acc] \leq maxNum'[a.acc]$ by lemma 3.5.2. If $a \in accepts' \setminus accepts$, then, since no action other than $Accept$ modifies $accepts$, $Accept(acc)$ must be true for some $acc \in Acceptor$; by definition, there is $p \in proposals$ such that

$$a = [acc \mapsto acc,\ p \mapsto p]$$

with $maxNum'[acc] = p.num$. Thus, since $p = a.p$ and $acc = a.acc$, $maxNum'[a.acc] = a.p.num$, which gives $MaxNumAccepts'$. □

The next two invariants say that $maxProp[acc]$ is either the null proposal or a proposal that acc has accepted, and that is has the greatest slot and balnum among all proposals accepted by acc.

Lemma 3.5.5.

$$MaxPropIsAccepted \triangleq \\ \forall acc \in Acceptor : \\ \lor\ maxProp[acc] \in NullProp \\ \lor\ [acc \mapsto acc,\ p \mapsto maxProp[acc]] \in accepts$$

is an invariant.

Proof. In the initial state, $maxProp[acc] \in NullProp$ for every $acc \in Acceptor$.

Consider a step and let $acc \in Acceptor$. The case $maxProp'[acc] = maxProp[acc]$ is easy since $accepts \subseteq accepts'$.

Suppose $maxProp'[acc] \neq maxProp[acc]$. Then $Accept(acc)$ must be true; no other action modifies $maxProp$. By definition of $Accept$, there is $p \in proposals$ such that $maxProp'[acc] = p$ and $[acc \mapsto acc,\ p \mapsto p] \in accepts'$, which gives the formula. □

Lemma 3.5.6. *For every step,*

$$\forall acc \in Acceptor : \\ \land\ maxProp[acc].num \leq maxProp'[acc].num \\ \land\ maxProp[acc].slot \leq maxProp'[acc].slot$$

Proof. The only action modifying $maxProp$ is $Accept$. Let $acc \in Acceptor$ and suppose that $maxProp'[acc] \neq maxProp[acc]$. By definition of $Accept$, $maxProp'[acc] = p$, where $p.slot \geq maxProp[acc].slot$, which gives the second conjuct.

Furthermore, $p.num \geq maxNum[acc]$ by the $Accept$ preconditions, so it is enough to prove that $maxNum[acc] \geq maxProp[acc].num$. If $maxProp[acc].num = 0$, there's

nothing to show (0 is the smallest balnum). Otherwise, by *MaxPropIsAccepted*, there is $a \in accepts$ with $a.acc = acc$ and $a.p = maxProp[acc]$; by *MaxNumAccepts*, $a.p.num \leq maxNum[a.acc]$, therefore $maxProp[acc].num \leq maxNum[acc]$. □

Lemma 3.5.7.

$$\begin{aligned} &MaxPropIsMaximal \triangleq \\ &\quad \forall a \in accepts : \\ &\quad\quad \land\ a.p.slot \leq maxProp[a.acc].slot \\ &\quad\quad \land\ a.p.num \leq maxProp[a.acc].num \end{aligned}$$

is an invariant.

Proof. True in the initial state since $accepts = \emptyset$.

For a step, if $a \in accepts$, then using lemma 3.5.6 and the inductive hypothesis,

$$a.p.slot \leq maxProp[acc].slot \leq maxProp'[acc].slot$$

and

$$a.p.num \leq maxProp[acc].num \leq maxProp'[acc].num$$

.

The other case is $a \in accepts' \setminus accepts$. Then, by definition of *Accept* (which is the only action modifying *accepts*), $a.p = maxProp'[a.acc]$, which immediately gives the proposition. □

We will now prove that acceptors "don't lie" when making promises: if $m \in promises$, then $m.mprop$ is indeed the greatest proposal accepted by $m.acc$ among all proposals previously accepted by $m.acc$, i.e. among all proposals accepted by $m.acc$ with balnums lower than $m.num$. *Greatest* means that the proposal has the greatest slot and balnum among those proposals. Furthermore, acceptors keep their promises: $m.mprop$ will remain the greatest among these proposals. Thus, after making a promise m, the acceptor $m.acc$ will only accept proposals with balnums greater or equal than $m.num$.

For $m \in promises$, let

$$PromAcc(m) \triangleq \{a \in accepts : a.acc = m.acc \land a.p.num < m.num\}.$$

Lemma 3.5.8.

$$\begin{aligned} &KeepingPromises \triangleq \\ &\quad \forall m \in promises : \\ &\quad\quad \land\ \forall a \in PromAcc(m) : \\ &\quad\quad\quad \land\ a.p.slot \leq m.mprop.slot \\ &\quad\quad\quad \land\ a.p.num \leq m.mprop.num \\ &\quad\quad \land\ \lor\ m.mprop \in NullProp \\ &\quad\quad\quad \lor\ m.mprop \in \{a.p : a \in PromAcc(m)\} \end{aligned}$$

is an invariant.

Proof. True in the initial state since $promises = \emptyset$.

Consider a step. If neither $Promise(acc, num)$ nor $Accept(acc)$ are true, then $accepts' = accepts$ and $promises' = promises$, so $PromAcc'(m) = PromAcc(m)$ for each $m \in promises$, hence none of the subformulas of $KeepingPromises$ change their values in the next state compared to the current state (so they are true by inductive hypothesis). Thus we only need to consider the actions $Promise$ and $Accept$.

Suppose that $Promise(acc, num)$ is true. Let $m \in promises'$. If $m \in promises$, then $PromAcc'(m) = PromAcc(m)$ since $accepts' = accepts$. Therefore the two conjucts in $KeepingPromises'$ are true since their values are the same as in $KeepingPromises$.

The other case is $m \in promises' \setminus promises$. By definition of $Promise$,

$$m = [acc \mapsto acc,\ num \mapsto num,\ mprop \mapsto maxProp[acc]].$$

Let $a \in PromAcc'(m)$. By $m.mprop = maxProp[acc]$, $a.acc = m.acc = acc$, and $MaxPropIsMaximal'$, we have $a.p.slot \leq m.mprop.slot$ and $a.p.num \leq m.mprop.num$. This gives the first conjuct of $KeepingPromises'$.

For the second conjuct, suppose $m.mprop \notin NullProp$. Using $MaxPropIsAccepted$ and $m.mprop = maxProp[acc]$, let $a \in accepts$ be such that $a.acc = acc$ and $a.p = m.mprop$. By $MaxNumAccepts$ and $a.acc = acc$, $a.p.num \leq maxNum[acc]$. But $maxNum[acc] < num$ by $Promise$ preconditions, and $num = m.num$, so $a.p.num < m.num$. Since $accepts \subseteq accepts'$, $a \in accepts'$. Thus, by definition of $PromAcc'$, $a.p \in PromAcc'(m)$. Since $a.p = m.mprop$, we get

$$m.mprop \in \{a.p : a \in PromAcc'(m)\}$$

which gives the desired proposition.

Now suppose that $Accept(acc)$ is true. Let $m \in promises'$. By definition of $Accept$, $promises' = promises$, so $m \in promises$.

Suppose $a \in accepts' \setminus accepts$. We have $accepts \subseteq accepts' = accepts \cup \{a\}$, hence

$$PromAcc(m) \subseteq PromAcc'(m) \subseteq PromAcc(m) \cup \{a\}.$$

If $a.acc \neq m.acc$ then $a \notin PromAcc'(m)$, so $PromAcc'(m) = PromAcc(m)$. Otherwise, using $MaxNumPromises$ and $a.acc = acc$ we have $m.num \leq maxNum[acc]$. By $Accept$ preconditions, $maxNum[acc] \leq a.p.num$. Thus $m.num \leq a.p.num$, hence $a \notin PromAcc(m)$, hence $PromAcc'(m) = PromAcc(m)$.

We've shown that $PromAcc'(m) = PromAcc(m)$, which means that the values of the two conjucts in $KeepingPromises'$ are the same as in $KeepingPromises$, so they are true by the inductive hypothesis.

□

For $num \in Balnum$ and $A \subseteq Acceptor$, define

$$PrevProm(A, num) \triangleq \{m \in promises : m.acc \in A \wedge m.num = num\}$$

$PrevProm(A, num)$ are *promises that acceptors from A made for balnum num*. Every member of A promised that they won't accept any more proposals before num (i.e.

proposals with balnums smaller than num). By *KeepingPromises* we know that with each promise m comes the greatest proposal, $m.mprop$, accepted before num by the promising acceptor $m.acc$.

For $num \in Balnum$, $A \subseteq Acceptor$ and $slot \in \mathbb{N}$, define

$$\begin{array}{l} PromisedEmpty(A, num, slot) \triangleq \\ \quad \exists ms \subseteq PrevProm(A, num) : \\ \quad\quad \land A \subseteq \{m.acc : m \in ms\} \\ \quad\quad \land \forall m \in ms : m.mprop.slot < slot. \end{array}$$

The formula $PromisedEmpty(A, num, slot)$ says that every member of A promised that they have not and will not accept proposals before num in slots greater or equal to $slot$. The greatest accepted proposal before num, if any, belongs to a slot smaller than $slot$.

For $num \in Balnum$, $A \subseteq Acceptor$, $slot \in \mathbb{N}$ and $dec \in Decree$, define

$$\begin{array}{l} PromisedDecree(A, num, slot, dec) \triangleq \\ \quad \exists ms \subseteq PrevProm(A, num) : \\ \quad\quad \land A \subseteq \{m.acc : m \in ms\} \\ \quad\quad \land \exists m_{max} \in ms : \\ \quad\quad\quad \land m_{max}.mprop.slot = slot \\ \quad\quad\quad \land m_{max}.mprop.dec = dec \\ \quad\quad\quad \land \forall m \in ms : \\ \quad\quad\quad\quad \lor slot > m.mprop.slot \\ \quad\quad\quad\quad \lor \land slot = m.mprop.slot \\ \quad\quad\quad\quad\quad \land m_{max}.mprop.num \geq m.mprop.num \end{array}$$

The formula $PromisedDecree(A, num, slot, dec)$ says the following. Every member of A made a promise for balnum num. Among these promises is a "greatest" one: m_{max}. This promise contains a proposal, $m_{max}.mprop$, with slot $slot$, and this slot is the greatest slot among all proposals accepted by members of A before num. When compared to other proposals in $slot$ accepted by members of A before num, $m_{max}.mprop$ has the greatest balnum. The decree of this greatest proposal is dec.

Note that even if $PromisedDecree(A, num, slot, dec)$ is true, some member of A might have accepted a proposal with balnum greater than $m_{max}.mprop.num$, but then by the formula it must belong to a smaller slot. That is, there might be $m \in ms$ such that $m.mprop.num > m_{max}.mprop.num$, but then $m.mprop.slot < m_{max}.mprop.slot$.

For $p \in proposals$, define

$$\begin{array}{l} PromisedSafe(p) \triangleq \\ \quad \exists Q \in AQuorum : \\ \quad\quad \lor PromisedEmpty(Q, p.num, p.slot) \\ \quad\quad \lor PromisedDecree(Q, p.num, p.slot, p.dec) \end{array}$$

$PromisedSafe(p)$ says that a quorum of acceptors (i.e. a subset of acceptors containing a majority) Q has promised that all proposals accepted before p (i.e. with

balnums smaller than $p.num$) by members of Q are in slots smaller or equal to $p.slot$, and either they are all in strictly smaller slots (the first disjunct), or $p.dec$ is equal to the decree of the highest-numbered proposal among the subset of these accepted proposals that are in slot $p.slot$ (the second disjunct). If $PromisedSafe(p)$ is true we say that p is *promised to be safe*.

We will now prove that proposals are not issued unless they are promised to be safe. Furthermore, a proposal that was promised to be safe remains promised to be safe.

Lemma 3.5.9.

$$PropsPromisedSafe \triangleq \\ \forall p \in proposals : PromisedSafe(p)$$

is an invariant.

Proof. In the initial state $proposals = \emptyset$ so the formula is true.

Consider a step. The only actions we need to analyze are $Promise$, $ProposeNew$ and $Repropose$; for if one of the other actions is true, then $promises' = promises$ and $proposals' = proposals$, so $PromisedSafe'(p) = PromisedSafe(p)$ for each $p \in proposals$ (the formula expanded depends only on the variable $promises$) and therefore $PropsPromisedSafe' = PropsPromisedSafe$ by $proposals' = proposals$.

Suppose $Promise(acc, num)$ is true, $acc \in Acceptor$, $num \in Balnum$. Let $p \in proposals'$. Since $proposals' = proposals$ for this action, we have $Q \in AQuorum$ such that either $PromisedEmpty(Q, p.num, p.slot)$ or $PromisedDecree(Q, p.num, p.slot, p.dec)$ is true by the inductive hypothesis.

Let $m \in promises' \setminus promises$, so by definition of $Promise(acc, num)$, $m.acc = acc$ and $m.num = num$. It is easy to see that

$$PrevProm(Q, p.num) \subseteq PrevProm'(Q, p.num) \subseteq PrevProm(Q, p.num) \cup \{m\}.$$

Suppose that $m.acc \in Q$. By the inductive hypothesis, $Q \subseteq \{mm.acc : mm \in ms\}$ where $ms \subseteq PrevProm(Q, p.num)$, so we have $mm \in promises$ such that $mm.acc = m.acc$ and $mm.num = p.num$. By $MaxNumPromises$, $mm.num \leq maxNum[mm.acc]$. By $mm.acc = m.acc = acc$, $maxNum[mm.acc] = maxNum[acc]$. By the preconditions of $Promise$, $maxNum[acc] < num$. Thus

$$mm.num \leq maxNum[mm.acc] = maxNum[acc] < num = m.num,$$

but $p.num = mm.num$, so $p.num < m.num$, which gives $m \notin PrevProm'(Q, p.num)$. Therefore $PrevProm(Q, p.num) = PrevProm'(Q, p.num)$.

Let

$$F \triangleq \\ \quad \vee PromisedEmpty(Q, p.num, p.slot) \\ \quad \vee PromisedDecree(Q, p.num, p.slot, p.dec).$$

The formula F depends only on $PrevProm(Q, p.num)$. We have shown that $PrevProm'(Q, p.num) = PrevProm(Q, p.num)$, which gives $F' = F$. We know that F is true by choice of Q, hence F' is true. That gives $PromisedSafe'(p)$.

Now suppose $Repropose(num)$ is true, $num \in Balnum$. Let $p \in proposals'$.

If $p \in proposals$, then $PromisedSafe(p)$ is true by the inductive hypothesis, which implies that $PromisedSafe'(p)$ is true as well, since the formula (after expanding all definitions) depends only on the $promises$ variable, but $promises' = promises$ for this action.

Suppose then that $p \in proposals' \setminus proposals$. By definition of $Repropose$ let $Q \in AQuorum$, $ms \subseteq PrevProm(Q, num)$, and $m_{max} \in ms$ be such that

$$
\begin{array}{l}
\land\ Q \subseteq \{m.acc : m \in ms\} \\
\land\ \forall m \in ms : \\
\quad \lor\ m_{max}.mprop.slot > m.mprop.slot \\
\quad \lor\ \land m_{max}.mprop.slot = m.mprop.slot \\
\qquad \land\ m_{max}.mprop.num \geq m.mprop.num \\
\land\ m_{max}.mprop.slot \neq 0 \\
\land\ proposals' = proposals \cup \\
\quad \{[num \mapsto num,\ dec \mapsto m_{max}.mprop.dec,\ slot \mapsto m_{max}.mprop.slot]\}
\end{array}
$$

(m_{max} corresponds to $maxp$ from the specification). Because $p \notin proposals$,

$$p = [num \mapsto num,\ dec \mapsto m_{max}.mprop.dec,\ slot \mapsto m_{max}.mprop.slot],$$

that is,

$$
\begin{array}{l}
\land\ m_{max}.mprop.dec = p.dec \\
\land\ m_{max}.mprop.slot = p.slot
\end{array}
$$

and $PrevProm(Q, num) = PrevProm(Q, p.num)$.

Furthermore, $PrevProm(Q, p.num) = PrevProm'(Q, p.num)$ since $promises' = promises$, so $ms \subseteq PrevProm'(Q, p.num)$.

This gives $PromisedDecree'(Q, p.num, p.slot, p.dec)$ which gives $PromisedSafe'(p)$.

Finally, suppose $ProposeNew(num, slot)$ is true, $num \in Balnum$, $slot \in \mathbb{N}_+$.

Let $p \in proposals'$. If $p \in proposals$ we use the same argument as for $Repropose$ to deduce that $PromisedSafe'(p)$. Hence suppose $p \in proposals' \setminus proposals$, so $p.slot = slot$ and $p.num = num$ by definition of $ProposeNew$.

By the same definition, we have $Q \in AQuorum$ and $ms \subseteq PrevProm(Q, num)$ such that

$$
\begin{array}{l}
\land\ Q \subseteq \{m.acc : m \in ms\} \\
\land\ \forall m \in ms : m.mprop.slot < slot.
\end{array}
$$

Since $p.num = num$, $PrevProm(Q, num) = PrevProm(Q, p.num)$. Since $promises' = promises$, $PrevProm(Q, p.num) = PrevProm'(Q, p.num)$. Thus

$$ms \subseteq PrevProm'(Q, p.num)$$

which with the above two conjucts gives us $PromisedEmpty'(Q, p.num, p.slot)$. $\square$

Lemma 3.5.10.

$$PropPosSlots \triangleq \\ \forall p \in proposals : p.slot > 0$$

is an invariant.

Proof. $ProposeNew(num, slot)$ is true only for $slot \in \mathbb{N}_+$ (definition of $Next$). If $Repropose$ is true and $p \in proposals' \setminus proposals$, then $p.slot = maxp.mprop.slot$ where $maxp \in promises$ and $maxp.mprop.slot \neq 0$, so $maxp.mprop.slot > 0$. Other actions don't modify $proposals$. □

We will now prove that the algorithm maintains property P1 from the outline.

For $A \subseteq Acceptor$, $num \in Balnum$, $slot \in \mathbb{N}$, and $dec \in Decree$, let

$$PrevAcc(A, num, slot) \triangleq \\ \{a \in accepts : a.acc \in A \land a.p.slot = slot \land a.p.num < num\},$$

$$PrevDecree(A, num, slot, dec) \triangleq \\ \exists a_{max} \in PrevAcc(A, num, slot) : \\ \land \forall a \in PrevAcc(A, num, slot) : a_{max}.p.num \geq a.p.num \\ \land a_{max}.p.dec = dec.$$

For $p \in proposal$, let

$$Safe(p) \triangleq \\ \exists Q \in AQuorum : \\ \lor PrevAcc(Q, p.num, p.slot) = \emptyset \\ \lor PrevDecree(Q, p.num, p.slot, p.dec).$$

Lemma 3.5.11.

$$PropsSafe \triangleq \forall p \in proposals : Safe(p)$$

is an invariant.

Proof. We won't analyze actions but use already proved invariants. Thus consider any state in any execution. Let $p \in proposals$. By $PropsPromisedSafe$ we have $PromisedSafe(p)$, so let $Q \in AQuorum$ be such that

$$\lor PromisedEmpty(Q, p.num, p.slot) \\ \lor PromisedDecree(Q, p.num, p.slot, p.dec).$$

Consider the first case: $PromisedEmpty(Q, p.num, p.slot)$. Let $a \in accepts$ be such that $a.acc \in Q$ and $a.p.num < p.num$. We will show that $a.p.slot \neq p.slot$, hence $a \notin PrevAcc(Q, p.num, p.slot)$, so $PrevAcc(Q, p.num, p.slot) = \emptyset$.

Indeed: by definition of $PromisedEmpty$ and $a.acc \in Q$, let $ms \subseteq PrevProm(Q, p.num)$ and $m \in ms$ be such that $m.acc = a.acc$ and $m.mprop.slot < p.slot$. By $a.p.num <$

$p.num$ and $p.num = m.num$ we have $a.p.num < m.num$, so $a \in PromAcc(m)$. By *KeepingPromises*, $a.p.slot \leq m.mprop.slot$, but $m.mprop.slot < p.slot$, hence $a.p.slot < p.slot$.

Now consider the other case: $PromisedDecree(Q, p.num, p.slot, p.dec)$. By definition let $ms \subseteq PrevProm(Q, p.num)$ and $m_{max} \in ms$ be such that

$$
\begin{aligned}
&\land\ Q \subseteq \{m.acc : m \in ms\} \\
&\land\ m_{max}.mprop.slot = p.slot \\
&\land\ m_{max}.mprop.dec = p.dec \\
&\land\ \forall m \in ms : \\
&\qquad \lor\ p.slot > m.mprop.slot \\
&\qquad \lor\ \land\ p.slot = m.mprop.slot \\
&\qquad\quad \land\ m_{max}.mprop.num \geq m.mprop.num
\end{aligned}
$$

Since $p \in proposals$, $p.slot > 0$ by *PropPosSlots*. With $m_{max}.mprop.slot = p.slot$ this gives $m_{max}.mprop \notin NullProp$. By *KeepingPromises*, $m_{max}.mprop \in \{a.p : a \in PromAcc(m_{max})\}$. Hence we have $a_{max} \in PromAcc(m_{max})$ such that $a_{max}.p = m_{max}.mprop$.

By definition of $PromAcc(m_{max})$, $a_{max} \in accepts$, $a_{max}.acc = m_{max}.acc \in Q$ and $a_{max}.p.num < m_{max}.num$. Since $m_{max}.num = p.num$, $a_{max}.p.num < p.num$. By $a_{max}.p = m_{max}.mprop$ and $m_{max}.mprop.slot = p.slot$, $a_{max}.p.slot = p.slot$. Therefore $a_{max} \in PrevAcc(Q, p.num, p.slot)$.

Let $a \in PrevAcc(Q, p.num, p.slot)$. By $a.acc \in Q$ and $Q \subseteq \{m.acc : m \in ms\}$, let $m \in ms$ such that $m.acc = a.acc$. By $ms \subseteq PrevProm(Q, p.num)$, $m.num = p.num$. By definition of *PrevAcc*, $p.num > a.p.num$. Thus $m.num > a.p.num$, so $a \in PromAcc(m)$.

By *KeepingPromises*, $a.p.num \leq m.mprop.num$. By choice of m_{max} and $m \in ms$, there are two cases: either $p.slot > m.mprop.slot$ or $p.slot = m.mprop.slot$ and $m_{max}.mprop.num \geq m.mprop.num$.

The first case does not hold: by *KeepingPromises* and $a \in PromAcc(m)$, $a.p.slot \leq m.mprop.slot$, but $a.p.slot = p.slot$, so $p.slot \leq m.mprop.slot$.

Thus the second case holds: $m_{max}.mprop.num \geq m.mprop.num$. By $m.mprop.num \geq a.p.num$ and $m_{max}.mprop = a_{max}.p$, $a_{max}.p.num \geq a.p.num$.

This gives $\forall a \in PrevAcc(Q, p.num, p.slot) : a_{max}.p.num \geq a.p.num$.

Finally, $a_{max}.p.dec = m_{max}.mprop.dec = p.dec$.

The above arguments together show that a_{max} is a witness for $PrevDecree(Q, p.num, p.slot, p.dec)$[3].

By considering both cases of $PromisedSafe(p)$, we've shown $Safe(p)$. □

[3]For a formula "$\exists x : P(x)$", a *witness* is a value y such that $P(y)$ is true.

Lemma 3.5.12.

$$\begin{aligned} &\mathit{UniqueProposals} \triangleq \\ &\quad \forall p_1, p_2 \in \mathit{proposals} : \\ &\qquad (p_1.\mathit{slot} = p_2.\mathit{slot} \land p_1.\mathit{num} = p_2.\mathit{num}) \Rightarrow p1 = p2 \end{aligned}$$

is an invariant.

Proof. Immediate by the preconditions of *Repropose* and *ProposeNew*. Other actions don't modify *proposals*. □

A proposal is *chosen* if every member of some quorum of acceptors has accepted it. Formally, for $p \in \mathit{proposals}$, define

$$\begin{aligned} &\mathit{ChosenProposal}(p) \triangleq \\ &\quad \exists Q \in \mathit{AQuorum} : \\ &\qquad \forall \mathit{acc} \in Q : [\mathit{acc} \mapsto \mathit{acc},\ p \mapsto p] \in \mathit{accepts} \end{aligned}$$

Lemma 3.5.13. *Let $p_0 \in \mathit{proposals}$ and $Q_0 \in \mathit{AQuorum}$ be such that*

$$\forall \mathit{acc} \in Q_0 : [\mathit{acc} \mapsto \mathit{acc},\ p \mapsto p_0] \in \mathit{accepts}$$

(i.e. p_0 is chosen and Q_0 is the witness). Let $p_1 \in \mathit{proposals}$ be such that $p_1.\mathit{num} > p_0.\mathit{num}$ and $p_1.\mathit{slot} = p_0.\mathit{slot}$.

Suppose that for all $p \in \mathit{proposals}$ such that $p_1.\mathit{num} > p.\mathit{num} \geq p_0.\mathit{num}$ and $p.\mathit{slot} = p_0.\mathit{slot}$ it is true that $p.\mathit{dec} = p_0.\mathit{dec}$. Then

$$p_1.\mathit{dec} = p_0.\mathit{dec}$$

as well.

Proof. Since $p_1 \in \mathit{proposals}$, by *PropsSafe* we have $Q \in \mathit{AQuorum}$ such that

$$\begin{aligned} &\lor \mathit{PrevAcc}(Q, p_1.\mathit{num}, p_1.\mathit{slot}) = \emptyset \\ &\lor \mathit{PrevDecree}(Q, p_1.\mathit{num}, p_1.\mathit{slot}, p_1.\mathit{dec}). \end{aligned}$$

Because Q and Q_0 are both quorums, they intersect; let $\mathit{acc}_0 \in Q \cap Q_0$. Let $a_0 = [\mathit{acc} \mapsto \mathit{acc}_0, p \mapsto p_0]$. By definition of Q_0, $a_0 \in \mathit{accepts}$. We have $a_0.p.\mathit{num} = p_0.\mathit{num} < p_1.\mathit{num}$, $a_0.p.\mathit{slot} = p_0.\mathit{slot} = p_1.\mathit{slot}$ and $a_0.\mathit{acc} = \mathit{acc}_0 \in Q$,
so $a_0 \in \mathit{PrevAcc}(Q, p_1.\mathit{num}, p_1.\mathit{slot})$. Hence $\mathit{PrevDecree}(Q, p_1.\mathit{num}, p_1.\mathit{slot}, p_1.\mathit{dec})$, the second disjunct of the above formula, must be true.

Therefore let $a_{max} \in \mathit{PrevAcc}(Q, p_1.\mathit{num}, p_1.\mathit{slot})$ be such that

$$\begin{aligned} &\land \forall a \in \mathit{PrevAcc}(Q, p_1.\mathit{num}, p_1.\mathit{slot}) : a_{max}.p.\mathit{num} \geq a.p.\mathit{num} \\ &\land a_{max}.p.\mathit{dec} = p_1.\mathit{dec}. \end{aligned}$$

By the first conjuct and $a_0 \in \mathit{PrevAcc}(Q, p_1.\mathit{num}, p_1.\mathit{slot})$, $a_{max}.p.\mathit{num} \geq a_0.p.\mathit{num} = p_0.\mathit{num}$. By definition of *PrevAcc*, $a_{max}.p.\mathit{slot} = p_1.\mathit{slot}$ and $a_{max}.p.\mathit{num} < p_1.\mathit{num}$. Finally, $a_{max}.p \in \mathit{proposals}$ by *AcceptingProposals*. Thus, by assumptions of the lemma, $a_{max}.p.\mathit{dec} = p_0.\mathit{dec}$. But then the second conjuct gives $p_1.\mathit{dec} = p_0.\mathit{dec}$. □

Theorem 3.5.1. *For all $p_0, p_1 \in proposals$ such that $ChosenProposal(p_0)$, $p_1.slot = p_0.slot$ and $p_1.num \geq p_0.num$,*

$$p_1.dec = p_0.dec.$$

Proof. Let $p_0 \in proposals$ be such that $ChosenProposal(p_0)$. Let

$$\begin{aligned} B = \{b \in Balnum : \\ &\wedge b \geq p_0.num \\ &\wedge \exists p_1 \in proposals : \\ &\quad p_1.num = b \wedge p_1.slot = p_0.slot \wedge p_1.dec \neq p_0.dec\}. \end{aligned}$$

This is the set of balnums of all proposals p_1 as in the theorem assumptions (i.e. same slot and higher balnum than p_0), but that don't satisfy the theorem (different decree). To prove the theorem it is sufficient to show that $B = \emptyset$.

Suppose that B is nonempty. By well-ordering of $Balnum$, B has a least element. Let b_1 be this element.

By definition of B, let $p_1 \in proposals$ be such that $p_1.num = b_1$, $p_1.slot = p_0.slot$ and $p_1.dec \neq p_0.dec$. Note that $p_1.num = b_1 \geq p_0.num$.

If $p_1.num = p_0.num$ then *UniqueProposals* gives a contradiction. Therefore $p_1.num > p_0.num$. Now, for any $p \in proposals$ satisfying $p.slot = p_0.slot$ and $p_1.num > p.num \geq p_0.num$, we have $p.num \notin B$ since $p_1.num$ is the least element of B, hence $p.dec = p_0.dec$. By lemma 3.5.13, $p_1.dec = p_0.dec$, a contradiction. □

Corollary 3.5.1. *If $p_1, p_2 \in proposals$ such that $p_1.slot = p_2.slot$ both satisfy $ChosenProposal$, then $p_1.dec = p_0.dec$.*

Proof. Observe that one of $p_1.num \geq p_2.num$ or $p_2.num \geq p_1.num$ is true and apply theorem 3.5.1. □

For slot $s \in \mathbb{N}_+$, define the set of *decrees chosen in slot s*:

$$Chosen(s) \triangleq \{p.dec : p \in proposals \wedge p.slot = s \wedge ChosenProposal(p)\}.$$

Corollary 3.5.2 (Consensus)**.** *For each $s \in \mathbb{N}_+$, $Chosen(s)$ is either empty or has one element.*

We will also extend *Chosen* to the slot 0:

$$Chosen(0) = \{p_0\},$$

where $p_0 = p0 \in Patch$ is the initial patch from the specification. Since $p0$ is a constant, $Chosen(0)$ is also constant.

3.5.3. Connecting the decrees

We have achieved consensus — only a single decree can be decided in each slot.

By *TypeOK* we know that the decrees contain patches. Are those patches related in any way? The answer is given in theorem 3.5.2 which is the goal of this subsection.

Lemma 3.5.14. *Consider a step of the algorithm. For $s \in \mathbb{N}$, if $dec \in Chosen(s)$ then $dec \in Chosen'(s)$.*

In other words: after a decree has been decided, it remains decided.

Proof. For $s = 0$, $Chosen(s)$ is constant by definition. Let $s > 0$ and $dec \in Chosen(s)$. Then there is $p \in proposals$ such that $p.slot = s$, $p.dec = dec$, and $ChosenProposal(p)$ holds. But since $accepts \subseteq accepts'$, $ChosenProposal'(p)$ also holds: we can use the same witness $Q \in AQuorum$ as for $ChosenProposal(p)$. Therefore $dec \in Chosen'(p)$. □

Lemma 3.5.15.

$$\begin{aligned} ChosenInLast \triangleq \\ \forall rep \in Replica : \\ \land\ last[rep] \geq 0 \\ \land\ Chosen(last[rep]) \neq \emptyset \end{aligned}$$

is an invariant.

Proof. In the initial state, $last[rep] = 0$ for every $rep \in Replica$, so the formula is true by definition of $Chosen(0)$.

Consider a step. If $last'[rep] = last[rep]$, then $Chosen'(last'[rep]) \neq \emptyset$ by inductive hypothesis and lemma 3.5.14.

Otherwise, $Apply(rep)$ must be true; no other action modifies $last$. By definition of $Apply$, there is $p \in proposals$ such that $ChosenProposal(p)$ is true (and therefore $ChosenProposal'(p)$ as well) and $last'[rep] = Max(last[rep], p.slot)$; since $last'[rep] \neq last[rep]$, $last'[rep] = p.slot$. By definition of $Chosen$, $Chosen'(p.slot) \neq \emptyset$. Finally, $last'[rep] \geq 0$ since $p.slot \geq 0$. □

Lemma 3.5.16.

$$\begin{aligned} ChosenInReadLast \triangleq \\ \forall r \in reads : \\ \land\ r.last \geq 0 \\ \land\ Chosen(r.last) \neq \emptyset \end{aligned}$$

is an invariant.

Proof. In the initial state $reads = \emptyset$ so the formula holds.

For a step, let $r \in reads'$. If $r \in reads$ then use the inductive hypothesis and lemma 3.5.14. Otherwise r has been added to $reads$ by the $Read$ action; by the action's definition, $r.last = last[r.rep] = last'[r.rep]$, but $last'[r.rep] \geq 0$ and $Chosen'(last'[r.rep]) \neq \emptyset$ by $ChosenInLast'$. □

Lemma 3.5.17.

$$\begin{aligned} PropsFollowReads \triangleq \\ \forall p \in proposals : \\ \exists r \in reads : p.slot = r.last + 1 \end{aligned}$$

is an invariant.

Proof. In the initial state $proposals = \emptyset$ so the formula holds.

For a step, let $p \in proposals'$. If $p \in proposals$, then by inductive hypothesis we have $r \in reads$ satisfying $p.slot = r.last + 1$. Since $reads \subseteq reads'$, r is a witness for the formula in the next state as well.

Otherwise $p \in proposals' \backslash proposals$ and p must have been added either by *Repropose* or *ProposeNew*. If it was added by *Repropose*, then $p.slot = maxp.mprop.slot$ where $maxp \in promises$ and $maxp.mprop.slot \neq 0$, so $maxp.mprop \in proposals$ by *KeepingPromises* and *AcceptingProposals*. Thus by the inductive assumption we have $r \in reads$ with $maxp.mprop.slot = r.last+1$. But $reads \subseteq reads'$, so $r \in reads'$, and $p.slot = maxp.mprop.slot$ which gives the formula.

If p was added by *ProposeNew*, then by definition of *ProposeNew* there is $Q \in RQuorum$ (in particular, $Q \neq \emptyset$) such that for all $rep \in Q$, there is $r \in reads$ with $r.rep = rep$ and $r.last = p.slot - 1$. Since $reads \subseteq reads'$, any such r is a witness for the formula. □

Observe that for $p \in proposals$, $Chosen(p.slot - 1)$ is well defined as $p.slot > 0$ by *PropPosSlots*.

Lemma 3.5.18.

$$\begin{aligned} &ConsecutiveProps \triangleq \\ &\quad \forall p \in proposals : \\ &\quad\quad Chosen(p.slot - 1) \neq \emptyset \end{aligned}$$

is an invariant.

Proof. Immediate from *PropsFollowReads* and *ChosenInReadLast*. □

Corollary 3.5.3.

$$\begin{aligned} &ConsecutiveDecrees \triangleq \\ &\quad \forall s \in \mathbb{N} : Chosen(s+1) \neq \emptyset \Rightarrow Chosen(s) \neq \emptyset. \end{aligned}$$

is an invariant.

The corollary is obvious by the previous lemma.

For every state of the algorithm, define the maximum slot in which a decree has been chosen:

$$MaxS \triangleq max(\{s \in \mathbb{N} : Chosen(s) \neq \emptyset\}).$$

This is well defined as the set of slots with chosen decrees is always finite; each action creates at most one new proposal, so there is a finite number of issued proposals in every state of any execution. The set is also non-empty since $Chosen(0) \neq \emptyset$ is an invariant.

MaxS is 0 in the initial state, and as the algorithm progresses and new proposals are chosen, *MaxS* increases one-by-one. By *ConsecutiveDecrees*,

$$\forall s \in \{0, \ldots, MaxS\} : Chosen(s) \neq \emptyset.$$

By corollary 3.5.2 the sets $Chosen(s)$ have at most one element. Thus, for each $s \in \{1, \ldots, MaxS\}$, we can define D_s to be the only element of $Chosen(s)$; i.e.

$$Chosen(s) = \{D_s\}.$$

By the definition of $Chosen$ we know that $D_s \in Decree$. For each $1 \leq s \leq MaxS$ define

$$p_s = D_s.p,$$

i.e. $p_s \in Patch$ is the patch of the sth decree. Recall that also $p_0 = p0 \in Patch$ is defined to be the only element of $Chosen(0)$.

Hence the algorithm constructs a sequence of patches, $p_1, \ldots, p_{MaxS}$. Soon we will prove an invariant which says that the sequence is a descendant sequence of patches for the patch machine $(Patch, Input, Output, Delta, p0, Merge)$.

In every state of any execution of the algorithm, the patch $store[rep]$ stored by each replica rep is the result of merging some of the patches chosen until now in some order; the patches may repeat, the sequence of merged patches starts with p_0 and contains $p_{last[rep]}$. Formally:

Lemma 3.5.19. *The following is an invariant. For each $rep \in Replica$, there exists a sequence of indices*

$$I_{rep} = \langle i_0, i_1, \ldots, i_{rep_n} \rangle,$$

where $0 \leq i_j \leq last[rep]$, such that

$$\begin{aligned} &\wedge\ i_0 = 0 \\ &\wedge\ \exists j \in \{0, \ldots, rep_n\} : i_j = last[rep] \\ &\wedge\ store[rep] = Merge(p_{i_0}, \ldots, p_{i_{rep_n}}). \end{aligned}$$

Recall that $last[rep] \leq MaxS$ (by $ChosenInReadLast$) so the p_{i_j}s are defined.

For $p_1, p_2, p_3 \in Patch$, the notation $Merge(p_1, p_2, p_3)$ means $Merge(Merge(p_1, p_2), p_3)$, but it also means $Merge(p_1, Merge(p_2, p_3))$ since we assumed that $Merge$ for patches is associative. Also define $Merge(p) = p$.

Proof. In the initial state, $last[rep] = 0$ and $store[rep] = p_0$ so $I_{rep} = (0)$ satisfies the conditions for every $rep \in Replica$.

Consider a step. Only the $Apply$ action modifies $store$ and $last$ so suppose that $Apply(rep)$ is true. Let $p \in proposals$ be such that

$$\begin{aligned} &\wedge\ store'[rep] = Merge(store[rep], p.dec.p) \\ &\wedge\ last'[rep] = Max(last[rep], p.slot). \end{aligned}$$

Let I_{rep} be such that

$$\begin{aligned} &\wedge\ i_0 = 0 \\ &\wedge\ \exists j \in \{0, \ldots, rep_n\} : i_j = last[rep] \\ &\wedge\ store[rep] = Merge(p_{i_0}, \ldots, p_{i_{rep_n}}) \end{aligned}$$

from the inductive assumption. Define

$$I'_{rep} = Append(I_{rep}, p.dec.p).$$

Then I'_{rep} satisfies the desired conditions. Indeed, take $rep'_n = rep_n + 1$ and $i_{rep'_n} = p.slot$; then $i_{rep'_n} \leq last'[rep]$. The precondition of *Apply* implies that $p.dec$ is chosen in slot $p.slot$, so $p_{i_{rep'_n}}$ is defined and equal to $p.dec.p$. By the equation on $store'[rep]$ and third conjuct for I_{rep}, we have

$$store'[rep] = Merge(Merge(p_{i_0}, \ldots, p_{i_{rep_n}}), p_{i_{rep'_n}}),$$

so the third conjuct for I'_{rep} is satisfied. The first conjuct is obvious.

For the second conjuct, suppose first that $last[rep] \leq p.slot$. Then $j = rep'_n$ satisfies $i_j = last'[rep]$ as $last'[rep] = p.slot = i_{rep'_n}$. Otherwise, $last[rep] > p.slot$, so $last'[rep] = last[rep]$; then take the same j as for I_{rep}.

For replicas other than rep, $store$ and $last$ are unchanged so the sequence from the current state is also good in the next state. □

Lemma 3.5.20. *The following is an invariant. For each $r \in reads$, there exists a sequence of indices*

$$I_r = \langle i_0, i_1, \ldots, i_{r_n} \rangle,$$

where $0 \leq i_j \leq r.last$, such that

$$\begin{aligned} &\wedge\ i_0 = 0 \\ &\wedge\ \exists j \in \{0, \ldots, r_n\} : i_j = r.last \\ &\wedge\ r.p = Merge(p_{i_0}, \ldots, p_{i_{r_n}}). \end{aligned}$$

Proof. The statement is true in the initial state since $reads = \emptyset$.

Consider a step and let $r \in reads'$. If $r \in reads$, then a sequence I_r from the inductive hypothesis continues to satisfy the formula in the next state since the chosen patches p_i don't change by lemma 3.5.14.

Hence suppose that $r \in reads' \backslash reads$. Then $Read(rep)$ must be true and $r.rep = rep$, $r.p = store[rep]$, and $r.last = last[rep]$. By lemma 3.5.19, there is a sequence $I_{rep} = \langle i_0, \ldots, i_{rep_n} \rangle$ with $0 \leq i_j \leq last[rep]$ such that

$$\begin{aligned} &\wedge\ i_0 = 0 \\ &\wedge\ \exists j \in \{0, \ldots, rep_n\} : i_j = last[rep] \\ &\wedge\ store[rep] = Merge(p_{i_0}, \ldots, p_{i_{rep_n}}). \end{aligned}$$

Take $I_r = I_{rep}$ and $r_n = rep_n$. By substituting $r.p$ for $store[rep]$ and $r.last$ for $last[rep]$ in the above formula, and observing that p_i don't change, we obtain the desired result. □

Given $rep \in Replica$ or $r \in reads$ we will use I_{rep} or I_r to refer to the sequences that were constructed in the proof of lemmas 3.5.19 and 3.5.20 respectively.

Note that there may be multiple sequences satisfying the statements of the lemmas; for example, if merging two consecutive patches in such a sequence commutes (and we will soon prove that indeed they all commute under *Merge*), we can reorder the patches and the lemmas will remain true.

But there is one "special" sequence I_{rep}, for $rep \in Replica$, given by the construction in the proof: it is obtained when rep merges its current $store[rep]$ with a new patch

in the *Apply* operation. This is the sequence we'll refer to. Similarily, there is one special sequence I_r for a read r.

If we wanted to make the following arguments completely formal, we would have to introduce an artificial variable into the specification itself for remembering the sequences. That is, we could introduce a variable $pseq \in [Replica \to Seq(Patch)]$ and extend the $Apply(rep)$ action as follows:

$$\land\ pseq' = [pseq\ EXCEPT\ ![rep] = Append(pseq[rep], p.dec.p)]$$

where $p.dec.p$ is the patch being merged (i.e. $store'[rep] = Merge(store[rep], p.dec.p)$). We would also remember this sequence in every $r \in reads$, e.g. using a field $r.pseq$. Then $pseq[rep]$ would be what we now refer to as I_{rep} and $r.pseq$ for $r \in reads$ would be I_r.

But since the variables are not used by the algorithm itself, only be the proof, we won't do that and instead leave it to the reader as an excercise (or have the reader trust us that it can be done in a sensible fashion).

When in the context of an algorithm step, we'll use I_{rep} to denote the sequence in the current state and I'_{rep} in the next state; same for I_r and I'_r.

For two sequences I_1 and I_2, we'll write $I_1 \sqsubseteq I_2$ to denote that I_1 is a prefix of I_2.

Lemma 3.5.21.

$$\begin{aligned} &ReadStorePrefix \triangleq \\ &\quad \forall r \in reads : \\ &\qquad \land\ I_r \sqsubseteq I_{r.rep} \\ &\qquad \land\ r.last \leq last[r.rep] \end{aligned}$$

is an invariant.

Proof. Consider a step. Observe that for all $rep \in Replica$, $I_{rep} \sqsubseteq I'_{rep}$. Indeed: the only action modifying I_{rep} is *Apply*, but that action only appends to the sequence as observed in the proof of lemma 3.5.19. Similarly, $last[rep] \leq last'[rep]$.

Let $r \in reads'$. If $r \in reads$ then the previous observation gives $I_r \sqsubseteq I_{r.rep} \sqsubseteq I'_{r.rep}$ and $r.last \leq last[rep] \leq last'[rep]$. Otherwise, $r \in reads' \setminus reads$ was added by the *Read* action, but then $I_r = I_{r.rep}$ and $r.last = last[r.rep] = last'[r.rep]$. □

Lemma 3.5.22.

$$\begin{aligned} &GrowingReadSequences \triangleq \\ &\quad \forall r_1, r_2 \in reads : \\ &\qquad (r_1.rep = r_2.rep \land r_1.last < r_2.last) \\ &\qquad\quad \Rightarrow I_{r_1} \sqsubseteq I_{r_2} \end{aligned}$$

is an invariant.

Proof. Consider a step. Let $r_1, r_2 \in reads'$, $r_1.rep = r_2.rep$ and $r_1.last < r_2.last$.

If $r_1, r_2 \in reads$ then $I_{r_1} \sqsubseteq I_{r_2}$ by the inductive hypothesis.

Otherwise, one of them has to be in *reads*, since at most one new read can be added to *reads* in a single step. Suppose that $r_2 \in reads$ but $r_1 \notin reads$; then r_1 was added by the *Read* action, hence $r_1.last = last[r_1.rep]$. But by *ReadStorePrefix*, $r_2.last \leq last[r_2.rep] = last[r_1.rep] = r_1.last$, which contradicts $r_1.last < r_2.last$.

Therefore $r_1 \in reads$ and $r_2 \notin reads$. Thus r_2 was added by *Read*, so $I_{r_2} = I_{r_2.rep}$. Using *ReadStorePrefix* we get

$$I_{r_1} \sqsubseteq I_{r_1.rep} = I_{r_2.rep} = I_{r_2}.$$

□

For $R \subseteq Replica$ and $slot \in \mathbb{N}_+$, define

$$Reads(R, slot) \triangleq \{r \in reads : r.rep \in R \wedge r.last = slot - 1\}.$$

Lemma 3.5.23.

$$\begin{array}{l} DeltaProposals \triangleq \\ \quad \forall p \in proposals : \exists req \in requests : \\ \quad\quad \exists Q \in RQuorum : \exists rs \subseteq Reads(Q, p.slot) : \\ \quad\quad\quad \wedge\ Q \subseteq \{r.rep : r \in rs\} \\ \quad\quad\quad \wedge\ (p.dec.p, p.dec.o) = Delta(MergeSet(\{r.p : r \in rs\}), req.input) \\ \quad\quad\quad \wedge\ p.dec.id = req.id \end{array}$$

is an invariant.

Proof. Consider a step and let $p \in proposals'$.

If $p \in proposals$, then let *req*, Q and *rs* be such as in the definition of *DeltaProposals*; they also prove *DeltaProposals*′, since $requests \subseteq requests'$ (so $req \in requests'$) and $reads \subseteq reads'$ (so $rs \subseteq Reads(Q, p.slot) \subseteq Reads'(Q, p.slot)$).

Suppose then that $p \in proposals' \setminus proposals$. p was added either by *Repropose* or *ProposeNew*. Suppose that $Repropose(num)$ is true. Then $p.dec = maxp.mprop.dec$ and $p.slot = maxp.mprop.slot$, where $maxp \in promises$ satisfies $maxp.mprop.slot \neq 0$, thus $maxp.mprop \notin NullProp$, so $maxp.mprop \in proposals$ by *KeepingPromises* and *AcceptingProposals*. Let $p_1 = maxp.mprop$. Since $p_1 \in proposals$, by the inductive hypothesis we have $req \in requests$, $Q \in RQuorum$ and $rs \subseteq Reads(Q, p_1.slot)$ such that

$$\begin{array}{l} \wedge\ Q \subseteq \{r.rep : r \in rs\} \\ \wedge\ (p_1.dec.p, p_1.dec.o) = Delta(MergeSet(\{r.p : r \in rs\}), req.input) \\ \wedge\ p_1.dec.id = req.id. \end{array}$$

As before, $req \in requests'$ and $rs \subseteq Reads'(Q.p_1.slot)$. Substituting $p.dec$ for $p_1.dec$ and $p.slot$ for $p_1.slot$ we obtain the desired formula for p.

Finally, suppose $ProposeNew(num, slot)$ is true. Then rs, Q and req are given by the action's precondition. □

For $0 \leq n \leq MaxS$, define

$$s_n = \begin{cases} p_0, n = 0 \\ Merge(s_{n-1}, p_n), n > 0. \end{cases}$$

Note that if $p \in proposals$, then $0 < p.slot \leq MaxS + 1$ by $ConsecutiveProps$, so $s_{p.slot-1}$ is defined.

Theorem 3.5.2.

$$\begin{aligned} &\forall p \in proposals : \\ &\quad \exists req \in requests : \\ &\qquad \wedge (p.dec.p, p.dec.o) = Delta(s_{p.slot-1}, req.input) \\ &\qquad \wedge p.dec.id = req.id \end{aligned}$$

is an invariant.

Proof. Consider a step. Let $p \in proposals'$. Let $n = p.slot - 1$.

We need to show

$$\begin{aligned} &\exists req \in requests' : \\ &\qquad \wedge (p.dec.p, p.dec.o) = Delta(s'_n, req.input) \\ &\qquad \wedge p.dec.id = req.id. \end{aligned}$$

But note that $requests \subseteq requests'$ and $s'_n = s_n$ as $p_i = p'_i$ for each $0 \leq i \leq n$ by lemma 3.5.14. Thus it is sufficient to show

$$\begin{aligned} &\exists req \in requests : \\ &\qquad \wedge (p.dec.p, p.dec.o) = Delta(s_n, req.input) \\ &\qquad \wedge p.dec.id = req.id. \end{aligned}$$

If $p \in proposals$, then there's nothing to prove — the statement is the inductive assumption. Hence suppose $p \in proposals' \setminus proposals$.

By $DeltaProposals$ let $req \in requests$, $Q \in RQuorum$, and $rs \subseteq Reads(Q, p.slot)$ be such that

$$\begin{aligned} &\wedge Q \subseteq \{r.rep : r \in rs\} \\ &\wedge (p.dec.p, p.dec.o) = Delta(MergeSet(\{r.p : r \in rs\}), req.input) \\ &\wedge p.dec.id = req.id. \end{aligned}$$

It is sufficient to show that

$$MergeSet(\{r.p : r \in rs\}) = s_n.$$

By the definition of $MergeSet$, there is an ordering of elements of rs,

$$r_1, \ldots, r_m,$$

where m is the size of rs, such that

$$s_n = Merge(r_1.p, \ldots, r_m.p).$$

Recall that for each r_j, $1 \leq j \leq m$, we have a sequence

$$I_{r_j} = \langle i_{j,0}, \ldots, i_{j,n_j} \rangle$$

where $0 \leq i_{j,k} \leq r_j.last$, $i_{j,0} = 0$ and I_{r_j} contains $r_j.last$, such that

$$r_j.p = Merge(p_{i_{j,0}}, \ldots, p_{i_{j,n_j}}).$$

Therefore

$$s_n = Merge(Merge(p_{i_{1,0}}, \ldots, p_{i_{1,n_1}}), \ldots, Merge(p_{i_{m,0}}, \ldots, p_{i_{m,n_m}})).$$

By associativity of $Merge$,

$$s_n = Merge(p_{i_{1,0}}, \ldots, p_{i_{1,n_1}}, \ldots, p_{i_{m,0}}, \ldots, p_{i_{m,n_m}}).$$

Let

$$I = \{i_{j,k} : 1 \leq j \leq m, 0 \leq k \leq n_j\}.$$

For each j, since $r_j \in rs \subseteq Reads(Q, p.slot)$ and $p.slot - 1 = n$, we have $r_j.last = n$. Therefore, by $0 \leq i_{j,k} \leq r_j.last$, $I \subseteq \{0, \ldots, n\}$. We will show that this inclusion is an equality.

Since $i_{1,0} = 0$, $0 \in I$.

Consider the chosen patches p_i, $0 < i \leq n$. By definition of $Chosen$, for each of these patches there is a proposal $prop_i \in proposals$ with $prop_i.slot = i$ and $prop_i.dec.p = p_i$. By $DeltaProposals$ applied to $prop_i$, we have $Q_i \in RQuorum$ and $rs_i \subseteq Reads(Q_i, i)$ such that $Q_i \subseteq \{r.rep : r \in rs_i\}$, for each i.

Let $rep_i \in Q \cap Q_i$. By $rep_i \in Q_i$ and the choice of Q_i, let $read_i \in rs_i$ be such that $read_i.rep = rep_i$. Note that $read_i.last = i$ by $rs_i \subseteq Reads(Q_i, i)$.

By $rep_i \in Q$ and the choice of Q, let $r_j \in rs$, $1 \leq j \leq m$, be such that $r_j.rep = rep_i$. Note that $r_j.last = n$ by $rs \subseteq Reads(Q, p.slot)$ and $p.slot = n + 1$.

If $i = n$ then $i = r_j.last$, and I_{r_j} contains $r_j.last$, ($i_{j,k} = r_j.last$ for some k), so $i \in I$.

Otherwise $i < n$, that is, $read_i.last < r_j.last$. Furthermore, $read_i.rep = rep_i = r_j.rep$. Thus, by $GrowingReadSequences$,

$$I_{read_i} \sqsubseteq I_{r_j}.$$

But the sequence I_{read_i} contains $read_i.last$, hence so does I_{r_j}. Thus

$$i = read_i.last \in \{i_{j,k} : 0 \leq k \leq n_j\}.$$

Therefore we've shown that the sequence

$$p_{i_{1,0}}, \ldots, p_{i_{1,n_1}}, \ldots, p_{i_{m,0}}, \ldots, p_{i_{m,n_m}}$$

lists each patch p_i, $0 \leq i \leq n$, at least once.

Consider again the proposals $prop_i$, $0 < i \leq n$. By the inductive assumption, for each of these proposals there is a request, req_i, such that

$$(prop_i.dec.p, prop_i.dec.o) = Delta(s_{i-1}, req_i.input)$$

In other words,

$$(p_i, out_i) = Delta(s_{i-1}, inp_i)$$

where $out_i = prop_i.dec.o \in Output$ and $inp_i = req_i.input \in Input$. Thus, the sequence

$$p_1, p_2, \ldots, p_n$$

is a descendant sequence of patches for $(Patch, Input, Output, Delta, p_0, Merge)$.

By the definition of a patch machine, $Merge$ commutes for the patches p_i. Using commutativity of $Merge$, in the expression

$$Merge(p_{i_{1,0}}, \ldots, p_{i_{1,n_1}}, \ldots, p_{i_{m,0}}, \ldots, p_{i_{m,n_m}})$$

we can reorder the patches so that the indices are non-decreasing, then using idempotency, replace subsequences of repeating patches by a single patch; by the previous result, each patch p_i, $0 \leq i \leq n$ appears exactly once in the resulting sequence. Thus we obtain

$$Merge(p_0, \ldots, p_n) = s_n.$$

□

3.5.4. Linearizability

In this subsection we establish a connection between the chosen decrees and the client requests. The goal is to prove that for idempotent patch machines (which we'll soon define), *history* is always linearizable. This is theorem 3.5.3.

Let

$$RequestedIds \triangleq \{req.id : req \in requests\}.$$

Lemma 3.5.24. *The following is an invariant.*

$$\forall req_1, req_2 \in requests : req_1.id = req_2.id \Rightarrow req_1 = req_2.$$

It follows immediately from the preconditions of *Request*.

By the above lemma, for each $id \in RequestedIds$ there is exactly one $req \in requests$ such that $req.id = id$. Let $req(id)$ denote this request.

Lemma 3.5.25. *The following is an invariant.*

$$\forall n \in \{1, \ldots, MaxS\} : D_n.id \in RequestedIds$$

(where D_n is the decree chosen in the nth slot).

This follows immediately from the definition of *ProposeNew* and the fact that $D_n = p.dec$ for some $p \in proposals$.

Thus, for $n \in \{1, \ldots, MaxS\}$, it makes sense to define

$$req_n = req(D_n.id)$$

Note that two proposals in different slots may use the same request, so for $i \neq j$ it may happen that $req_i = req_j$.

Recall the definition of s_n, for $0 \leq n \leq MaxS$:

$$s_n = \begin{cases} p_0, n = 0 \\ Merge(s_{n-1}, p_n), n > 0. \end{cases}$$

Lemma 3.5.26. *The following is an invariant.*

$$\begin{aligned} &\forall n \in \{1, \ldots, MaxS\} : \\ &\quad (p_n, D_n.o) = Delta(s_{n-1}, req_n.input). \end{aligned}$$

Proof. Let $p \in proposals$ be such that $p.dec = D_n$, so $p.slot = n$. By theorem 3.5.2, let *req* be such that $(p.dec.p, p.dec.o) = Delta(s_{n-1}, req.input)$ and $p.dec.id = req.id$. Substituting D_n for $p.dec$ in the previous equations and observing that $req = req_n$ (since $D_n.id = req.id$) gives the lemma. □

Below we'll use the following shorthand notation: if *seq* is a sequence, then "$x \in seq$" means $\exists i \in \{1, \ldots, Len(seq)\} : x = seq[i]$.

Lemma 3.5.27. *The following is an invariant.*

For every $e \in history$ such that $e.type =$ "response", $e.id = req_n.id$ for some $1 \leq n \leq MaxS$. Furthermore, $e.output = D_n.o$.

Proof. By the definition of *Respond*, $e.id = p.dec.id$ and $e.output = p.dec.o$ where $ChosenProposal(p)$ holds. Thus, for $n = p.slot$, $p.dec = D_n$, so $e.id = D_n.id = req_n.id$ and $e.output = D_n.o$. □

For $id \in RequestedIds$, let

$$slots(id) = \{n \in \{1, \ldots, MaxS\} : req_n.id = id\}.$$

The lemma above says in particular that for $e \in history$ with $e.type =$ "response", $slots(e.id) \neq \emptyset$.

Lemma 3.5.28. *For every step of the algorithm:*

$$\begin{aligned} &\forall id \in RequestedIds : \\ &\quad \forall n \in slots'(id) : \\ &\qquad \vee\ n \in slots(id) \\ &\qquad \vee\ n = MaxS' = MaxS + 1. \end{aligned}$$

Furthermore, if $id \in RequestedIds$ satisfies $slots(id) \neq \emptyset$, then

$$min(slots(id)) = min(slots'(id)).$$

Proof. Let $id \in RequestedIds$ and suppose that $n \in slots'(id) \setminus slots(id)$.

Let $k \in \{1, \ldots, MaxS\}$. Lemma 3.5.14 says that chosen decrees do not change, thus $D_k = D'_k$. In particular, $D_k.id = D'_k.id$ and $req'_k = req_k$. Suppose that $k \in slots'(id)$; by definition, $req'_k.id = id$, so $req_k.id = id$, hence $k \in slots(id)$, so $k \neq n$.

Therefore $n > MaxS$. But $n \leq MaxS'$ and $MaxS' \leq MaxS + 1$ (at most one proposal can be accepted in a single step, so at most one new decree can be chosen in a single step), so $MaxS < n \leq MaxS + 1$, meaning that $n = MaxS + 1$.

The second statement follows immediately from the fact that $slots(id) \subseteq slots'(id)$ (easy to see) and what we've just proven. □

Lemma 3.5.29. *For every step of the algorithm:*

$$\begin{aligned} \forall i \in \{1, \ldots, Len(history')\} : \\ (\wedge\ history'[i].type = \text{“}request\text{”} \\ \wedge\ slots'(history'[i].id) \neq \emptyset) \\ \Rightarrow i \leq Len(history). \end{aligned}$$

Intuitively, this says that in a single step we cannot both add a new request ($i > Len(history)$, $history'[i].type =$ “$request$”) and make a decision using this request ($slots'(history'[i].id) \neq \emptyset$). If we have a decision using this request at the end of the step then the request must have been added in an earlier step.

Proof. Suppose that $Len(history) < i \leq Len(history')$ and $history'[i].type =$ “$request$”. The only way to append to $history$ is either through $Request$ or $Response$ actions, and in a single step we add at most one event. From the event's type we deduce that the $Request$ action happened in this step and $i = Len(history')$.

Let $id = history'[i].id$. By the definition of $Request$ there is no $req \in requests$ such that $req.id = id$, i.e. $id \notin RequestedIds$.

Suppose that $slots'(id) \neq \emptyset$ and let $n \in slots'(id)$. By definition, $req'_n.id = id$, where req'_n is the only $req \in requests'$ with $req.id = D'_n.id$; in other words, $id = D'_n.id$. Since the $Request$ action happened, $MaxS' = MaxS$ (the only way to change the value of $MaxS$ is to decide a new decree, which can only happen in the $Accept$ action). Thus $n \leq MaxS$. By lemma 3.5.14, $D_n = D'_n$, so $id = D_n.id$. By lemma 3.5.25, $id \in RequestedIds$, which contradicts the previous paragraph.

Thus either $slots'(id) = \emptyset$, $history'[i].type \neq$ “$request$”, or $i \leq Len(history)$. □

The following lemma says that for two requests req_1, req_2 that got “executed” ($slots(req_1.id) \neq \emptyset$ and $slots(req_2.id) \neq \emptyset$), if the response to req_1 comes in $history$ before req_2 first appears, then the first execution of req_1 (i.e. $min(slots(req_1.id))$) comes before all executions of req_2.

Lemma 3.5.30. *The following is an invariant.*

Suppose that $i_1 \in \{1 \ldots Len(history)\}$ *satisfies* $history[i_1].type =$ “$response$”. *Let*

$$n_1 = min(slots(history[i_1].id)).$$

Then

$$\begin{aligned}\forall i_2 \in \{i_1 + 1, \ldots, Len(history)\} :& \\ history[i_2].type = \text{“}request\text{”}& \\ \Rightarrow \forall n \in slots(history[i_2].id) :& \\ n_1 < n.&\end{aligned}$$

Proof. The statement is true in the initial state since $Len(history) = 0$.

Consider a step of the algorithm and suppose the statement is true in the current state. Let

$$1 \leq i_1 < i_2 \leq Len(history')$$

be such that $history[i_1].type =$ “*response*”, $history[i_2].type =$ “*request*”, and let $n \in slots'(history'[i_2].id)$. We have to show that $n'_1 < n$, where

$$n'_1 = min(slots'(history'[i_1].id)).$$

Because $i_1 < Len(history')$, $i_1 \leq Len(history)$ (*history* can grow by at most one entry in each step). Because the *history* variable is only appended to (existing entries do not get modified in any step), $history[i_1] = history'[i_1]$. Therefore

$$n'_1 = min(slots'(history[i_1].id)).$$

By lemma 3.5.28, $n'_1 = n_1$, where

$$n_1 = min(slots(history[i_1].id)).$$

Thus we only need to show that $n_1 < n$.

Because $slots'(history'[i_2].id)$ is nonempty, $i_2 \leq Len(history)$ by lemma 3.5.29. As before, $history[i_2] = history'[i_2]$. Therefore $n \in slots'(history[i_2].id)$. Let

$$id_2 = history'[i_2].id = history[i_2].id.$$

Either $n \in slots(id_2)$ or $n \in slots'(id_2) \setminus slots(id_2)$. In the first case $n > n_1$ by the inductive assumption. In the second case, by lemma 3.5.28, $n = MaxS' = MaxS + 1 \geq n_1 + 1 > n_1$. □

A single request may be “executed” multiple times by the algorithm. Formally, $slots(id)$ may have more than one element for a given $id \in RequestedIds$. The reason is that even though a $req \in requests$ is used, it stays in the *requests* set; this models the possibility of the client message being duplicated.

But if we want to build a linearizable service, we want each request to appear as if it executed at most once after it was sent — and if it has a response, exactly once, some time between sending the request and receiving the response by the client. We can achieve this effect using *idempotent* patch machines, i.e. machines that do not change their state the second time they execute a given request, and the returned output is the same as it was when the request was executed for the first time.

Definition 3.5.1 (Idempotent machine)**.** A patch machine $(P, I, O, \delta, p_0, \sqcup)$ is **idempotent** if for every execution of the machine

$$s_0, i_1, s_1, o_1, i_2, s_2, o_2, i_3, s_3, o_3, \ldots,$$

for every $k < l$ such that $i_k = i_l$,

$$s_l = s_{l-1} \wedge o_l = o_k.$$

□

Note that the definition doesn't say anything about the produced patches. It might happen for $k < l$ that $p_k \neq p_l$, even though $(p_k, o_k) = \delta(s_{k-1}, i)$ and $(p_l, o_l) = \delta(s_{l-1}, i)$ (the same input was used), but then merging the second patch to the previous state gives the same state: $s_l = s_{l-1} \sqcup p_l = s_{l-1}$.

In section 4.2 we further discuss the problem of executing a request multiple times; in particular, we show how to "augument" any patch machine to turn it into an idempotent patch machine.

Let $\mathcal{M}$ denote the patch machine $(Patch, Input, Output, Delta, p0, Merge)$.

Given a sequence of IDs

$$I = \langle id_1, id_2, \ldots, id_n \rangle,$$

where $id_i \in RequestedIds$ for each $1 \leq i \leq n$, define $s_i^I \in Patch$, $p_i^I \in Patch$, $o_i^I \in Output$ by the following equations:

$$\begin{aligned} &s_0^I = s_0 = p_0 \\ &(p_i^I, o_i^I) = Delta(s_{i-1}^I, req(id_i).input) \text{ for } 1 \leq i \leq n, \\ &s_i^I = Merge(s_{i-1}^I, p_i^I) \text{ for } 1 \leq i \leq n \end{aligned}$$

(there is no o_0^I). That is, s_i^I is the ith state of an execution of $\mathcal{M}$ obtained by inputting $req(id_1).input$, $req(id_2).input$, and so on.

Theorem 3.5.3 (Linearizability)**.** *Suppose that $\mathcal{M}$ is idempotent. Then the following is an invariant.*

There exists a sequence of IDs

$$I = \langle id_1, id_2, \ldots, id_n \rangle,$$

where $id_i \in RequestedIds$, such that:

- *$id_{i_1} \neq id_{i_2}$ for $i_1 \neq i_2$.*
- *For every $e \in history$ such that $e.type =$ "response",*

  $$e.id = id_i$$

  for some $1 \leq i \leq n$. Furthermore,

  $$e.output = o_i^I.$$

- *Suppose that* $1 \leq j_1 < j_2 \leq Len(history)$ *and* $1 \leq i_1, i_2 \leq n$ *satisfy*

$$\begin{aligned} &\wedge\ history[j_1].type = \text{“}response\text{”} \\ &\wedge\ history[j_1].id = id_{i_1} \\ &\wedge\ history[j_2].type = \text{“}request\text{”} \\ &\wedge\ history[j_2].id = id_{i_2}. \end{aligned}$$

  Then $i_1 < i_2$.

The theorem states the following. There is an ordering of a subset of operations appearing in the history. The first bullet above says that each operation appears in the ordering at most once. The second bullet says that all operations that have a response must appear in the ordering (operations that do not have a response may or may not appear in the ordering), and that the outputs of the operations that do have a response are as if they were produced by a single instance of the machine $\mathcal{M}$ executing according to the ordering. The third bullet says that if the response for one operation comes before the request of another, the ordering places the first operation before the second.

This is a formalization of the intuitive notion that every operation appears to execute atomically, exactly once, between its request and response; operations without responses may or may not have been executed.

Proof. We will inductively define a sequence of slots $K = (k_1, \ldots, k_n)$ with chosen decrees. If $MaxS = 0$, then the sequence is empty. Otherwise, we start with slot $k_1 = 1$, and define k_{i+1} to be the first slot after k_i whose decree's ID is different than that of each previously picked slot.

Formally, let

$$\begin{aligned} &k_1 = 1 \\ &k_{i+1} = min(\{k : \\ &\qquad \wedge\ k_i < k \leq MaxS \\ &\qquad \wedge\ \forall 1 \leq j \leq i : D_{k_j}.id \neq D_k.id\}). \end{aligned}$$

This sequence ends as soon as there is no next slot with a different request ID (i.e. the set expression in the definition of k_{i+1} gives an empty set).

Then I is defined by $id_i = D_{k_i}.id$, for $1 \leq i \leq n$.

We will now prove that

$$\begin{aligned} &\wedge\ o_i^I = D_{k_i}.o \\ &\wedge\ s_i^I = s_{k_i}. \end{aligned}$$

The proof proceeds by induction on i. For $i = 1$,

$$\begin{aligned} (p_1^I, o_1^I) &= Delta(s_0^I, req(id_1).input) \\ &= Delta(s_0, req(D_1.id).input) = Delta(s_0, req_1.input), \end{aligned}$$

so $(p_1^I, o_1^I) = (p_1, o_1) = (p_{k_1}, o_{k_1})$ by lemma 3.5.26, and

$$s_1^I = Merge(s_0, p_1^I) = Merge(s_0, p_1) = s_1 = s_{k_1}.$$

Suppose that $1 < i \leq n$ and $o_j^I = D_{k_j}.o$ for every $1 \leq j < i$. We then have

$$\begin{aligned}(p_i^I, o_i^I) &= Delta(s_{i-1}^I, req(id_i).input) \\ &= Delta(s_{k_i-1}, req(D_{k_i}.id).input) = Delta(s_{k_i-1}, req_{k_i}.input).\end{aligned}$$

On the other hand, by lemma 3.5.26,

$$(p_{k_i}, D_{k_i}.o) = Delta(s_{k_i-1}, req_{k_i}.input).$$

Thus, if we prove $s_{k_i-1} = s_{k_i-1}$, we will get $(p_i^I, o_i^I) = (p_{k_i}, D_{k_i}.o)$, and

$$s_i^I = Merge(s_{i-1}^I, p_i^I) = Merge(s_{k_i-1}, p_{k_i}) = Merge(s_{k_i-1}, p_{k_i}) = s_{k_i}.$$

It only remains to observe that for each

$$k = k_{i-1} + 1, k_{i-1} + 2, \ldots, k_i - 1,$$

we have $s_{k-1} = s_k$. Indeed: we have $s_k = Merge(s_{k-1}, p_k)$, where $(p_k, D_k.o) = Delta(s_{k-1}, req_k.input)$. But by the definition of k_{i-1} and k_i, there is some $k' \leq k_{i-1}$ such that $D_k.id = D_{k'}.id$, so $req_k.input = req(D_k.id).input = req(D_{k'}.id).input = req_{k'}.input$. Thus we've used the same input to produce $s_{k'}$ and s_k; since $k' \leq k_{i-1} < k$ and $\mathcal{M}$ is idempotent, $Merge(s_{k-1}, p_k) = s_{k-1}$ by the definition of idempotency.

We are ready to prove that I satisfies the conditions of the theorem. The first bullet is immediate from the definition of the sequence K.

To prove the second bullet, suppose that $e \in history$ satisfies $e.type =$ "$response$". By lemma 3.5.27, there is k, $1 \leq k \leq MaxS$, with $e.id = req_k.id$ and $e.output = D_k.o$. Let $k_i \leq k$ be some element of K such that $D_{k_i}.id = D_k.id$; it is easy to see that such k_i exists directly from the definition of K. Then $req_{k_i} = req_k$ since they have equal IDs. Thus, by $req_{k_i}.input = req_k.input$ and idempotency of $\mathcal{M}$, $D_k.o = D_{k_i}.o$.

Therefore $e.id = D_{k_i}.id = id_i$ and $e.output = D_{k_i}.o = o_i^I$.

It remains to prove the third bullet. Let $1 \leq j_1 < j_2 \leq Len(history)$ and $1 \leq i_1, i_2 \leq n$ be such that

$$\begin{aligned}&\wedge\, history[j_1].type = \text{"}response\text{"} \\ &\wedge\, history[j_1].id = id_{i_1} \\ &\wedge\, history[j_2].type = \text{"}request\text{"} \\ &\wedge\, history[j_2].id = id_{i_2}.\end{aligned}$$

Observe that $k_{i_1} \in slots(history[j_1].id)$. Indeed, $req_{k_{i_1}}.id = D_{k_{i_1}}.id = id_{i_1} = history[j_1].id$.

Furthermore, by the definition of the sequence K, there is no $k < k_{i_1}$ with $req_k.id = req_{k_{i_1}}.id$. Thus

$$k_{i_1} = min(slots(history[j_1].id)).$$

Similarily, $k_{i_2} \in slots(history[j_2].id)$. By lemma 3.5.30,

$$k_{i_1} < k_{i_2}.$$

But the sequence K is increasing, so this implies $i_1 < i_2$. □

Chapter 4

Implementation

In this chapter we discuss some important problems and solutions needed in order to practically use the algorithm from the previous chapter. We also present *LattiStore*, a proof-of-concept implementation of a key-value store that uses LPaxos underneath for consistent transactions. The last two sections discuss ideas for further research to improve the algorithm in order to make it better suited for use in large production systems.

4.1. Reading large state

Recall that the $Read(rep)$ action specifies a replica rep sending a message with its currently stored patch:

$$\begin{aligned} &reads' = reads \cup \\ &\quad \{[rep \mapsto rep, p \mapsto store[rep], last \mapsto last[rep]]\} \end{aligned}$$

Each read represents partial information about the state. In the $ProposeNew(num, slot)$ action, the owner of num uses such patches retrieved from a quorum of replicas to calculate the state s_{slot-1} by merging the patches together:

$$\begin{aligned} &\exists Q \in RQuorum : \\ &\quad \exists rs \in SUBSET\ \{r \in reads : r.rep \in Q \wedge r.last = slot - 1\} : \\ &\qquad \wedge \ldots \\ &\qquad\quad LET\ res \triangleq Delta(MergeSet(\{r.p : r \in rs\}), req.input) \\ &\qquad\quad IN \ldots \end{aligned}$$

If one were to directly translate the specification to code, the resulting implementation would have each replica sending its entire stored state on each read request. With large states this would render the implementation impractical.

However, the reason for the specification sending the entire state is only simplicity of the spec itself. The implementation can use better methods of obtaining the state to calculate res, the result of $Delta$, to achieve the same result.

For example, certain types of patch machines, where the state can be "sharded" into small independent pieces, the transition function may require inspecting only a "small" part of the state in order to produce the output and the next patch. Before we formalize this notion let's look at a couple of examples.

Example 4.1.1. Consider the grow-set machine from example 2.2.2. Its set of states is $\mathbf{P}(\mathbb{Z})$, i.e. each state of the machine is a set of integers. The $add(x, y)$ command may add an integer to this set; hence the state can grow linearly with the length of the machine's execution.

The transition function δ is formally defined to be a function of the entire state, but it only cares whether or not the state contains a single element, it doesn't need all of the state to produce the correct output. Formally, it is immediate from the definition of δ that

$$\delta(s, cas(x, y)) = \delta(s \cap \{x\}, cas(x, y)).$$

Thus in the replication algorithm, when the proposer calculates the state

$$Delta(MergeSet(\{r.p : r \in rs\}), req.input),$$

which for this particular patch machine is

$$\delta(\bigcup_{r \in rs} r.p, cas(x, y))$$

for some $x, y \in \mathbb{Z}$ (and the patches $r.p$ are sets of integers), it can instead calculate

$$\delta((\bigcup_{r \in rs} r.p) \cap \{x\}, cas(x, y)).$$

But now we can use the fact that

$$(\bigcup_{r \in rs} r.p) \cap \{x\} = \bigcup_{r \in rs} (r.p \cap \{x\}),$$

so for the proposer to obtain the right result, each read message r doesn't need to contain the entire state $r.p$ sent by the replica; it needs only to contain $r.p \cap \{x\}$.

Thus, in an efficient implementation of the replication algorithm for this particular patch machine, the proposer wouldn't ask the replicas to send the entire state; it would only ask whether or not they contain x. Then, if all replicas in some quorum return the answer "I don't have x", the proposer can deduce that the current state indeed does not have x. □

Example 4.1.2. Consider the key-value store from example 2.2.3. Recall the definition of δ:

$$\delta((s, n), f) = ((ver(f(s'), n + 1), n + 1), s'), \text{ where } s' = unver(s).$$

The input f is an arbitrary function $(K \dashrightarrow V) \to (K \dashrightarrow V)$, so for δ to compute $f(s') = f(unver(s))$ it may require inspecting the value and version for every key present in s. Furthermore, the output is $s' = unver(s)$, which again requires looking at every key.

Each command to this key-value store returns the entire "previous state" of the store prior to the command's execution, and also has the possibility to inspect and modify the entire state. This is probably an overkill; in a real implementation, we may want the user to specify a small subset of keys that they are reading or modifying in a given request, and perhaps even limit the set of keys so that the queries don't become too large and overload our database.

We'll modify this example a bit by changing the set of inputs:

$$I = \{(D, f) : D \subseteq K, f \in (D \dashrightarrow V) \to (K \dashrightarrow V)\}.$$

This time an input is a pair: the first element is a subset D of the set of keys, specifying which keys are inspected by the function. The second element is a function that looks only at the values under the keys from D. The letter "D" comes from (function) domain.

We must modify δ to accomodate for the different input type:

$$\delta((s, n), (D, f)) = ((ver(f(s'|_D), n + 1), n + 1), s'|_D), \text{ where } s' = unver(s).$$

The symbol $s'|_D$ denotes the restriction of the partial function $s' : K \dashrightarrow V$ to domain $D \subseteq K$. Therefore $s'|_D : D \dashrightarrow V$, i.e. it is a map containing only keys from D. Thus we can apply f to that to obtain a new map, $f(s'|_D) : K \dashrightarrow V$. Notice that we still allow the produced patch to modify any key; an implementation would probably want to put a limit on this as well, but that's not the issue we're focusing on at the moment.

Thus we only need $s'|_D$. Observe that

$$s'|_D = unver(s)|_D = unver(s|_D).$$

We've formally defined *unver* to be a function of $(K \dashrightarrow V \times \mathbb{N})$, but the definition works for any $(D \dashrightarrow V \times \mathbb{N})$ where $D \subseteq K$.

Now suppose that $r_1.p = (p_1, n_1), \ldots, r_n.p = (p_k, n_k)$, where $r_1, \ldots, r_k$ are reads obtained from some quorum of replicas. The proposer calculates

$$(s, n) = (p_1, n_1) \sqcup \cdots \sqcup (p_k, n_k).$$

But we've just seen we don't need s, we only need $s|_D$. It only remains to observe that

$$(s|_D, n) = (p_1|_D, n_1) \sqcup \cdots \sqcup (p_k|_D, n_k).$$

Thus a smart proposer wouldn't ask the replicas to send the entire state; it would ask them to send only n and the tuples with keys from D (i.e. if $store[rep] = (p, n)$, *rep* would send only $(p|_D, n)$). Then the size of the exchanged messages is linear in the size of D.

This trick is used in the implementation of LattiStore. □

We can put the ideas from the above examples into a more general framework.

For a patch machine with a set of inputs I, suppose that each $i \in I$ has an associated *projection function*

$$\pi_i : P \to P,$$

which is distributive over $\sqcup$, i.e. $\pi_i(p_1 \sqcup p_2) = \pi_i(p_1) \sqcup \pi_i(p_2)$, such that δ satisfies

$$\delta(p, i) = \delta(\pi_i(p), i)$$

for each $p \in P$.

Intuitively, each patch represents partial information about the state; if the states are in some way "sharded", meaning that they can be somehow split into independent pieces, then the patches can also be split accordingly. Each projection π_i takes

a patch and "forgets" the information about some of these pieces. The distributivity expresses the intuitive notion that the merging operation $\sqcup$ merges the pieces independently, so if we drop a certain subset of pieces from two patches and then merge them, it's the same as if we first merged them and then dropped that subset of pieces. Finally, we drop only those pieces which are not needed by the transition function to correctly compute a result for the given input.

In example 4.1.1,

$$\pi_{cas(x,y)}(p) = p \cap \{x\}.$$

In example 4.1.2,

$$\pi_{(D,f)}((s, n)) = (\, s|_D \,, n).$$

When a proposer sends a read request to a replica, it will attach π_i, where i is the input that the proposer is currently handling. Each replica *rep* will then send a message with $\pi_i(store[rep])$ instead of the entire $store[rep]$.

4.2. Making patch machines idempotent

In section 3.5.4 we've proven that for idempotent patch machines, the replication algorithm guarantees that all observed histories of request-response events produced by the algorithm are linearizable — clients are given the illusion of working with a single instance of the machine executing sequentially.

For non-idempotent machines there is no such guarantee as the algorithm allows each request to be executed multiple times. This follows from the fact that we don't remove the request from the *requests* set (which represents the messages sent from clients to proposers) when it gets processed by a proposer; the request remains there, ready to be used in a later proposal.

One could argue: but if I send a message to a proposer and the proposer receives it, how can the proposer receive the message *again*?

It is often assumed in the distributed algorithms literature that messages can be lost, reordered, or even duplicated. The duplication of a message is what may cause a request to be handled twice.

However, in practice it is common to use protocols such as TCP [18] which guarantee that a single message is processed at most once. We can build our messaging protocol on top of TCP and enjoy this additional assumption if clients don't explicitly attempt to send a message twice — for example, when a client's TCP connection to the server gets broken and he retries sending the last message on a new connection.

Still, we want to give the choice to system developers. They can either try ensuring that no client request gets delivered twice, in which case the algorithm will provide linearizability for any patch machine (showing this would require a minor modification the spec), or they may want to, say, give the possibility for clients to retry their requests without losing the guarantee of having the request (appear to) execute at most once. They may also want to use a transport protocol other than TCP with better efficiency but weaker safety properties.

There are many useful examples of patch machines that are not idempotent, such as 2.2.1. Looking at its set of inputs, it would be very weird for this machine to be idempotent; after all, one would probably want to request $cas(0, 1)$ many times

during the service's lifetime, giving a meaningful result every time, not just for the first execution.

What we want to achieve is idempotency of *requests*. That is, two $cas(0,1)$ commands sent in two different requests should be executed independently, but if they are sent twice "in the same request", the second command should have no effect.

In the algorithm specification we've used request identifiers and provided each new request with a unique ID in order to reason about the different requests even if they contained the same input. Intuitively, if we want an input to have a real effect only the first time it executes *by a given request* (but not when executed by different requests), the input must somehow identify the request itself. We will now formalize this intuition.

Let $\mathcal{M} = (P, I, O, \delta, p_0, \sqcup)$ be a patch machine and $ReqId$ be a set of request identifiers.

Let $I' \subseteq ReqId \times I$ be such that if $(id_1, i_1), (id_2, i_2) \in I'$ satisfy $id_1 = id_2$, then $i_1 = i_2$. Thus it is a set of identifier and input pairs where the identifier uniquely defines the input. Or, in other words, it is a partial function from $ReqId$ to I.

For each $p \in P$, let $nop(p)$ be such that $p \sqcup nop(p) = p$. In particular, $nop(p) = p$ satisfies this property (by idempotency of $\sqcup$), but for some patch machines there might be a "simpler" $nop(p)$, e.g. $nop(p) = \emptyset$ works for the machine from example 2.2.2.

Let $\mathcal{M}' = (P', I', O, \delta', p_0', \sqcup')$, where

- $P' = P \times (ReqId \dashrightarrow O)$,
- $\delta'((p, f), (id, i)) = \begin{cases} ((nop(p), \emptyset), o), & \text{if } f(id) = o, \\ ((p', [id \mapsto o]), o), \text{where } (p', o) = \delta(p, i), & \text{if } id \notin dom(f), \end{cases}$
- $p_0' = (p_0, \emptyset)$,
- $(p_1, f_1) \sqcup' (p_2, f_2) = (p_1 \sqcup p_2, f_1 \sqcup f_2)$, where

$$dom(f_1 \sqcup f_2) = dom(f_1) \cup dom(f_2),$$
$$(f_1 \sqcup f_2)(id) = \begin{cases} f_1(id), \text{ if } id \in dom(f_1), \\ f_2(id), \text{ otherwise,} \end{cases} \quad \text{for } id \in dom(f_1) \cup dom(f_2).$$

In addition to the state that $\mathcal{M}$ stores, $\mathcal{M}'$ remembers the outputs of previously executed commands. If it is given the same command with the same request identifier as one that it has already seen, it will return the remembered output and produce a patch that will leave the state unmodified when merged with the current state. Otherwise, it will compute the next patch and the output using the original transition function, remember the output for the given identifier, and return it.

Idempotency of $\sqcup'$ is obvious. The proof of associativity is quite similar to the one in example 2.2.3 so we'll skip it. For commutativity, observe that if

$$(p_1, f_1), (p_2, f_2), (p_3, f_3), \ldots$$

is a descendant sequence of patches, then $dom(f_i) \cap dom(f_j) = \emptyset$ for $i < j$. This can be easily shown by induction. Thus $f_i \sqcup f_j = f_i \cup f_j$, which is a commutative operation.

The proof of the following lemma is not hard; we will thus give only a quick sketch.

Lemma 4.2.1. *Let*

$$(s_0, f_0), (id_1, i_1), (s_1, f_1), o_1, (id_2, i_2), (s_2, f_2), o_2, \ldots$$

be an execution of $\mathcal{M}'$.

If $i_k = i_j$ for $1 \leq j < k$, then $o_k = o_j$ and $(s_k, f_k) = (s_j, f_j)$ (idempotency of $\mathcal{M}'$).

Furthermore, let

$$k_0, k_1, k_2, \ldots$$

be the sequence of indices defined inductively as follows:

$$\begin{aligned} &k_0 = 0, \\ &k_{j+1} = min(\{k : k > k_j \wedge \forall 1 \leq j' \leq j : id_{k_{j'}} \neq id_k\}). \end{aligned}$$

Then the sequence

$$s_{k_0}, i_{k_1}, s_{k_1}, o_{k_1}, i_{k_2}, s_{k_2}, o_{k_2}, \ldots$$

is an execution of $\mathcal{M}$.

Proof sketch. For the first statement, take any $j \geq 1$. A quick induction shows that for any $k > j$, $id_j \in dom(f_k)$ and $f_k(id_j) = o_j$. Then it immediately follows from the definition of δ' and *nop*.

For the second statement, let $j \geq 0$. Observe that for all $k \in \{k_j + 1, \ldots, k_{j+1} - 1\}$, $(s_k, f_k) = (s_{k_j}, f_{k_j})$, hence

$$((p_{k_{j+1}}, f_{k_{j+1}}), o_{k_{j+1}}) = \delta'(s_{k_{j+1}-1}, (id_{k_{j+1}}, i_{k_{j+1}})) = \delta'(s_{k_j}, (id_{k_{j+1}}, i_{k_{j+1}})).$$

Furthermore, $id_{k_{j+1}} \notin f_{k_j}$, so $(p_{k_{j+1}}, o_{k_{j+1}}) = \delta(s_{k_j}, i_{k_{j+1}})$ by definition of δ', but $s_{k_{j+1}} = s_{j_k} \sqcup p_{j_{k+1}}$, which gives the result. $\square$

The formal definition of $\mathcal{M}'$ says that the set of inputs assigns to each identifier a unique input from the original machine $\mathcal{M}$. An implementation wouldn't of course know this assignment a priori, but construct it on-the-fly. When a client wants to send a request with input $i \in I$, the client would first choose an identifier $id \in ReqId$ which hasn't been chosen before (by that or any other client), assign i to id, and send (id, i) as the requested command.

An easy way to choose such an identifier is to make sure that the set *ReqId* is large enough and pick the id randomly with uniform distribution from the set. For example, suppose that *ReqId* is the set of 64-bit integers. Obviously this doesn't completely guarantee uniqueness of the identifiers, so the implementation wouldn't *formally* comform to the spec, but *in practice* it would since the probability of collision is miniscule (assuming that the distribution is truly uniform, or close to it).

The following practical observation can make an implementation more efficient. Suppose that during an execution of the algorithm for the idempotent machine $\mathcal{M}'$ (constructed from some $\mathcal{M}$), a proposer obtains $(s, f) \in P \times (ReqId \dashrightarrow O)$ by merging

reads from a quorum of replicas and observes that $req.id \in dom(f)$, where req is the request currently being handled. Then the proposer *can immediately return* $f(req.id)$ *to the client*, without making any proposal. Indeed: the fact that $req.id \in dom(f)$ proves that a decree D with $D.id = req.id$ and $D.output = f(req.id)$ was chosen in an earlier slot. Furthermore, there is no point in calculating $\delta'((s, f), req.input)$ and sending a proposal — the proposer knows that the obtained patch will have no effect on on the state and the obtained output will be equal to $D.output$.

Furthermore, as explained in the previous section, the proposer doesn't need the replicas to send all the remembered outputs f; in this case, the proposer only needs $f|_{\{id\}}$.

Storing previous outputs in the state introduces a problem: the state grows with each new request. A real implementation probably can't afford that.

A practical way to solve this problem is to *expire* the *id*s and outputs from the stored state after some time. This, of course, is not formally allowed by the specification. However, recall the motivation for introducing this mechanism in the first place: we wanted to allow clients to retry requests and correctly handle duplicated messages. Now, given a request, if a client eventually stops retrying that request (either because it finally succeeded or the client gave up), and if eventually the network stops duplicating the messages that contain that request, then it is safe to forget that it was handled; the system wouldn't handle that particular request "correctly" if it appeared again, but if it won't, it doesn't matter. The pragmatic approach to expiring remembered *id*s would be simply to wait for a "long enough" period of time.

There are other methods to solve the idempotency problem, perhaps better. For example, the Raft thesis [17] suggests introducing the concept of a *client session*, where we don't treat and remember each request independently, but keep requests in a set associated to each client which has recently contacted the system, using unique client identifiers. The client then takes care to maintain a *serial number* that grows with each new request for a given client ID. As before, we need to expire sessions after some time.

4.3. LattiStore

In order to test his ideas and convince himself that the algorithm can be used in practice, the author has built *LattiStore*, a distributed system that provides the functionality of a simple key-value store and tolerates the failures of up to less than a half of all the nodes. The implementation is available at GitHub[1].

The system provides a transaction API. Transactions are simple imperative programs that can read and modify values stored under multiple keys; the keys and values are strings. An example transaction is:

```
x = get "x";
y = get "y";
if x == "1" {
    put "x" "2";
    put "y" "3";
} else {
```

[1] `https://github.com/LattiStore/lattistore`

```
        put "x" x + y + "1";
    }
```

The + operator is string concatenation. The response to each transaction contains the value for each key that was accessed through a `get` operation and that was set using a `put` operation in one of the previous transactions. When a transaction accesses a value under a key that does not have a value yet, it interprets it as an empty string.

The above transaction executed three times in a row will return the following results in order:

```
{}
{"x": "1"}
{"x": "2", "y": "3"}
```

The implementation provides a simple client application that can be used to connect the system and issue transactions.

The transactions are fully isolated — to the clients, each issued transaction appears to execute indivisibly somewhere between the request and response. Formally, the system provides strict serializability [7]. Or, if we view the entire system as a state machine with the transactions as inputs, the system appears to provide linearizable semantics for that state machine.

Underneath, LattiStore uses LPaxos to replicate the patch machine described in example 2.2.3, augumented with idempotency using the technique from section 4.2. It also uses the optimization from section 4.1.

The implementation is written in the Rust programming language[2] using the Tokio framework[3] for writing asynchronous applications. A LattiStore system is a *cluster* of *nodes*. Each node executes 4 main tasks (*tasks* are Tokio's lightweight threads) corresponding to the different roles that a node performs: a proposer, an acceptor, a replica, and a failure detector. The proposer task has access to interfaces which allow it to contact the acceptors and replicas (both on the local and remote nodes). Communication between nodes is implemented with Tonic [4] — a Rust implementation of gRPC[5]. The proposer can also query its local failure detector. Due to lack of time, instead of implementing a "real" failure detector, the author left a placeholder implementation that rotates between the nodes in a round-robin fashion, choosing a leader based on the current time.

The proposer's operation is very similar to the outline described in section 3.3. It runs in a loop. Each iteration of the loop starts by querying the local failure detector oracle, asking it to return a node that is the suggested leader. If the returned node ID is the local node, the proposer starts a recovery procedure. It first updates the highest known balnum (which it remembers between the loop iterations) by querying its local acceptor (the acceptor task running on the local node) — the acceptor might have been contacted by a remote proposer and observed a higher balnum.

[2] `https://www.rust-lang.org/`
[3] `https://github.com/tokio-rs/tokio`
[4] `https://github.com/hyperium/tonic`
[5] `https://grpc.io/`

It then calculates an even higher balnum *num* that it owns and broadcasts a message to the acceptors, asking them to perform the $Promise(acc, num)$ step and send the promises back to the proposer. An acceptor might reject the request if it has seen a higher balnum (from another proposer), in which case it informs the proposer about the higher balnum; if the proposer receives such a rejection message, it updates the highest known balnum and continues to the next loop iteration. It also continues to the next iteration if no quorum of acceptors answers within a given period of time (currently hardcoded as 5 seconds).

After receiving a quorum of promises, the proposer checks if the greatest returned proposal was returned by each member of the quorum (i.e. everyone returned the same proposal). If so, it proceeds to normal operation. Otherwise, it first attempts to get the decree of this proposal chosen using the ballot *num* by broadcasting a new proposal to the acceptors (this corresponds to the $Repropose(num)$ action) and waiting for them to accept (perform the *Accept* action) and send a confirmation. As in the case of promises, the proposer may receive a rejection as a response to the accept request informing it about a higher balnum, or it may timeout, in which case it proceeds to the next loop iteration.

Normal operation is a nested loop. In each iteration the proposer remembers the proposal chosen in the previous iteration (if this is the first iteration of the normal operation loop, this is the proposal obtained from the recovery procedure; either the greatest proposal returned by a quorum of acceptors, or the one chosen by *Repropose*) and waits for a client request. Let *slot* denote the next free slot (i.e. $slot - 1$ is the slot of the proposal chosen in the previous iteration). When the client request arrives, the proposer sends a message to replicas containing the patch of the previously chosen proposal, $slot - 1$, and the set of keys that it wants to read (based on the client request). The replicas apply the proposal's patch (which corresponds to the $Apply(rep)$ action) and return the values and versions that they currently store under the requested keys ($Read(rep)$ action, optimized using the technique from section 4.1). If a replica sees that the slot $slot - 1$ sent by the proposer is smaller than its last known slot, it informs the proposer, in which case the proposer will break the normal operation loop and go back to the outer loop (querying the failure detector). If the proposer obtains the values and versions under the requested set of keys from a quorum of replicas, it merges them together (as described in example 2.2.3) and applies the transaction contained in the client request to obtain a patch. It then proposes the patch using its current ballot and slot *slot* by broadcasting a message to the acceptors (which corresponds to $ProposeNew(slot, num)$). If this succeeds — a quorum of acceptors perform *Accept* and return a confirmation — it continues to the next iteration of the inner loop; otherwise, it breaks and goes back to the outer loop.

To use the system, one has to first setup a cluster. Each node must be started and provided the IP addresses of other nodes. Each pair of nodes must be able to communicate for initialization, so the nodes can generate and exchange unique node identifiers. The identifiers are then used by each node to identify other nodes and are used to construct ballot numbers.

After the cluster initializes, the client can start sending requests to the nodes. The node will agree to handle the request if the failure detector tells it to be the current "leader"; otherwise, it will redirect the client to the node chosen by the failure

detector. To handle a request the node must be able to reach a majority of nodes (the node itself can be a part of the majority). For example, if the cluster was initialized with 5 nodes, then the node must contact at least 2 other nodes to handle a request, since together they form a majority of 3 nodes.

LattiStore serves only as a proof of concept and is not fit for production use; for one thing, it does not use persistent storage. This means that once a node shuts down, it cannot restart and keep being a part of the cluster, since it lost its acceptor and replica states. The shutting down of a node represents an irrecoverable crash in the model of failures assumed by the algorithm. In particular, once a majority of nodes shut down, the cluster becomes permanently unavailable (no longer able to process any more client requests).

The system also assumes static node membership; there is no way to add or remove nodes (note that a node crashing is not equivalent to removing the node — it is still taken into account when counting quorums, for example). Changing membership would require augumenting the algorithm with a reconfiguration procedure which could change the set of replicas, acceptors, and proposers, without breaking the safety properties of the algorithm. We discuss this in section 4.5.

LattiStore was tested using the *Jepsen* testing framework [9]. Jepsen was successfully used to find serious bugs in multiple databases; as the website says[6]:

> Since 2013, Jepsen has analyzed over two dozen databases, coordination services, and queues—and we've found replica divergence, data loss, stale reads, read skew, lock conflicts, and much more.

Jepsen takes a black-box approach to testing distributed systems: it simulates a client (or set of clients) executing queries against the system and constructs a history of requests and responses. It then verifies whether the history satisfies the consistency properties (such as linearizability) claimed in the documentation or marketing materials of the system under test. While the queries are being executed, the framework simulates failures occuring in real systems, such as network partitions.

The author of this thesis has adapted the Jepsen etcd tests described in a recent analysis of the etcd key-value store[7]. The code repository is available at GitHub[8]. The tests were executed against a cluster of 5 nodes running in LXC containers.

There are two types of tests:

- a *register test* performs reads, writes, and CAS operations over single keys that store integers. The histories observed by the framework are then checked against linearizability using the Knossos[9] linearizability checker.
- An *append test* stores lists of unique integers under multiple keys. The test executes transactions that append new integers to a subset of these lists. The framework then uses Elle [10][10], a transactional consistency checker, to construct a dependency graph between the executed transactions to find isolation anomalies [1].

[6] Taken from `https://jepsen.io/analyses`.
[7] https://jepsen.io/analyses/etcd-3.4.3
[8] `https://github.com/LattiStore/jepsen`
[9] `https://github.com/jepsen-io/knossos`
[10] `https://github.com/jepsen-io/elle`

The framework injected network partitions and process pauses while the tests were running; it did not perform node restarts as the system does not support persistent storage. Over the course of multiple executions, the tests did not observe violations of strict serializability in the latest version of LattiStore.

4.4. Data removal

LattiStore's `put` operation updates the value under the given key. There is no way to delete a value. The lack of such feature is an obstacle to making LattiStore a system appropriate for general use.

However, to completely remove the data corresponding to a given key, it would be necessary to contact every replica, since every replica may potentially store the value. If we only removed the data from, say, a quorum, then a later read operation may still observe the value if it contacts a different quorum, resulting in "data resurrection". On the other hand, it should be possible to perform the delete operation even though only some quorum of the nodes is available as promised by the system's fault tolerance properties.

The deletion operation can be made a special case of the update operation by introducing a special *tombstone* value. If a given key k holds a tombstone it means that the value under k was deleted. The usual merging operation performed by proposers when reading from replicas would return a tombstone for a given key if its version is higher than than the versions of non-tombstone values possibly stored on other replicas.

Unfortunately tombstones continue taking space. A practical system should be able to eventually get rid of them.

A possible solution to this problem would be to introduce a garbage collection mechanism whose purpose would be to remove a tombstone when it's safe to do so.

Recall that in example 2.2.3, $P = (K \dashrightarrow V \times \mathbb{N}) \times \mathbb{N}$. For $rep \in Replica$, let $s_{rep} = store[rep][1]$, i.e. $s_{rep} \in K \dashrightarrow V \times \mathbb{N}$ is the set of tuples (k, v, n) stored by rep (recall how we identify partial functions with sets of tuples).

Suppose that $(k, T, n) \in s_{rep}$ for some $rep \in Replica$, where T denotes the special tombstone value. Then it is safe to remove (k, T, n) from s_{rep} if for every $rep' \in Replica$, if $(k, v, n') \in s_{rep'}$ for some v, n', then $n' \geq n$. In other words, every replica must either not have a tuple with key k ($k \notin dom(s_{rep'})$), or the tuple for key k that it stores has a version greater or equal to n ($s_{rep'}[k][2] \geq s_{rep}[k][2] = n$).

An informal argument follows. Suppose we perform a read from a quorum of replicas and merge the results to obtain a state $s : K \dashrightarrow V \times \mathbb{N}$. Note that $s[k][2] \geq n$ by the property above. There are two cases:

- $s[k][2] = n$: then $s[k][1] = T$ since $s_{rep}[k][1] = T$ (the version determines the value). If we had removed (k, T, n) from a subset of the replicas prior the read, then each read result used to obtain s would either be the same or would differ by missing (k, T, n). Thus, s would either be the same or would be equal to $s \setminus (k, T, n)$. But $s[k][1] = T$ is treated as if $k \notin dom(s)$, so in this case the end result does not change.

- $s[k][2] > n$: then some read had a tuple (k, v, n') with $n' > n$. Removing (k, T, n) from a subset of the replicas prior to the read wouldn't affect these higher-versioned tuples, so it wouldn't change the result.

With this observation we can implement a garbage collection mechanism as follows. Each replica *rep* has an associated garbage collection process, which we'll call $gc(rep)$. This process stores a single value, $gcsafe[rep] \in \mathbb{N}$, with the following property:

$$gcsafe[rep] \geq n \Rightarrow \text{ it is safe to remove all tuples } (k, T, n) \text{ from } rep.$$

The previous observation says that, for $rep \in Replica$,

$$(\forall k : s_{rep}[k] = (T, n) \Rightarrow \forall rep' \in Replica : k \notin dom(s_{rep'}) \vee s_{rep'}[k][2] \geq n) \\ \Rightarrow \text{ it is safe to remove } (k, T, n) \text{ from } s_{rep}.$$

Thus it is sufficient for $gcsafe[rep]$ to satisfy the following property:

$$gcsafe[rep] \geq n \Rightarrow \\ \forall k : s_{rep}[k] = (T, n) \Rightarrow \forall rep' \in Replica : k \notin dom(s_{rep'}) \vee s_{rep'}[k][2] \geq n.$$

The $gc(rep)$ process runs in the following loop:

1. choose n, $gcsafe[rep] \leq n \leq last[rep]$.
2. Ensure that, for each k such that $s_{rep}[k] = (T, n)$,

$$\forall rep' \in Replica : k \notin dom(s_{rep'}) \vee s_{rep'}[k][2] \geq n.$$

3. Update $gcsafe[rep]$ to n.
4. Clear all tombstones with versions $\leq gcsafe[rep]$.

In step 1 we couldn't have chosen $n > last[rep]$ because then we wouldn't be able to perform step 2 without blocking all replicas from applying patches with tuple versions in the range $[last[rep], n]$, which is definitely not something we want to do (the system could be halted from making any progress).

But even $n = last[rep]$ is dangerous since it might be true that no quorum of replicas has yet applied $p_{last[rep]}$ (which contains tuples with the $last[rep]$ version). The safe choice is $n = last[rep] - 1$, since the patch $p_{last[rep]}$ has already been decided, hence $p_{last[rep]-1}$ and earlier patches must have already been applied by some quora, which means that the algorithm can make progress even though replicas no longer agree to apply patches $p_{last[rep]-1}$ and earlier.

Step 2 above can be performed as follows. For each $rep' \in Replica$:

1. tell rep' to not apply patches with versions $\leq n$. By the definition of the patch machine in example 2.2.3, these are the patches $\{p_1, \ldots, p_n\}$ (patch p_i contains tuples with versions i).

   After receiving a confirmation, proceed to the next step.

2. For each $(k, T, n') \in s_{rep}$ such that $n' \leq n$, send (k, T, n') to the process $gc(rep')$.

   $gc(rep')$, upon receiving (k, T, n') from $gc(rep)$, compares n' to $gcsafe[rep']$. If $n' \leq gcsafe[rep']$, it means that rep' already ensured that tuples with versions n' are safe to garbage-collect and can simply drop this tuple. Otherwise, it applies (k, T, n') to $store[rep']$. In any case, it then sends a confirmation back to $gc(rep)$.

   $gc(rep)$ considers the step successful after receiving a confirmation (for all $(k, T, n') \in s_{rep}$).

Note that this might take a long time as some of the other replicas may currently be unavailable. It might happen that only some quorum is currently responsive and the cluster is handling new client requests, but the *gc* process is stuck. This is not a big problem: we want to collect the tombstones eventually, not necessarily right now. If the other replica eventually becomes available, the process will eventually succeed.

However, there is a problem with replicas that have crashed irrecoverably; the other replicas will never be able to contact them, so their *gc* processes will be forever stuck, not able to collect tombstones anymore. By the asynchrony of the assumed model, there is no way for one replica to discern whether another replica is (permanently) crashed or just very slow (perhaps temporarily crashed).

But there is a backdoor... it's called the *system administrator*. The replica could allow an input from the administrator so they can promise that the other replica is truly crashed irrecoverably and will never come back to life. In that case the *gc* process could simply skip that replica. Unfortunately, this assumes that the admin does not lie; if they make a mistake and the other replica actually comes back to life, we could have erroneously removed a tombstone that wasn't applied to the other replica and experience the phenomenon of data resurrection. A wise man (private contact) once said that anything involving a human is a byzantine failure.

Another way would be to perform a membership change that removes the supposedly crashed replica from the set of replicas (in effect ensuring that it really won't come back). The next section is dedicated to this topic.

A potential research problem is to generalize the above discussion to all patch machines. Is there a general construction that can be performed on a patch machine which introduces some notion of a tombstone that can later be garbage collected?

4.5. Membership changes

Consider a system with A acceptors, R replicas, and P proposers.

In order to get a proposal chosen, a quorum of acceptors is required. In order to apply a patch or perform a read, a quorum of replicas is required. And at least one proposer must be running in order to drive the algorithm.

Thus the system tolerates failures of up to $\left\lfloor \frac{A-1}{2} \right\rfloor$ acceptors, $\left\lfloor \frac{R-1}{2} \right\rfloor$ replicas, and $P - 1$ proposers.

A typical deployment would have N nodes running on different servers, where each node contains a single acceptor, replica, and proposer. This is how *LattiStore* is

designed, for example. In that case the system tolerates failures of up to $\left\lfloor \frac{N-1}{2} \right\rfloor$ servers.

Using persistent storage may delay the irrecoverable crash in time, as it changes some crashes that would otherwise be permanent into recoverable crashes (like a server restarting). But the system should be prepared for eventualities like, say, a server getting burned down.

After an acceptor suffers a permanent failure, for example, the system still formally has A acceptors and still requires a majority of them to choose a proposal; one of them is simply not responding anymore. A practical system needs a way to replace the acceptor, or more generally, be able to add or remove an acceptor. The same holds for replicas and proposers.

An *acceptor configuration* is a set of acceptors. A *replica configuration* is a set of replicas. The replication algorithm from chapter 3 assumes *static* acceptor and replica configurations that are used to decide patches and store the state of the patch machine. The *reconfiguration problem*, or the *membership change* problem, is the problem of changing configurations (of acceptors or replicas) without breaking consistency properties (such as linearizability of histories observed by clients of the system) while maintaining system availability.

A complete and formal treatment of the reconfiguration problem would require making the specification and the proof much more complex. Due to the limited time constraints, following the steps of other authors (see e.g. the Raft thesis [17]), the author of this thesis decided to present an informal discussion on augumenting the LPaxos algorithm with the ability of reconfigurations.

4.5.1. Replica reconfiguration

Let $\mathcal{M}$ be the replicated patch machine. We will assume that each state s of $\mathcal{M}$ stores a value, which we'll denote $\mathit{NextRepConf}(s)$, not used by $\mathcal{M}$ for anything (it will be only used by the replication algorithm). That is, given an execution

$$s_0, i_1, s_1, o_1, i_2, s_2, o_2, \ldots$$

each s_i has a $\mathit{NextRepConf}(s_i)$ value not modified or accessed in any way by the patch machine commands.

Any patch machine can be easily modified to contain an additional value. For example, if P is the set of patches for some patch machine and V is some set of values, define $P' = P \times (V \times \mathbb{N})$. A patch $p' \in P'$ is then a tuple $(p, (v, t))$, where $v \in V$ and t is a natural number. The merge operation $\sqcup'$ of the new machine can be defined using the equation

$$(p_1, (v_1, t_1)) \sqcup (p_2, (v_2, t_2)) = (p_1 \sqcup p_2, \begin{cases} (v_1, t_1) \text{ if } t_1 > t_2, \\ (v_2, t_2) \text{ otherwise} \end{cases}).$$

The new δ' function can be defined by

$$\delta'((s, (v, t)), i) = ((p, (v, t+1)), o), \text{ where } (p, o) = \delta(s, i).$$

As before we will be deciding decrees in consecutive slots. For each $n \in \{1, \ldots, \mathit{MaxS}+1\}$, where MaxS is the maximum slot in which a decision has been made, define

$$\mathit{RepConf}(n) \triangleq \mathit{NextRepConf}(s_{n-1})$$

(recall that s_n, for $0 \leq n \leq MaxS$, is the state of the replicated machine after applying patches $p_0, \ldots, p_n$). Note that the configuration of slot $MaxS+1$ is defined even though no decree has been chosen in $MaxS+1$ yet; the configuration is given by the state $s_{MaxS} = p_0 \sqcup \cdots \sqcup p_{MaxS}$.

The informal meaning of $RepConf(n)$ is: it is the configuration of replicas to which we apply patch p_n and from which we read state s_n (after applying p_n).

We assume that $RepConf(s_0) = RepConf(1)$ is given as an algorithm constant. We will also define $RepConf(0) = RepConf(1)$. It is the initial replica configuration with which the system starts; it is some finite set of replicas that store the initial state $s_0 = p_0$, same as in the static-configuration version of the algorithm.

For each state s of an execution of $\mathcal{M}$, $NextRepConf(s)$ will be either of the two following types:

- a set of replicas, in which case $NextRepConf(s)$ will be called a *normal configuration*, and the replicas will be referred to as the *current replicas* (of state s),
- or a pair of sets of replicas, in which case $NextRepConf(s)$ will be called a *joint configuration*. For a joint configuration $NextRepConf(s) = (R_1, R_2)$, the elements of R_1 will be called the *current replicas* and the elements of R_2 will be called the *new replicas* (of s).

Additionally to the state, we will keep the configuration inside decrees. That is, for each decree dec, there is a $RepConf(dec)$ value stored inside the decree which is either a normal or a joint configuration as in the case of states. We will be maintaining the following invariant:

$$\text{If } p \in proposals, \text{ then } RepConf(p.dec) = RepConf(p.slot).$$

In particular, if a decree D_n is chosen in slot n, then $RepConf(D_n) = RepConf(n)$.

As usual, to enter normal operation, a proposer first needs to perform recovery. That is, it needs to obtain promises from a quorum of acceptors and if necessary, perform the *Repropose* action to choose a decree from a previous proposer in the new picked ballot.

Suppose that during the recovery procedure the proposer has obtained a decree $p.dec$, where p is the greatest previously accepted proposal sent in the acceptors' promises; the proposer then ensured that $p.dec$ is chosen in slot $p.slot$ by performing *Repropose* and waiting for a quorum of acceptors to confirm. Let $n = p.slot$. The proposer knows $RepConf(n)$, since by the above invariant, $RepConf(n) = RepConf(p.dec)$. This is the configuration onto which the proposer applies the patch $p_n = p.dec.p$ (below we define what it means to apply a patch to a joint configuration).

Now, in normal operation, the proposer can read s_n from the replicas (below we explain how a read is performed using joint configurations). This is how the proposer obtains $RepConf(n+1) = NextRepConf(s_n)$, which will be used to construct the decree that it will propose for slot $n+1$. That is, the new decree dec will satisfy $RepConf(dec) = RepConf(n{+}1)$. After choosing the decree in slot $n{+}1$ the proposer will apply the patch $dec.p$ to the configuration $RepConf(n{+}1)$, and continue normal operation with slot $n+1$.

If during the recovery procedure a quorum of acceptors sent promises with no previous proposal (i.e. $m.mprop = \bot$), then the proposer starts normal operation with $n = 0$, i.e. $n + 1 = 1$ is the first slot in which it will propose a decree. The proposer knows the initial configuration of replicas $RepConf(0)$ (it is a constant of the algorithm) which it contacts to obtain the initial state s_0.

We will now define how patches are applied to and how states are read from joint configurations.

For a finite set of replicas R, let $Quorums(R)$ be the set of all quorums of R, i.e. all subsets of R that contain more than half of the replicas from R. For example,

$$Quorums(\{A, B, C\}) = \{\{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}.$$

The *current quorums* of a normal configuration R are the quorums of R:

$$CurrQuorums(R) = Quorums(R).$$

The *current quorums* of a joint configuration (R_1, R_2) are the quorums of current replicas R_1:

$$CurrQuorums((R_1, R_2)) = Quorums(R_1).$$

The *new quorums* are the quorums of new replicas:

$$NewQuorums((R_1, R_2)) = Quorums(R_2).$$

A *joint quorum* of a joint configuration (R_1, R_2) is a subset of $R_1 \cup R_2$ that contains some R_1 quorum and some R_2 quorum. Formally:

$$JointQuorums((R_1, R_2)) = \\ \{R \subseteq R_1 \cup R_2 : R \cap R_1 \in Quorums(R_1) \wedge R \cap R_2 \in Quorums(R_2)\}.$$

For example,

$$JointQuorums((\{A, B, C\}, \{B, C, D\})) = \\ \{\{A, B, C\}, \{A, B, D\}, \{A, C, D\}, \{B, C, D\}, \{A, B, C, D\}\}.$$

To apply a patch to a joint configuration, it must be applied to some joint quorum of the configuration. Equivalently, it must be applied to some quorum of current replicas and some quorum of new replicas. Formally, every $rep \in Q$ for some $Q \in JointQuorums((R_1, R_2))$ must update $store[rep]$ to $store[rep] \sqcup p$, where p is the patch. It will become clear soon why we do this.

To read state from a joint configuration (R_1, R_2), we simply read it from a current quorum, i.e. a quorum in R_1.

Thus, when a proposer chooses a decree dec in slot $n + 1$, if $RepConf(n + 1)$ is a joint configuration, the proposer must ensure that the patch $dec.p$ is applied to a joint quorum of $RepConf(n + 1)$ before proceeding. To read s_{n+1} (after applying p_{n+1}), the proposer must contact a current quorum of $RepConf(n + 1)$ and merge the obtained patches. As usual it can optimize by using a single message to apply a patch and perform a read (for current replicas; it doesn't have to read from new replicas).

Applying to and reading from normal configurations is the same as before: we apply to and read from a current quorum.

To switch a configuration, a proposer simply proposes a patch which modifies the state to store a different configuration. For example, if n is the next free slot, the proposer will send a proposal p such that

$$NextRepConf(s_{n-1} \sqcup p.dec.p) = C,$$

where C is the desired new configuration. If the proposal becomes chosen, i.e. p_n is decided to be $p.dec.p$, then

$$RepConf(n+1) = NextRepConf(s_n) = NextRepConf(s_{n-1} \sqcup p_n) = C.$$

However it is not safe to arbitrarily switch configurations. For example, let $R_1 = \{A, B, C\}$ and $R_2 = \{D, E, F\}$. Suppose that $RepConf(n) = R_1$. A patch p_n is chosen in slot n such that $NextRepConf(s_{n-1} \sqcup p_n) = NextRepConf(s_n) = R_2$. Thus, starting from slot $n+1$, R_2 is used, i.e. $RepConf(n+1) = R_2$. The next chosen patch p_{n+1} is then applied to a quorum in R_2; so far so good. But then a read is performed from a quorum in R_2, and the result is $p_0 \sqcup p_{n+1}$ (assuming that the R_2 replicas initially stored the p_0 patch) — all the data from patches p_1 to p_n is missing!

Before we switch to a new normal configuration R_2, we must ensure that reading from any quorum of R_2 and merging the results gives us the correct state (which is the result of applying all previous patches). A simple method would be as follows. Suppose a proposer is in normal operation and the next free slot is n.

1. Stop handling all client operations.
2. Perform a read from R_1 to obtain s_{n-1}.
3. Apply s_{n-1} to R_2.
4. Calculate a patch p that switches the configuration to R_2.
5. Apply p to R_2.
6. Try to choose p in the nth slot.
7. If successful, $p = p_n$; apply p_n to R_1.
8. Resume client operation.

If every step up to step 7 succeeds, the next patch p_{n+1} will be applied to R_2. The previous steps ensured that R_2 contains s_n. Thus, after applying p_{n+1}, R_2 will contain s_{n+1}, and the system will continue working from there using replicas R_2 with consistency preserved.

If one of the steps from 1 to 5 fails, the procedure can simply be retried.

A more complex case is if a failure occurs at step 6 — for example, the proposer who initially tried to get p decided in slot n crashed in the middle of the operation. Its proposal might have been accepted by some acceptors, and a different proposer in a higher ballot may repropose this patch and get it chosen. Thus, whoever initiated the reconfiguration procedure (a system administrator, or perhaps some auto-scaling cluster orchestration mechanism) may not know if it succeeded or not, in case of

failure of the initial proposer which attempted the switch. This information may be important e.g. in order to decide whether or not the system runs on replicas R_2 so it's safe to turn off replicas $R_1 \setminus R_2$.

A naive approach would be to perform a read in order to check if the state stores R_2 or R_1. However, if we've never got a response to the reconfiguration request, the read is formally happenning concurrently with the reconfiguration, so it may be ordered before the reconfiguration even starts. Consider the following sequence of events:

- the system administrator sends a reconfiguration request from R_1 to R_2.
- There is no response for a long period of time, so the impatient administrator sends a read request.
- The read request returns R_1.
- But then the original reconfiguration request actually reaches the cluster for the first time, and the reconfiguration happens.

This may leave the administrator thinking that the reconfiguration failed and won't happen. The scenario may seem improbable, but the asynchronous distributed system model allows these kinds of failures.

We need a viable way of telling whether the given reconfiguration request happened or not. We can use a similar trick as the one from section 4.2. With each reconfiguration request, a unique ID is associated. The patch that switches the configuration from R_1 to R_2 saves this ID in the machine's state. If the agent who originally sent the request wants to know if it was successful (because they didn't receive any response), they send another request which contains this unique ID and does the following:

- if the state already contains the ID, it means that the reconfiguration was performed.
- Otherwise, the new request saves the ID in the state which will block the reconfiguration request if it is later attempted.

Thus we need to modify the reconfiguration procedure above slightly: after step 2 (but before step 4), the proposer checks if the obtained state contains the ID. If it does, it will not propose the reconfiguration patch.

This works since the algorithm enforces one of the requests to be ordered before the other. If the reconfiguration request gets ordered before the check request, the check request will acknowledge that the reconfiguration was performed. Otherwise the check request will get ordered before the reconfiguration request so it will confirm that reconfiguration was not (and will not) be performed, and the admin should try again if they want the reconfiguration to happen (but with a different ID).

Observe that the check request is idempotent (for a given ID) so the administrator can perform it multiple times until they learn what they wanted to know.

But there's another problem with this solution: it's not viable for large states. Steps 2 and 3 may take a long time if the state is large since the entire state of a replica quorum must be obtained and then merged, and the service cannot serve any client requests until the procedure is finished or interrupted.

This is where joint configurations help. First, observe the following:

- it is always safe to switch configurations from R_1 to (R_1, R_2).
- It is always safe to switch from (R_1, R_2) to R_1.

"Safe" means that the system won't start giving inconsistent results after performing the switch.

Indeed: after we switch from R_1 to (R_1, R_2), all reads are still performed using R_1 and patches are still applied to R_1. The difference is only that additional work is being done: a patch must be also applied by a quorum in R_2.

Switching from (R_1, R_2) back to R_1 causes all the work performed on R_2 to be effectively lost, but as above it doesn't break consistency since all patches were still being applied to R_1.

The problem we will now solve is the following:

$$\text{ensure that it is safe to switch from } (R_1, R_2) \text{ to } R_2.$$

As we discussed, it is necessary to ensure that reading from R_2 gives the correct state. The general idea is essentially the same as in the previous solution:

1. read from R_1 to obtain a patch s.
2. Apply s to R_2.

This time however, the two steps can be performed in parallel while new client requests are being handled and new patches are being applied to the joint configuration (R_1, R_2). The intuitive argument is as follows: suppose that $\mathit{RepConf}(n) = (R_1, R_2)$. This means that patches starting from n are applied to R_2. By performing the above procedure, we ensure that all patches previously applied to R_1 (p_1 to p_{n-1}) also get applied to R_2. Thus the end result is that R_2 has each previous patch applied.

The details of how to do it efficiently and the degree of possible parallelism depend on the specific patch machine. For a general patch machine one could proceed as follows. A proposer first tells the replicas from R_1 to save a snapshot of their state in-between applying two patches. Thus, additionally to $\mathit{store}[\mathit{rep}]$ that is used to apply new patches and perform standard reads, each contacted *rep* stores a separate copy of the state that is equal to what $\mathit{store}[\mathit{rep}]$ was at some point in the past. The proposer continues handling client requests as normal, but it spawns a background process. The process performs a read from a quorum of replicas which performed the snapshot, asking them to return it. It merges the obtained snapshots and sends the result to R_2 for application. After finishing, the process informs its proposer, and the proposer now knows that it's safe to switch the configuration to R_2, as R_2 has effectively applied all previous patches.

Using snapshots is one way, but concrete patch machines may allow more efficient methods. For example, consider the key-value store (2.2.3). The read on each replica *rep* can be performed in parallel as new patches are applied using the same $\mathit{store}[\mathit{rep}]$; the result of the read has to be isolated only at the level of keys, i.e. each obtained tuple must come from some single state s_k, but different tuples may come from different times. The result of the read will be a patch $s = s_n \sqcup p$, where n is some slot from before the read started, and p contains tuples from p_{n+1}, p_{n+2}, and so on, but not necessarily all tuples from p_{n+1}, or p_{n+2}, etc. But that is fine as long

as patches $p_{n+1}, p_{n+2}, \ldots$ are also applied to R_2, which we guaranteed by switching the configuration to (R_1, R_2) prior to the read (so $RepConf(n+1) = (R_1, R_2)$). Applying s onto R_2 gives the desired end result of all patches from p_0 onwards to be applied to R_2 (and the application of s to R_2 can also happen in parallel with application of other patches — again, isolation needs only be ensured on the level of single keys).

The procedure of reconfiguring replicas can thus be performed as follows. During normal operation, a proposer:

1. chooses a patch to switch a normal configuration R_1 to a joint configuration (R_1, R_2).

2. Spawns a background task which reads from R_1, merges the obtained patches, and applies the result to R_2.

3. After the background task finishes, it informs the proposer.

4. The proposer then proposes a patch that switches from (R_1, R_2) to R_1.

Suppose the proposer fails during the procedure or gets preempted by another proposer with a higher ballot. A higher-ballot proposer will then see that the current configuration is a joint configuration (R_1, R_2) so it can finish the procedure by spawning its own background task. The implementation may allow saving the progress of the read-merge-apply procedure, in which case the new task can use the saved progress to resume from the middle instead of starting from scratch. For example, with the key-value store, if the set of keys is ordered and the copying process is performed in the order of keys, the process may periodically update the last copied key so if this process crashes, the next process can continue from that key.

A proposer can also abort the reconfiguration procedure as long as the system is using the joint configuration, since switching back to R_1 from (R_1, R_2) is safe. The abort may happen due an admin request. However, it may turn out that the abort request is handled by the proposer when the system is already in R_2, which the proposer will see by looking at the current state and/or the next chosen patch; in that case it can only inform the admin about being too late.

This reconfiguration procedure has a similar problem as the previous procedure. Suppose the administrator sends a reconfiguration request and the proposer never answers. The administrator then doesn't know if the reconfiguration has happened (or will eventually happen). The solution to this problem is the same as before. Before the initial patch that switches R_1 to (R_1, R_2) is proposed, a check is performed if the previous state (the one used to calculate the patch) contains the reconfiguration request's ID; if it does, the request does not get performed. Otherwise, the patch writes that ID to the state. To perform a check, the administrator sends a check request with that ID which either writes the ID to the state (if it wasn't already there) or returns a confirmation that the reconfiguration has started. Note that in this case a started reconfiguration may sometimes still be aborted, but remember that the aborting request (if performed) is racing with the reconfiguration procedure and may fail.

4.5.2. Acceptor reconfiguration

Similarily to replica configurations, with each slot n we assign a set of acceptors $AccConf(n)$ — the configuration used at slot n. It means that in order to choose a decree in slot n, a proposer must first obtain promises from a quorum in $AccConf(n)$ and send a proposal that gets accepted by a quorum in $AccConf(n)$. This time, fortunately, there won't be a need for joint configurations; however, changing acceptors comes with its own set of challenges.

$AccConf(n{+}1)$ is given by $NextAccConf(s_n)$ where s_n is the nth state in an execution of the patch machine. We can use the same method to store $NextAccConf(s_n)$ as we did with $NextRepConf(s_n)$.

Furthermore, each proposed decree dec stores a configuration $AccConf(dec)$ used by this decree. We maintain an invariant: for each proposal p,

$$AccConf(p.slot) = AccConf(p.dec).$$

Consider a proposer during normal operation. To change the configuration of acceptors, it proposes a patch p in the next free slot, say n, such that $NextAccConf(s_{n-1} \sqcup p)$ is the desired configuration. After the proposal is chosen the proposer knows that $AccConf(n) = NextAccConf(s_{n-1} \sqcup p) = NextAccConf(s_n)$ is the new set of acceptors.

Let $A_1 = AccConf(n)$ and $A_2 = AccConf(n+1)$. At this point, the proposer does not necessarily have promises from a quorum in A_2; it may, if the sets A_1 and A_2 intersect, but it's not guaranteed. In that case it can reuse the same ballot it was currently operating with, say num, to obtain the missing promises from members of A_2. Note that for each promise m returned from acceptors in $A_2 \setminus A_1$, $m.mprop = \bot$ — these acceptors couldn't have accepted a proposal before $m.num = num$ because such a proposal would need to had been created by a lower-ballot proposer, but we assumed that slot n was free so no proposal could have been chosen in slot n by such a lower-ballot proposer, which means that our num proposer is the first to know about the configuration change to A_2.

After ensuring that it has a quorum of promises from A_2, the proposer will attempt to choose a decree in slot $n{+}1$, the first slot that uses A_2. This will be a special decree whose entire purpose is to store metadata ($RepConf$ and $AccConf$); specifically, later higher-ballot proposers must be able to learn which replicas to use when recovering with the new set of acceptors A_2 and what is the next free slot. Thus, the proposer performing the reconfiguration procedure will send a proposal p with $p.slot = n+1$ and such that $RepConf(p.dec) = RepConf(n+1)$, but without a patch $p.dec.p$. New patches can be chosen starting from slot $n+2$.

After the metadata decree in slot $n+1$ is chosen (by being accepted by a quorum in A_2), the proposer considers the procedure finished and updates the last known acceptor configuration to be $AccConf(n+1)$.

A recovering proposer always starts with the last acceptor configuration known to have been used to choose a decree, or the initial configuration if it doesn't know about any chosen decrees. Let's say that the configuration is A_1. As always, recovery starts with the proposer obtaining promises from a quorum of these acceptors. However, it may turn out that the greatest previously accepted proposal $mprop$ obtained

from the promises has a different configuration: $A_2 = AccConf(mprop.dec) \neq A_1$. This means that a reconfiguration patch must have already been chosen using A_1 (otherwise no proposals could have been sent with the new configuration yet), so the proposer restarts recovery, attempting to obtain promises from a quorum in A_2 (but it can reuse the same ballot and obtain only the missing promises from $A_2 \setminus A_1$).

For example, consider the following scenario. $A_1 = \{A, B, C\}$, $A_2 = \{C, D, E\}$. A proposer first proposes a patch in slot 10 that changes the set of acceptors to A_2. The proposal gets accepted by A and B. Then it performs recovery using A_2 and proposes the special metadata decree in slot 11, which gets accepted by C. Now a higher-ballot proposer recovers with the old acceptors A_1 by obtaining promises from B and C. It sees that the greatest accepted proposal p has $p.slot = 11$ and $AccConf(p.dec) = A_2$, so it restarts recovery using A_2 with the same ballot, remembering that it already has one promise from C. D and E don't return any previously accepted proposal so the proposer continues to choose $p.dec$ in slot 11.

If, on the other hand, the greatest obtained proposal *mprop* has the same configuration, $AccConf(mprop.dec) = A_1$, then the proposer finishes recovery as usual by ensuring that $mprop.dec$ is chosen in $mprop.slot$. Now, it might turn out that $mprop.dec.p$ is a reconfiguring patch, i.e. $NextAccConf(s_n)$, where $n = mprop.slot$, is a different configuration than $AccConf(n)$. In this case, after ensuring that $mprop.dec$ is chosen, the proposer will continue the reconfiguration procedure as described previously: that is, it will perform obtain missing promises from the new set of acceptors $AccConf(n+1)$ and if no decree was proposed yet for slot $n+1$, it will propose the special metadata decree *dec* with $RepConf(dec) = RepConf(n+1)$.

Reconfiguring acceptors has one problem that does not occur for replica reconfigurations. A proposer can always learn the latest replica reconfiguration by performing recovery (contacting a quorum of acceptors). However, a proposer may not know the latest acceptor configuration — while it was asleep, another proposer might have performed a reconfiguration that changed the set of acceptors completely, and the previous set of acceptors might be unavailable at this point (e.g. because the administrator turned them off). Thus, when the proposer will try to contact its last known set of acceptors, it will not receive a response, so it will not be able to perform recovery, learn the new set of acceptors, and choose any decrees.

If we're not careful, this may result in total system unavailability. For example, consider a system with 3 proposers $\{X, Y, Z\}$. X is currently in normal operation while Y, Z are asleep. The system administrator sends a reconfiguration request to X that changes the set of acceptors from A_1 to A_2, where $A_2 \cap A_1 = \emptyset$. X performs the reconfiguration and returns a success response to the admin. The admin turns off the A_1 acceptors. Unfortunately, X decides to permanently crash at this moment. Now Y wakes up (perhaps its failure detector saw that X is unresponsive) and tries to contact A_1, but it fails; Z is also not able to do anything. Thus, even though we have 2 proposers still alive, the system is effectively dead.

We can come up with many solutions for this problem; the choice is up to the implementation. For example, before performing acceptor reconfiguration, a proposer could first announce its plan to the other proposers, and continue only after sufficiently many proposers acknowledge that they heard the announcement. These proposers will then remember that if they fail to contact their last known acceptor set, they can retry with the other set which they received from the reconfiguring

proposer. A proposer that successfully performs a reconfiguration should also inform others so they update their last known configurations and don't unnecessarily try to recover with previous configurations. Note that if a proposer does successfully manage to obtain a quorum of promises from an old configuration, it will learn about the new configuration through the greatest accepted proposal returned in the promises.

4.5.3. Changing proposers

While reconfiguring replicas or acceptors requires achieving consensus (by choosing an appropriate decree), changing proposers can be done in a conceptually much simpler way.

Instead of changing the set of proposers P, we assume that P is infinite from the start; it's just that all except some finite subset of proposers are operating at a time. We never remove a proposer from the system — we simply turn it off. Similarily, we never add a proposer to the system — we simply turn it on.

The only problem for achieving safety is to ensure that the set of ballot numbers is partitioned between the infinite set of proposers such that each proposer, for each balnum, can find a greater balnum that it owns. This can be done using the standard balnum construction that we've previously described. We require only that the set of proposers P is well ordered. Then the set of balnums can be defined as, for example, $\mathbb{N} \times P$, with the lexicographical ordering (we compare balnums first by the number, then by the proposer). Observe that by well-ordering of proposers and natural numbers, the obtained order is also a well-order.

There is also the availability issue described in the previous section. A new proposer must be able to somehow learn a recent acceptor configuration. However, it is not a safety issue if a proposer remembers some older acceptor configuration (for example, the initial one); it is only necessary to maintain liveness. Thus, the proposer must only learn about the recent acceptor configuration eventually, we don't need a complex protocol that maintains strong consistency. For example, we could use some kind of eventual consistency protocol (based e.g. on an epidemic algorithm [6]) executed between proposers to update the information about currently operating proposers and exchange acceptor configurations. And if, after all, we want to store some consistently replicated state in order to run a strongly consistent proposer membership protocol, we can use the replicated patch machine as we did for acceptors and proposers. The algorithm itself does not enforce any particular solution in this matter.

4.5.4. Putting it together

Consider a standard deployment consisting of a cluster of nodes, where each node runs a single proposer, acceptor, and replica. A full reconfiguration procedure that changes the set of nodes in the cluster can be composed of the reconfiguration procedures described in previous sections. For example, the procedure could be performed as follows:

1. the administrator turns on new nodes that they want to add. Each node has an acceptor and a replica with the initial state, and a proposer that does not do anything until it learns an initial acceptor configuration.

2. The administrator sends a reconfiguration request to one of the proposers. Let's call it the reconfiguration leader. The admin gives the leader the IPs of the new nodes and the identifiers of nodes currently present in the system that they want to remove.
3. The leader performs recovery if necessary, entering normal operation.
4. The leader contacts each new node, sending its proposer the last acceptor configuration known by the leader and adding them to the set of proposers that are present in the system. The new proposers are now members, ready to perform recovery and enter normal operation if necessary.
5. The leader performs replica reconfiguration as described in section 4.5.1, adding the replicas of new nodes and removing the replicas of nodes that are to be removed.
6. The leader performs acceptor reconfiguration as described in section 4.5.2, adding the acceptors of new nodes and removing the acceptors of nodes that are to be removed.
7. The leader removes the proposers of nodes that are to be removed.
8. The leader sends a message back to the administrator that the reconfiguration is finished.
9. The administrator shuts down the removed nodes.

To handle failure in the middle of the procedure, before each step, the proposer can store information in the replicated machine's state that its going to perform the next step. Thus, if another proposer recovers in a higher ballot (because e.g. the original proposer failed), it can continue the procedure by checking if the last stored step was successful and retrying it if necessary. For step 5 we've described how to perform the check in a detailed way in section 4.5.1; for step 6 we can use the same exact method.

The order of operations in the above procedure was picked to maximize fault tolerance. New proposers are first turned on, so they can choose new proposals and finish the procedure in case of other proposers failing. We then reconfigure the replicas; it is probably the longest step, and if it fails, we can still easily abort the whole procedure (by simply removing the new proposers) if we want to. After the switch to the new sets of replicas is performed, reverting would be quite expensive (we would have to go through the joint replica configuration phase to switch from new to old replicas), so we've left short steps for the end: changing the acceptors and removing the proposers should be pretty fast.

Chapter 5

Summary

The author of this thesis believes that consistent state replication based on consensus is still an open research problem and new solutions are yet to be found. The most famous algorithms used in production systems today are based on the idea of log replication; algorithms that do not use a distributed log are not so common.

Our goal in this thesis was to develop such a logless solution. In effect, we contributed the following:

- we presented a new abstraction — the patch machine — that formalizes the idea of storing distributed partial information about the state of a service and executing commands that modify this state.
- We introduced LPaxos, a (relatively) simple distributed algorithm that replicates patch machines in a fault-tolerant way, achieving linearizability; the algorithm does not use a distributed log and therefore does not have to deal with all the complexity of maintaining such a log. LPaxos has a formal high-level specification in TLA+ that allows multiple different implementations; we discussed some of these implementation possibilities.
- We gave a complete proof of correctness of LPaxos.
- Finally, we presented LattiStore — a key-value store built using LPaxos that supports powerful and easy to use multi-key transactions, showing that indeed LPaxos can be used to build real systems that operate with large state.

The author hopes that the idea of consistent logless replication will eventually gain more awareness in the distributed algorithms research literature.

Bibliography

[1] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 67–78. IEEE, 2000.

[2] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[3] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[4] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[5] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[6] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, page 1–12, New York, NY, USA, 1987. Association for Computing Machinery.

[7] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[8] Heidi Howard and Richard Mortier. Paxos vs Raft. *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, Apr 2020.

[9] Kyle Kingsbury. *Jepsen*, accessed November 7, 2020. `https://jepsen.io/`.

[10] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554*, 2020.

[11] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978.

[12] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[13] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[14] Leslie Lamport. Specifying concurrent systems with TLA+. *Calculational System Design*, pages 183–247, April 1999.

[15] Leslie Lamport. *TLA+ Video Course*, accessed November 6, 2020. `http://lamport.azurewebsites.net/video/videos.html`.

[16] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[17] Diego Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, Stanford University, 2014. `https://github.com/ongardie/dissertation`.

[18] J. Postel. Rfc0793: Transmission control protocol, 1981. `https://tools.ietf.org/html/rfc793`.

[19] Denis Rystsov. CASPaxos: Replicated state machines without logs. *arXiv preprint arXiv:1802.07000*, 2018.

[20] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[22] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.