DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Distributed Consensus: Performance Comparison of Paxos and Raft

HARALD NG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Distributed Consensus: Performance Comparison of Paxos and Raft

HARALD NG

Master in Software Engineering of Distributed Systems
Date: September 10, 2020
Supervisor: Lars Kroll (RISE), Max Meldrum (KTH)
Examiner: Seif Haridi
School of Electrical Engineering and Computer Science
Host company: RISE
Swedish title: Distribuerad Konsensus: Prestandajämförelse mellan Paxos och Raft

Abstract

With the growth of the internet, distributed systems have become increasingly important in order to provide more available and scalable applications. Consensus is a fundamental problem in distributed systems where multiple processes have to agree on the same proposed value in the presence of partial failures. Distributed consensus allows for building various applications such as lock services, configuration manager services or distributed databases.

Two well-known consensus algorithms for building distributed logs are Multi-Paxos and Raft. Multi-Paxos was published almost three decades before Raft and gained a lot of popularity. However, critics of Multi-Paxos consider it difficult to understand. Raft was therefore published with the motivation of being an easily understood consensus algorithm. The Raft algorithm shares similar characteristics with a practical version of Multi-Paxos called Leader-based Sequence Paxos. However, the algorithms differ in important aspects such as leader election and reconfiguration.

Existing work mainly compares Multi-Paxos and Raft in theory, but there is a lack of performance comparisons in practice. Hence, prototypes of Leader-based Sequence Paxos and Raft have been designed and implemented in this thesis. The prototypes were implemented using the Rust programming language and the message-passing framework Kompact and then benchmarked in real-world scenarios to compare the performance of Leader-based Sequence Paxos and Raft.

The results show that Leader-based Sequence Paxos and Raft have similar performance in geographically distributed deployments. However, the unpredictable leader election in Raft could greatly affect the performance if the elected leader is in an undesired location. In our experiments, the location of the Raft leader affected the average throughput by up to 35%. Furthermore, the results indicate that implementation details could have a significant impact on performance even in the parts where the algorithms are similar. By batching messages more efficiently, Leader-based Sequence Paxos achieved up to 17% higher average throughput than Raft.

Keywords — Paxos, Multi-Paxos, Leader-based Sequence Paxos, Raft, Distributed Systems, Consensus, Replicated State Machine

Sammanfattning

Med tillväxten av internet har distribuerade system blivit allt mer viktiga för att bygga mer tillgängliga och skalbara applikationer. Konsensus är ett fundamentalt problem i distribuerade system där flera processer ska komma överens om samma föreslagna värde, samtidigt som partiella fel kan ske. Distribuerad konsensus kan appliceras till olika användningsområden som låstjänster, konfigurationshanterare och distribuerade databaser.

Två välkända konsensusalgoritmer för att bygga distribuerade loggar är Multi-Paxos och Raft. Multi-Paxos publicerades nästintill tre årtionden före Raft och blev populär. Men kritiker av Multi-Paxos anser att algoritmen är svår att förstå. Av denna anledning publicerades Raft med motivationen att vara en konsensusalgoritm som är enkel att förstå. Raft delar likheter med Leader-based Sequence Paxos, en praktisk version av Multi-Paxos. Dock skiljer sig algoritmerna i viktiga aspekter som leaderval och rekonfigurering.

Befintliga arbeten jämför i huvudsak Multi-Paxos och Raft i teorin, men det saknas jämförelse av prestandan i praktiken. Av denna anledning har prototyper av Leader-based Sequence Paxos och Raft blivit designade och implementerade i denna avhandling. Dessa prototyper implementerades i programmeringsspråket Rust och message-passing ramverket Kompact, som sedan testades i verkliga situationer för att jämföra Leader-based Sequence Paxos och Raft.

Resultaten visar att Leader-based Sequence Paxos och Raft har liknande prestanda i geografiskt distribuerade sammanhang. Dock kan det oförutsägabara ledarvalet i Raft påverka prestandan avsevärt ifall den valde ledaren befinner sig på en oönskad plats. I våra experiment påverkade Raft ledarens plats den genomsnittliga kapaciteten med upp till 35%. Resultaten visar även att implementationsdetaljer kan ha en signifikant effekt på prestandan även i de delar där algoritmerna är liknande. Genom att sammanfoga meddelanden mer effektivt uppnådde Leader-based Sequence Paxos 17% högre genomsnittlig kapacitet än Raft.

Nyckelord — Paxos, Multi-Paxos, Leader-based Sequence Paxos, Raft, Distribuearade System, Konsensus, Replikerad Tillståndsmaskin

Contents

Chapter 1

Introduction

Distributed systems have become increasingly important as the growth of the internet has driven application to become more available and scalable. A distributed system consists of a set of processes which communicate and coordinate their actions by passing messages in a connected network to appear as a single coherent system to its users. Distributed systems enable parallelism, scalability and availability. However, distributed systems also need to handle problems that are naturally correlated. *Partial failure* is a characteristic of distributed systems of which some of the processes might crash or lose connection to its peers, while others continue to operate normally [6]. Furthermore, communication in asynchronous networks implies that messages have an unbounded delay and there is no synchronised global clock among the processes. These problems raise challenges in consistency, availability and partition tolerance of distributed systems.

A fundamental problem in distributed systems is *consensus*. Distributed consensus is the problem of multiple processes agreeing on the same proposed value in the presence of partial failures [8, 11]. A sequence of agreed values using consensus forms a *distributed log*, which in turn can be used to build useful abstractions such as *replicated state machine*. Replicated state machines provide fault-tolerant replication by having processes receive and process the same sequence of deterministic operations which eventually converges to the same state [27, 41]. Replicated state machines can be used for various purposes, such as lock services [5], configuration manager services [18] and distributed databases [13].

Historically, consensus has often been associated with the *Paxos* algorithm. Paxos was published by Leslie Lamport more than two decades ago [24, 25], and has since been prominent in both academia and industry, where it is used by Google's distributed lock manager *Chubby* [5] amongst others [4, 36, 46, 48].

When used for distributed logs rather than a single-value, the Paxos algorithm is optimised and often referred to as *Multi-Paxos*. Multi-Paxos is notoriously known for being difficult to understand and therefore also considered complicated to implement. Haridi et al. address these problems with the *Leader-based Sequence Paxos* algorithm [15]. The algorithm is based on Multi-Paxos and provides a step-wise refinement with complete pseudo code that makes the algorithm easy to understand and implement.

The difficulty of understanding Multi-Paxos also motivated Ongaru et al. to publish the Raft algorithm in 2014 [32]. The design of Raft focuses more on understandability in order to make it easier to implement in practice. Raft also gained popularity [3, 7, 38], and is implemented in *etcd* [13], an open-source distributed key-value database used by *Kubernetes* [23].

There has been research since that compares Multi-Paxos and Raft in theory [17, 47]. However, to the best knowledge of the author, there has not been any performance comparison of the algorithms based on empirical data from benchmarks in real-world scenarios.

1.1 Problem Statement

The lack of real-world performance comparison between Multi-Paxos and Raft implies that some uncertainty remains in which algorithm to choose when building distributed systems that require consensus. When deployed in practice, there could be different network settings that might affect the performance of the algorithms. For instance, there could be high latency between all processes due to geographically far-apart locations, or high latency only between some specific processes. Furthermore, the behaviour of the clients might also affect performance. Will the algorithms perform the same with many concurrent proposals? Investigating in different scenarios such as varying latencies and client behaviour could be useful for the applicability of the algorithms in practice.

As such, it motivates to implement Multi-Paxos using the Leader-based Sequence Paxos algorithm and compare its performance to Raft in practical deployments. Hence, the problem statement is defined as:

"What is the difference in performance between Leader-based Sequence Paxos and Raft in different real-world scenarios?"

1.2 Goals

The goal of this thesis project is to highlight the strengths and weaknesses of the algorithms when deployed in practice. By providing results from real-world benchmarks that test the latency and throughput of the algorithms, it could aid in deciding which algorithm to employ when developing distributed systems that require consensus. The results could also motivate further research on optimising the algorithms where the results show flaws. Concretely, the contributions of this thesis project are defined as follows:

- Design and implement prototypes of the Leader-based Sequence Paxos and Raft algorithms.
- Create benchmark experiments for the implementations.
- Evaluate the performance of the algorithms using results from the experiments.

1.3 Benefits and Ethics

The outcome of this thesis project mainly benefits developers that are building distributed systems which require distributed logs or abstractions based on distributed logs. The results could help developers to decide which algorithm to implement. Furthermore, the prototype implementations of Leader-based Sequence Paxos and Raft could serve as a foundation for developing libraries of the algorithms.

No direct ethical issues could be related to this thesis. However, as distributed consensus is related to data transmission over networks, certain implications of network security must be considered. While network security is beyond the scope of this thesis, it should be carefully considered when building applications based on our findings, as it may have an additional impact on design decisions.

1.4 Methodology and Methods

The work of this project is divided into four phases:

1. Conduct a literature study. Facilitate an understanding of Raft and Paxos and discover other related work to help implement the algorithms and designing the experiments in later phases.

2. Design the benchmark experiments. Determine which scenarios and parameters should be included in the experiments. Decide on programming language and frameworks to be used.

3. Implement the algorithms and experiments using the chosen programming language and frameworks.

4. Collect empirical data by running the benchmark experiments. Evaluate the collected data.

A quantitative method is chosen as a research method for this project. Quantitative methods are methods that handle measurable data. In this case, the benchmarks will collect numerical data that reflects the performance of each algorithm. Hence, a quantitative method is considered suitable for this project. The results will be analysed to gain insight into how Leader-based Sequence Paxos and Raft behave in various real-world situations.

1.5 Delimitations

Both implementations of Leader-based Sequence Paxos and Raft in this project use in-memory storage. This implies that the log and all other state variables of each process are stored in memory. Therefore, the results from this thesis do not cover other common use cases of storing the log and states such as persisting on files or databases.

Furthermore, only one programming language and message passing framework were used. Using other programming languages or message-passing frameworks could affect the benchmark results, but the relative performance between the algorithms should not vary significantly.

1.6 Outline

The rest of this thesis is structured as follows. Chapter 2 covers the relevant fundamentals of distributed systems that are needed to understand the rest of the thesis. In Chapter 3, the frameworks used to implement the algorithms are presented and motivated. Chapter 4 details the implementation of Raft and Leader-based Sequence Paxos using the frameworks presented in the preceding chapter. Additionally, the experiments used to benchmark the algorithms are also explained. In Chapter 5 and 6, the results of the experiments performed in local and distributed environments are presented. Finally, the work is summarized in Chapter 7 where the main learning points from the thesis and the potential future work are described.

Chapter 2

Theoretical Background

In this chapter, the relevant fundamentals of distributed systems are presented, with the main focus on consensus. Various abstractions of distributed systems are described along with their assumptions. Furthermore, consensus and its properties are defined formally. The Leader-based Sequence Paxos and Raft algorithms are then described, followed by an overview of related work done prior to this thesis.

2.1 Abstractions in Distributed Systems

Applications deployed in practical distributed systems could be running on a wide range of different physical machines and communication infrastructures. For instance, physical machines could vary in hardware configuration and communication infrastructures might differ in aspects such as latency and reliability. Thus, there is a purpose in defining basic abstractions of distributed systems that is applicable to a wide range of environments [6]. Three essential abstractions for distributed systems are *process*, *communication link* and *time* abstraction.

Process abstraction defines how processes behave upon failure. A failure is defined as when a process does not behave according to the intended algorithm [6]. In this thesis, the processes will be assumed to be using the *crash-stop* abstraction. This implies that a process is assumed to correctly execute its algorithm and interchange messages with other processes. A process is failed when the process stops executing steps. If a process fails at time t, it will never recover at any time after t [6, 39, 40].

The abstraction of communication link represents the network components in a distributed system. In this thesis, the *FIFO perfect link* abstraction is assumed, which is characterized by the following properties [6]:

- **Reliable delivery**: If a correct process p_1 sends a message m to a correct process p_2, then p_2 eventually delivers m.
- **No duplication**: No message is delivered by a process more than once.
- **No creation**: If a process p_2 delivers a message m with sender p_1, then m was previously sent to p_2 by p_1.
- **FIFO delivery**: If some process first sends message m_1 and afterwards sends message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

The FIFO perfect link abstraction could be implemented using TCP, but only during a single session. If a session is dropped, messages could be lost. Hence, a more realistic abstraction would be a session-based FIFO perfect link abstraction. Both Raft and Leader-based Sequence Paxos can handle dropped sessions. Raft does it by Log Reconciliation which will be described later. The version of Leader-based Sequence Paxos that is implemented in this thesis does not handle dropped sessions, however, it can easily be extended to support it as described in [15]. This does not affect the results as the code for dropped sessions would not be used in our experiments.

Abstracting time is based on the time assumption made for the system. Timing assumptions concerns with time-bounds that can be assumed for communication delays. In this thesis, *partial synchrony* is assumed. Partial synchrony implies that the system initially does not have any time-bound on communication delays. However, some time-bound will eventually hold such that there exists enough time for the algorithm to achieve its goal [6, 11]. Assuming partial synchrony, the *eventual leader election* abstraction can be derived. Eventual leader abstraction denotes that eventually, the correct processes will elect the same correct process as a leader [6]. The partial synchrony timing assumption implies that there is a possibility of arbitrary leader changes in an arbitrary period of time, but eventually, one correct leader is elected.

The crash-stop, FIFO perfect link and eventual leader abstraction combined form a distributed systems model, referred to as the *fail-noisy* model [6]. The model defines and restricts the scope of problems that need to be solved.

2.2 Consensus

Consensus is the problem of multiple processes commonly agreeing on one of the proposed values in the presence of partial failures [8, 11]. Consensus is a fundamental problem in distributed systems. Any algorithm that requires a common state or action, in a model where processes could fail, requires solving consensus [6].

In the consensus abstraction, each process in a consensus instance can *propose* and *decide* a value. Additionally, the following properties specify consensus [6, 8]:

- **Validity**: Only proposed values may be decided.
- **Agreement**: No two correct processes decide differently.
- **Integrity**: No process decides more than once.
- **Termination**: Every correct process eventually decides some value.

These properties hinder trivial algorithms from qualifying as a consensus algorithm. For example, the validity property prevents an algorithm that just decides a fixed value from satisfying consensus. Furthermore, the termination and integrity properties are also noteworthy. These properties prevent the consensus algorithm to be stuck indefinitely, however, without having any limited time-bounds. The only restriction is that each node eventually has to decide, which the partial synchrony time assumption allows an algorithm to satisfy.

2.3 Replicated State Machine

Replicated state machine is an abstraction for fault-tolerant replication. A state machine executes a sequence of commands and may produce some output. The commands are deterministic, hence the outputs of a state machine are determined by its initial state and the sequence of commands executed [6]. Replicated state machine provides fault-tolerant replication by replicating the state machine on multiple processes and having them execute the same commands in the same order [8, 24, 26, 41]. For the processes to execute in the same order, an agreement on the order of commands to execute is required. Hence, a distributed log (or any equivalent) abstraction is needed.

2.3.1 Sequence Consensus

It is feasible to use the single-value consensus abstraction previously described in Section 2.2 to build replicated state machines. One could do it by using one consensus instance for each proposed command [24]. However, using single-value consensus to agree the order of each command would be sequential. One command at a time will be decided. In replicated state machine, the aim is to agree on a growing sequence of commands, rather than to decide at most one value (per consensus instance) according to the integrity property. Thus, the single-value consensus is a mismatch to the replicated state machine abstraction. To match the abstraction with the requirements of a distributed log, some changes to the properties can be made. The following are the properties of *Sequence Consensus* [15]:

- **Validity**: If process p decides cs, then cs is a sequence of proposed commands.
- **Uniform Agreement**: If process p decides cs and process q decides cs' then one is a prefix of the other.
- **Integrity**: If process p decides cs and later decides cs' then cs is a strict prefix of cs'.
- **Termination**: If a command c is proposed infinitely often by a correct process, then eventually every correct process decides a sequence which will contain c infinitely often.

Using this abstraction, each process can propose a command c and a sequence of commands cs is decided. These properties are applied to both Raft and Multi-Paxos.

2.4 Leader-based Sequence Paxos

Leader-based Sequence Paxos is a Multi-Paxos algorithm based on having a single leader that proposes commands (and thus the order of commands) and coordinates the actions of the other processes until a new leader takes over. The algorithm is optimised around rarely changing leader which allows for minimal message round trips to reach decisions [24]. This section will describe the Leader-based Sequence Paxos algorithm in detail. However, as the leader election is a central requirement for the algorithm, it will be described first to give the reader a better context.

2.4.1 Ballot Leader Election

The Leader-based Sequence Paxos algorithm is based on having a leader that coordinates the actions of the other processes. One leader election abstraction which could be implemented in our fail-noisy model is *Ballot Leader Election* (BLE). BLE is an eventual leader abstraction that has a single output event $\langle \textsc{Leader} \mid p, n \rangle$, where p is the process elected as leader and n is its corresponding *ballot number*. A ballot number is simply a number used as a logical clock in Paxos. An implementation of BLE must fulfil the following properties:

- **Completeness**: Eventually, every correct process elects a correct process, if a majority of processes is correct.
- **Eventual Agreement**: Eventually, no two correct processes elect different correct processes.
- **Monotonic Unique Ballots**: If a process L with ballot n is elected as leader by a process p, then all previously elected leaders by p have ballot numbers m with $m < n$, and the pair (L, n) is globally unique.

An implementation of BLE using gossiping could be as follows. Each process p has a unique ballot n formed from an integer i and its process id pid, i.e. $n_p = (i, p_{pid})$. The ballot is unique to each process as the process id is unique. Each process has a local round number r and sends `HeartbeatRequest` messages tagged with r to all its peers. A `HeartbeatReply` with the highest ballot seen so far will be replied to each request using the same round number as the request. Upon a timeout, the process will check if it has got a majority of replies (including its own ballot). If so, it will elect the process that has seen the highest ballot as its leader. Upon each timeout, the local round number is incremented and new `HeartbeatRequests` are sent. If a process does not find the leader's ballot among the majority set of replies, it will increment its own ballot such that it becomes greater than that leader's ballot and wait for a new, greater, ballot to be in the majority of replies.

If a reply of r is received but the process is at round r' such that $r' > r$, then it means that the process got a late reply. To avoid timing out too early, the process will increase the delay for timeout. Further details of the BLE algorithm can be found in Appendix A.

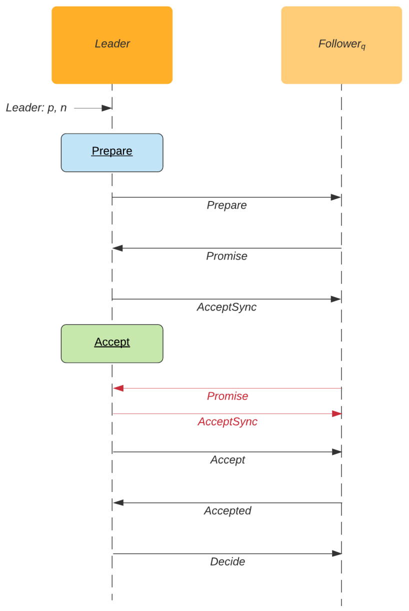


Figure 1: Sequence diagram of Paxos in normal execution without failures. The red messages represent messages after the leader has got a majority of promises and is already in the accept phase.

2.4.2 The Leader-based Sequence Paxos Algorithm

The Leader-based Sequence Paxos algorithm [15] to be described was compiled by Seif Haridi, Lars Kroll and Paris Carbone for a distributed systems course at KTH Royal Institute of Technology [22] and edX [37]. It is based on *Paxos made simple* by Lamport et al. [24] with some additional ideas from *Revisiting the paxos algorithm* by De Prisco et al. [9] and "Stoppable Paxos" presented in [28]. Unless explicitly stated differently, Leader-based Sequence Paxos will simply be referred to as "Paxos" for the rest of this paper.

The algorithm works as follows. Each process p has a $state$ variable, which tracks the current role and phase that p has in the algorithm. A process in a Paxos instance can have two possible roles; `LEADER` and `FOLLOWER`, and three possible phases; `NONE`, `PREPARE` and `ACCEPT`. Paxos is based on the assumption that the leader will rarely change. The idea is to perform the prepare phase when a new leader is elected and then pipeline commands in the accept phase. In this way, a command can be decided locally in only a single round-trip in the accept phase. The rest of this section will elaborate on this. A sequence diagram of Paxos in the general case can be seen in Figure 1.

Initially, all processes are in the `NONE` state. When a process p is elected as leader and receives the $\langle \textsc{Leader} \mid p, n \rangle$, it will set its state to `(LEADER, PREPARE)`. The goal of the prepare phase is for p to receive a majority of promises from the followers, meaning they will not append any proposed command by another process unless that process has a higher ballot (i.e. a more recent leader). Additionally, as any correct process can become the leader, it implies that the sequence of p might not contain all the elements that some other process q has. Hence, there is a need for synchronizing the sequences of the replicas in the prepare phase as well. The newly elected leader will begin the prepare phase by sending a `Prepare` message containing:

- n: The leader ballot number,
- ld: Index of the last decided element in the sequence.
- n_a: The last accepted ballot, i.e. the ballot of the latest accept phase.

If a follower receives a `Prepare` message $prep$ where n_{prep} is less than the ballot it previously promised — n_{prom} — the message will simply be discarded as it is an old message. If $n_{prom} < n_{prep}$, then the state is set to `(FOLLOWER, PREPARE)` and $n_{prom} = n_{prep}$. Furthermore, it will compare $n_{a,prep}$ to its own n_a. If $n_{a,prep} < n_a$, it implies that the follower might have a more updated sequence than the newly elected leader. Hence, it will include the suffix from ld_{prep} and its own ld in the response message `Promise`.

The newly elected leader will wait for a majority of promises. When it has received a majority of promises, the leader will synchronize its sequence by appending the suffix coming from the promise which contained the highest n_a (and longest suffix if multiple promises with same n_a). Proposals received from clients are then appended to this sequence. The leader then sends an `AcceptSync` message to all the followers which have promised. For each follower f, the `AcceptSync` message will contain the leader's ballot n and the suffix from index ld_f (received in the promise) of the leader's sequence. If the ballot of the `AcceptSync` matches the promised ballot of a follower, the follower will append this suffix after the last decided command in its sequence. This synchronises the sequences of all the followers that have promised and the leader. Furthermore, the follower sets its state to `(FOLLOWER, ACCEPT)` and replies to the leader with an `Accepted` message. The `Accepted` message contains the ballot and length of the sequence. The prepare phase is completed when a majority of followers has accepted the `AcceptSync` and the leader sets its state to `(LEADER, ACCEPT)`. From this point, the leader can upon receiving a proposal from a client directly send an `Accept` message with the proposed command and its ballot n. As our model is based on FIFO message ordering, it implies that any new command proposed to the followers will arrive after the preceding messages. Hence, proposals from clients can be pipelined in the accept phase.

Followers will respond `Accepted` to an `AcceptSync` or `Accept` message if the included ballot matches the promised ballot. Additionally, the state must be correct as well. An `AcceptSync` is only handled in the `(FOLLOWER, PREPARE)` state and an `Accept` is only handled in the `(FOLLOWER, ACCEPT)` phase. The leader keeps track of the longest accepted index each process p, $las[p]$. When a leader receives an `Accepted` message that accepts index la from p, it updates $las[p] = la$. If a majority have accepted at least up to index la, la is *chosen*. This implies that a majority of processes have appended all elements up to the chosen index in its sequence and these elements can therefore be decided. The leader keeps track of the chosen index in the local variable lc. When an index is chosen, a `Decide` message with lc is sent. All processes receiving a `Decide` message d sets its decided index ld to ld_d if $n_{prom} = n_d$.

As a command is chosen when a majority of processes have accepted it, it also explains why a leader only waits for a majority of promises when synchronizing the sequences in the prepare phase. In any majority, there will always be at least one process that has the longest sequence. This also implies that a Paxos instance can tolerate less than a majority of process failures.

One case in the normal execution of Paxos has not been considered yet. It is when the leader receives a promise in the accept phase. As the leader moves to the accept phase after getting a majority, it means that a leader could receive promises from followers in the accept phase as well. In this case, the leader simply sends the `AcceptSync` accordingly. Furthermore, if there are commands that have been decided, i.e. $lc > 0$, the corresponding `Decide` message is sent.

When a replica has decided a command, that command can be used in the application layer. For instance, in a replicated state machine, the command can be executed on the state machine. Pseudo code of the described algorithm can be found in Appendix B.

2.4.3 Reconfiguration

Reconfiguration concerns adding or removing processes in Paxos. There are different reasons to why and when reconfiguration is needed. For instance, a node might need to be removed due to hardware failure, or simply adding and removing a node could be advantageous when the workload is changed.

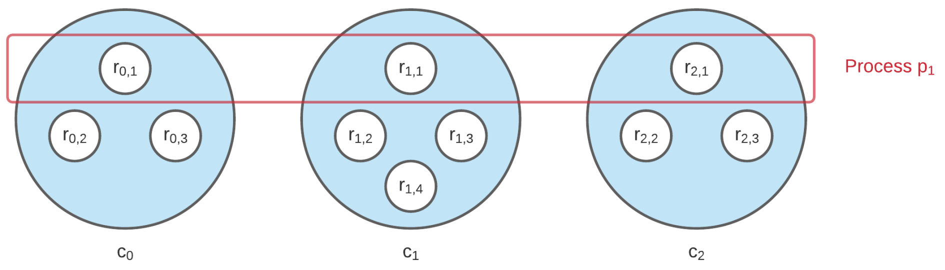


Figure 2: Example of a process acting as a replica in multiple configurations. The larger blue circles represent configurations and the small white circles represent replicas. A replica is identified by its configuration id and process id.

This is referred to as the *policy* of reconfiguration. As the reconfiguration policy is dependent on application and environment, we will only consider the *mechanism* of reconfiguration, which considers the actions and coordination the processes performs to complete a reconfiguration in a Paxos instance.

Reconfiguration in Paxos is based on modelling the system such that a process can have multiple, logically separated, instances of Paxos each with its own instance of Ballot Leader Election. We will refer such instance as a *configuration* and each process act as a *replica* in each configuration. As illustrated in Figure 2, a process could have multiple configurations at the same time. However, only one configuration can be active at a time, that is where the sequence can be extended. A configuration becomes active once it has a majority of active members.

A new special command, *stop-sign* (SS), is introduced to mark the end of a configuration. Once a leader in configuration c_i proposes a sequence σ_i containing SS_i, it must be such that SS_i is the last command in σ_i. Hence, the leader should not propose any more commands after that. Once σ_i is decided, no commands may ever extend it. Thus is σ_i the *final sequence* of c_i and c_i is called *stopped*. The stop-sign SS_i contains all information for a process to start its configuration c_{i+1}:

- Π_{i+1}: The set of processes in the new configuration c_{i+1} and their corresponding replica identifiers.
- cid: The id of the new configuration, i.e $i + 1$,
- σ_i: The initial sequence of c_{i+1}. This corresponds to the final sequence of c_i

Table 2.1: Mapping of terminology between Paxos and Raft.

Paxos	**Raft**
Sequence	Log
Decided Sequence	Committed prefix of Log
Ballot	Term
Process	Server
Element in sequence	Entry

As σ_i is the initial sequence of the new configuration c_{i+1}, there are two cases that need to be handled for each process in Π_{i+1}. If a process is a replica in both c_i and c_{i+1}, it will simply pass σ_i locally to the new configuration. In the other case, where a process p is present in c_{i+1} but not in c_i, it implies that p does not have σ_i locally. Hence, p must fetch σ_i from the other processes in c_i. This is a resource-intensive action and there are various optimisations one could use to make it more efficient, such as fetching different segments in parallel of σ_i from multiple replicas in c_i.

Once a process decides SS_i and has the final sequence σ_i, it can start its replica in c_{i+1}. The configuration c_{i+1} will be active when a majority of replicas is active and will then function as described in the previous section. A process can however not terminate its replica in the previous configuration c_i yet. When SS_i was decided, it meant that at least a majority had accepted the final sequence. The leader could crash in the middle of sending the `Decide` message, meaning some replicas have not accepted or decided the SS_i. Hence, a process' replica in the previous configuration must still be alive to help replicas of other processes to reach the final sequence. A replica is safe to terminate when all replicas have decided the final sequence. To detect when it is time to terminate a replica requires some external decision-maker. The implementation of this could be specific to certain applications and deployed environments and hence will not be considered in this thesis.

2.5 Raft

The Raft algorithm [32] uses different terminology from Paxos but with similar meaning. Table 2.1 shows the mapping of terminology between the two algorithms.

Raft and Paxos share some common characteristics. Raft also implements

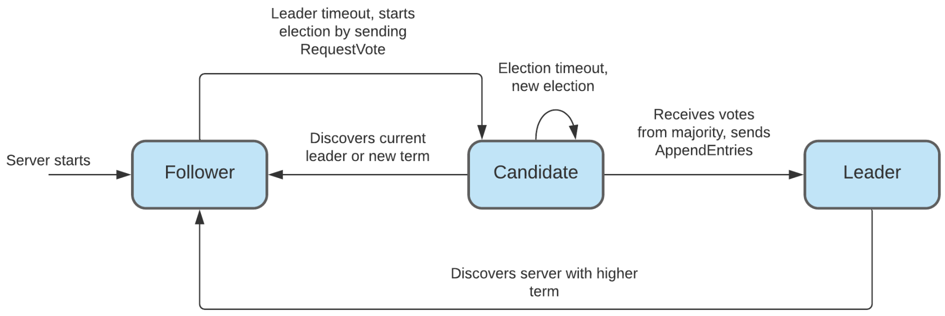

Leader timeout, starts election by sending RequestVote
Election timeout, new election
Receives votes from majority, sends AppendEntries
Server starts
Follower
Discovers current leader or new term
Candidate
Leader
Discovers server with higher term

Figure 3: Possible states of a Raft server. The descriptions above each arrow describe the trigger to change state and its subsequent action.

sequence consensus by electing a leader that will then handle proposals from clients, replicate them on the follower servers and notify them when it is safe to commit a certain entry. The main difference between Raft and Paxos is how a leader is elected. In Raft, only servers with up-to-date logs can become the leader. Furthermore, the leader election is fused with sequence consensus, as opposed to Paxos where the leader election is separated.

Raft uses a heartbeat mechanism for leader election together with *terms*, which is the logical clock in Raft. All servers start in the `FOLLOWER` state and remain as followers as long as a valid heartbeat from a leader is received before an *election timeout*. Heartbeats are sent using empty `AppendEntries` RPC (also used for replication, which is described later). If a follower receives no communication from the leader within the election timeout, it begins an election for a new leader. The follower will increment its term and set its state to `CANDIDATE`. An overview of the Raft states can be seen in Figure 3. It then votes for itself and issues `RequestVote` RPCs in parallel to all other servers. The `RequestVote` RPC contains the term and the index of the latest entry in the candidate's log. By including the index and term of the latest entry of the log, the election is restricted such that only candidates with all committed entries can win an election. The authors of Raft considers it more simple if log entries only flow in one direction: from leader to followers. Hence, when electing a leader, the candidate that wins the election must have all committed entries in its log. Upon receiving a `RequestVote` RPC, the server will check if the term is equal or greater than its own term. If so, and the candidate's last entry has an equal or greater term and index than the last entry of its own log, the vote is granted. If not, the vote is not granted. If a candidate server manages to be granted a majority of votes, it becomes the leader and starts sending heartbeat messages to all other servers to assert its authority and to prevent new elections. Else, if an `AppendEntries` RPC from another server s is received in the candidate state, the same check will be performed. If s has a higher term and a longer log, it is considered as the leader and the candidate will revert to the follower state. If the term is smaller, then the candidate rejects the `AppendEntries` RPC and remains in the candidate state.

There could also be an outcome where no leader is elected, due to split votes. This can occur when multiple servers become candidate at the same time resulting in none of the candidates granting a majority of votes. In this case, the candidates will timeout and then increase their term to initiate a new round of `RequestVote` RPCs. Eventually, there will be only one leader. To reduce the occurrence of many candidates and the risk of split votes, each server uses randomized election timeouts (within a fixed interval).

Once a leader has been elected, it can start serving client proposals. The leader appends the proposed commands to its log and then issues `AppendEntries` RPC in parallel to the followers. The leader includes the index and the term of the entry that immediately precedes the new entries. Upon receiving an `AppendEntries` RPC, a follower will check if its log contains an element matching the index and term of the preceding entry of the leader. If it does, the entries will be appended after that entry. If there was already an existing entry, that entry and all entries that follow are removed. In this way, the followers are forced to have the same log as the leader. However, if the preceding term and index from the leader do not match any entry in its current log, the new entries are rejected. In this case, the leader must find the latest entry in its log which matches with that follower's log. This is called *Log Reconciliation*, where the leader backtracks in the log and retries with a new `AppendEntries` RPC. Eventually, there will be a previous entry $prevEntry$ that exists in both the follower's and the leader's log. The follower will then accept the `AppendEntries` which will contain all elements of the leader's log starting from the index of $prevEntry$. The backtracking could be expensive if the leader has to retry by going back one entry at a time in its log. Hence, an optimisation is that the follower includes the term of conflicting entry and the first index of the entry in that term. The leader can then bypass all the conflicting elements in that term and one additional `AppendEntries` RPC is enough to synchronize that follower's log.

The leader will keep track of what the latest index each follower should have and will use it as the preceding entry index in the next `AppendEntries` RPC. Furthermore, the leader also keeps track of which index each follower has replicated its log to. When a majority of servers have replicated to index i, i is committed. This implies that all elements preceding i are also committed. The leader will include the commit index in the next `AppendEntries` RPC, which can be either a heartbeat or contain new entries. Upon receiving, all followers can apply those entries to their local state machine.

2.5.1 Reconfiguration

Raft takes a different approach to reconfiguration compared to Paxos. Raft uses the idea of *joint consensus*, where there will be a transition period where the servers from both the old and new configurations form a common temporary configuration. The servers in this temporary configuration will behave as previously described, being able to vote, candidate and replicate entries.

When the leader receives a request of reconfiguration from c_{old} to c_{new}, it appends a new entry with configuration $c_{old,new}$ to its log and starts replicating it to all the followers of both c_{old} and c_{new} as any normal proposal. As soon as the leader appends the new configuration entry to its log, it will be used for all future decisions. This implies that when the leader handles new proposals while it has $c_{old,new}$ in its log (committed or not), these proposals can only be committed if it gets a majority from *both* c_{old} and c_{new}.

Once $c_{old,new}$ is committed, it means that neither c_{old} or c_{new} can make decision without having a majority from the other configuration. Hence, if a leader crashes in $c_{old,new}$, only servers with $c_{old,new}$ in its log can be elected as the leader. As a result, the leader of $c_{old,new}$ can now append c_{new} to its log and start replicating it. Once c_{new} is committed, the old configuration is not needed and can be terminated.

The motivation of the joint consensus approach is that the cluster will not have any down-time during reconfiguration if the leader is not removed, i.e. client proposals can still be handled during reconfiguration. However, there are some issues that need to be handled to completely avoid down-time. If a new server s with an empty log is added, $c_{old,new}$ might not be able to commit any new entries because s must catch up all entries before being able to append new entries. Therefore, if c_{old} is not a majority in $c_{old,new}$, there will be down-time as some of the new nodes must first catch up to make up a majority. One optimisation is to delay the reconfiguration and add the servers but not counting them in majorities for a limited period of time to let the new servers catch up.

Another possible issue is when the cluster commits c_{new} in $c_{old,new}$. If the leader in $c_{old,new}$ that commits c_{new} is not included in c_{new}, it will step down and there will be no leader until the next election succeeds. Furthermore, this also means that while replicating c_{new}, the leader will manage a configuration it is not part of, because as soon as a configuration is added to the log, it is used instantly. This implies that the leader will replicate entries including the c_{new} entry, but not count itself to the majority.

2.6 Related Work

Comparisons between Paxos and Raft. There exists previous work that compares both single-value Paxos and Multi-Paxos to Raft. Howard et al. attempted to determine which of Multi-Paxos and Raft is better by describing Multi-Paxos using Raft's terminology and comparing them theoretically [17]. The comparison accounts both understandability and performance. The outcome of their work indicated that the impression of Raft being easier to understand than Multi-Paxos is based on the paper's clear presentation rather than the algorithm itself being significantly more understandable. Furthermore, the algorithms are deemed very similar, differing only in leader election. Raft's leader election is considered more lightweight, as only candidates with up-to-date logs can become the leader which avoids sending parts of the sequence during leader election. Additionally, when a new leader is elected, Multi-Paxos might send commands that already exist in the log of the followers. This is not the case in Raft.

In [10], Dubinin et al. implemented distributed semaphores using Raft and single-value Paxos. The performance was evaluated using simulations on coloured Petri nets and showed that the Raft implementation is more efficient than single-value Paxos, which essentially is sequential.

Other Consensus algorithms. In addition to Paxos and Raft, there exist other consensus algorithms. ZooKeeper Atomic Broadcast (ZAB) [19] is a well-known algorithm. ZAB shares many characteristics with Paxos but is designed for primary backup systems rather than replicated state machine, with focus on handling the client requests in FIFO order.

Another consensus algorithm is Viewstamped Replication (VR) [30]. VR is similar to Raft but uses a simpler approach for leader election. If a leader times out, the replica with the smallest IP-address is chosen as leader. However, this requires transferring and synchronising sequences as in Paxos.

Chapter 3

Frameworks for Building Distributed Systems

This chapter presents and motivates the tools used to implement Paxos and Raft. This includes the choice of programming language, message-passing and testing framework, and other relevant libraries.

3.1 Rust

Rust is a natively compiled programming language which provides memory safety using a powerful compiler rather than garbage collection [14]. By avoiding garbage collection, cleanup of resources become predictable and the Rust compiler automatically inserts deallocation code based on *lifetimes* of data. The choice of Rust for this project was based on its speed and memory safety. Furthermore, Rust was also the language used in the frameworks described below.

3.2 Kompact

Kompact [20] is a message-passing framework in Rust that provides a hybrid approach between the *Actor* model [16] and the *Kompics component* model [1, 2]. An actor or component is defined as a light-weight process with internal state in its respective model. Actors and components can communicate with each other using messages (Actor model) or events (Kompics component model). The core difference between the models is in how the receivers of messages and events are abstracted. In the Actor model, the receivers are explicitly addressed in contrast to the Kompics component model, where the receivers are abstracted away and instead relies on channels with subscription-checking [21]. Kompact addresses that both these models have their advantages and disadvantages and provides a hybrid approach where the user can utilize both models in a suitable manner. In Kompact, a component is also an actor and vice versa. Hence, we will simply refer to these using the term “component”.

Kompact was considered suitable for this project as the clear abstractions of the Kompics component model would be useful to implement the relations of replicas and election components in Paxos and the Actor model would benefit in performance by avoiding subscription-checking. Additionally, Kompact has shown promising results when building similar systems such as Atomic Register in comparison to other message-passing frameworks [21].

The remainder of this section presents some of the core features of Kompact that are needed for the implementation.

3.2.1 Communication in Kompact

Local communication between components in Kompact is through triggering events in channels (Kompics component model). Events can only travel in certain directions of a channel defined by its *port*. A port p can *indicate* events of type i and *request* events of type r. If a component *provides* p, then it can only trigger outgoing events of i and receive incoming events of r. If a component *requires* p, then it can only trigger outgoing events of r and receive incoming events of i.

Components can also communicate by messaging to data structures that represent addresses of the receiver (Actor model). Local and remote addresses are represented by `ActorRef` and `ActorPath` respectively. An `ActorRef` is typed, i.e. a component can only send a message of a valid type to other local components. However, `ActorPath` is not typed, as it is generally harder to restrict what can be sent over the network.

3.2.2 Timers

Kompact supports timers in components, such that a component can schedule some work to be performed in the future. The timers are either triggered once or periodically and allow for the state of the component to be changed upon timeout.

3.3 Message-Passing Performance Suite

To evaluate the performances of the algorithms implemented in Rust using Kompact, a benchmarking framework was needed. The Message-passing Performance Suite (MPP) [21] was chosen as it provides a convenient benchmarking suite that can be deployed in a distributed setting.

MPP supports different running modes, such as *remote* and *fakeRemote* [21]. In “remote” mode, a configurable number of executables are started via ssh and hence is useful for distributed benchmarks. The “fakeRemote” mode is similar to the “remote” mode but starts the executables in temporary directories on the host instead. This is useful for testing and for simulating distributed environments without having latency affecting the results.

MPP measures the execution time from start to end of a benchmark defined by the user. The user defines a set of experiment parameters and MPP will run each benchmark with certain parameters at least 30 times until the relative standard error is below 10%. The results are then stored as comma-separated values (CSV) files that can be used for automated plotting.

3.4 LogCabin - The Original Raft

Despite the clear presentation and skeleton code of the Raft algorithm, important implementation details regarding performance remained as future work in the original paper [33]. For instance, a naive implementation of Raft using synchronous RPCs and without pipelining would be very inefficient. Many of these issues were later addressed in the dissertation of the Raft author Diego Ongaro [31], in which an implementation in C++ called LogCabin [29] was presented. To support pipelining, an optimistic approach is used. The leader assumes that an issued AppendEntries RPC to a follower will be successful and therefore instantly updates the next index for that follower, rather than wait for an acknowledgement. This allows RPCs to be pipelined at the cost of more bookkeeping in case an RPC fails. If an RPC times out, the leader must decrement the follower’s index to the original value before the pipeline to retry. If an AppendEntries fails on consistency check (i.e. mismatching previous entry), the leader decrements the next index according to the hint and then retries sending the first AppendEntries RPC in the pipeline.

LogCabin is a full system with disk storage, cryptographic security of the stored data and network implementation. Using LogCabin to compare to another Paxos implementation would be slightly misleading. The results would be heavily affected by the differences in implementation of other parts of the system such as storage and network, rather than the differences between Paxos and Raft. Additionally, it would be difficult to integrate LogCabin into MPP due to its lack of support for C++. Thus, another Raft library (described in the next section) was chosen to be compared to Paxos. However, to ensure that the chosen Raft library could act as a fair representation for Raft, there was a need to show that its performance is at least as good as LogCabin. Hence, a comparison between these was carried out as well.

3.5 TiKV raft-rs

TiKV raft-rs [44] is a Raft implementation in Rust used in the distributed transactional key-value database TiKV [43]. This library provides an implementation of Raft represented as a data structure in Rust, where the user calls certain methods on it to handle incoming messages and fetch outgoing messages. Furthermore, the user must provide its own storage and network implementations. TiKV raft-rs was deemed as a suitable library to use, as it supported pipelining and batching but also gave the freedom to use the desired storage and network implementations. This allowed for a more fair comparison as similar storage and network implementations could be used in both Raft and Paxos. More specifically, it implied that in-memory storage could be used and Kompact could be used for passing messages.

At the time of this project, the latest version of TiKV raft-rs was version 0.6.0-alpha and worked as follows. A Raft server is simply a data structure (`struct`) in Rust, called `RawNode`. Some of the core methods of `RawNode` for a user are:

- `new(c: Config, store: T)`. Constructs a `RawNode` object with the given config and store. The config is a data structure including settings such as process id, election timeout, heartbeat period and to use batching or not. The store parameter is of a generic type, but it must implement a certain collection of methods (*trait* in Rust) needed for storage. The store is used to store the raft log, metadata and configuration.
- `tick()`. The tick method advances the Raft server's logical clock. In the configuration of a `RawNode`, all the time-related settings such as timeouts and how often a leader sends heartbeats are defined by the number of ticks. For instance, if a user defines the leader heartbeat period to 2 ticks, and then calls `tick()` every second, a leader heartbeat is sent every 2 seconds.

- `ready()`. Returns the latest state of the `RawNode`. This ready state includes the committed entries and outgoing messages to other Raft servers. The committed entries should be handled by the user's application and the outgoing messages need to be transmitted to the other Raft servers by the user's network implementation.
- `step(m: Message)`. Upon receiving a message `m` via the user's network implementation, the user should call `step(m)` to let the raft server process it. This will change the state of the Raft server and produce the outgoing messages as needed (that will later be in `ready()`). If a certain number of ticks has passed and no messages from the leader have been handled by `step`, the `RawNode` will start a leader election and produce the corresponding outgoing messages.
- `propose(data: Vec<u8>)`. Makes the Raft server propose a normal entry that is serialised. The data correspond to the data a client proposes. If successful, the same data will be among the entries of the committed entries in a ready state later.
- `propose_membership_change(c: Configuration)`. Method to propose a reconfiguration. This method proposes configuration using Joint Consensus as described in Section 2.5.1. However, the library states that this is still an experimental feature, but supports the basic functionality needed in this project, that is replacing one node with another [33].

As seen, the library only handles the incoming messages in `step()` and generates outgoing messages in `ready()`, but not how the messages are being sent. In our project, it will be handled by Kompact. Thus, our implementation of Raft uses message-passing as opposed to remote procedure calls (RPC) that were proposed in the original Raft paper [32]. An example of how TiKV raft-rs could be used with a message-passing framework such as Kompact can be found in Algorithm 1.

To store metadata and the log of a raft server, the library requires a data structure that implements the `Store` trait, which is passed as an argument to create an object of `RawNode`. This allows for the user to provide a storage implementation fitting for its purpose, which in this case is in-memory storage.

Algorithm 1: RaftReplica using TiKV raft-rs

```
c;                      /* parameters for TiKV raft-rs */
$rawRaft \leftarrow RawNode.\textsc{new}(c)$;    /* the TiKV raft-rs data structure */
$d_{tick}$;                              /* outgoing delay */
$d_{ready}$;                             /* decided delay */
$\textsc{startTickTimer}(d_{tick})$; /* schedule periodic timeout event in $d_{tick}$ timeunits to increment the logical clock */
$\textsc{startReadyTimer}(d_{ready})$;   /* schedule periodic timeout event in $d_{ready}$ timeunits to handle outgoing messages and committed entries */
---------------------------------------------------------------
Upon $\langle \textsc{Proposal} \mid p \rangle$ from client
    if $p.reconfiguration = true$ then
        $rawRaft.\textsc{propose\_membership\_change}(p)$;
    else
        $rawRaft.\textsc{propose}(p)$;
Upon $\langle \textsc{rawRaftMessage} \mid m \rangle$ from peer
    $rawRaft.\textsc{step}(m)$;
Upon $\langle \textsc{Tick Timeout} \rangle$
    $rawRaft.\textsc{tick}()$;
Upon $\langle \textsc{Ready Timeout} \rangle$
    $ready \leftarrow rawRaft.\textsc{ready}()$;
    $outgoing \leftarrow ready.\textsc{messages}$;
    $committed \leftarrow ready.\textsc{committed\_entries}$;
    foreach $m \in outgoing$ do
        send $\langle \textsc{rawRaftMessage} \mid m \rangle$ to $m.\textsc{to}$;
    foreach $e \in committed$ do
        /* handle committed entry                    */
```

Chapter 4

Implementation of Raft and Paxos

This chapter will present how the Paxos and Raft algorithms were implemented using the tools presented in Chapter 3. First, the experiments are described in order to motivate why the system is implemented as it is. Then, an overview of the system is presented before going deeper into details of the implementation of the respective algorithm.

4.1 Experiment Design

The design of the experiment is rather simple. An experiment consists of a number of processes that forms a consensus instance and a single client. The client proposes values to the consensus instance and the experiment is completed when the client has received responses to all its proposals. For simplicity, the data proposed by the client are of type 64-bit unsigned integers. Different parameters of the client allow for testing different scenarios of the consensus instance. The benchmark is divided into three main experiments.

Throughput of normal proposals. The client proposes n normal proposals, and at most c concurrent proposals at a time. Concurrent proposals are defined as proposals that are proposed but not responded yet and represent how heavily loaded the system is.

Each proposal is unique and the experiment is complete when the client has received responses of all n proposals. The average throughput is calculated by $n/t_{execution}$.

Throughput in presence of reconfiguration. The client proposes n normal proposals, with at most c concurrent proposals at a time. Upon receiving $n/2$ responses, a reconfiguration to replace one process is proposed by the client. The rest of the $n/2$ proposals will continue to be proposed during and after reconfiguration. The experiment is complete when at least one process has confirmed successful reconfiguration and n responses have been received. The average throughput is calculated by $n/t_{execution}$.

In Raft, there could be a difference in performance depending on which server is replaced among the first configuration. If the leader is removed, a new election must occur. If a follower is removed, the Raft cluster can continue without a new election. Hence, we will also split up the reconfiguration experiments in Raft into these two scenarios. This is not relevant in Paxos as the configurations are logically separated. Paxos has instead two variants of transferring the final sequence to new processes during reconfiguration. One approach is to let the new processes pull the sequence. Another approach is letting the continued processes transfer the sequence to the new processes when deciding the StopSign. The reconfiguration experiments of Paxos are therefore split into two parameters; *pull* and *eager*.

Latency of normal proposals. Similar to the throughput of normal experiments, but with the number of concurrent proposals set to 1. The client will thus only propose a new proposal when the previous proposal has been responded. Additionally, the client keeps track of the timestamps of when each proposal was proposed and responded. This allows for more in-depth results than just average latency.

4.2 System Overview

In this section, we will describe how the experiment is implemented in Rust using MPP, Kompact and TiKV raft-rs[1]. A distributed benchmark in MPP requires implementations of a benchmark master and a benchmark client. The benchmark master bootstraps the benchmark clients and then runs the user-provided implementation to measure execution time. Hence, the experiment client component is implemented in the benchmark master. The consensus algorithms are implemented as benchmark clients, such that one benchmark client corresponds to one consensus process. As such, one needs to start the MPP benchmark with n benchmark clients in order to have a consensus instance with n processes. An overview of the system can be found in Figure 4.

As Raft and Paxos both provide the same abstraction, some parts of the implementation in Kompact can be shared by both algorithms. In such cases, we

[1]Source code: `https://github.com/haraldng/kompicsbenches/tree/master/kompact/src/bench/atomic_broadcast`

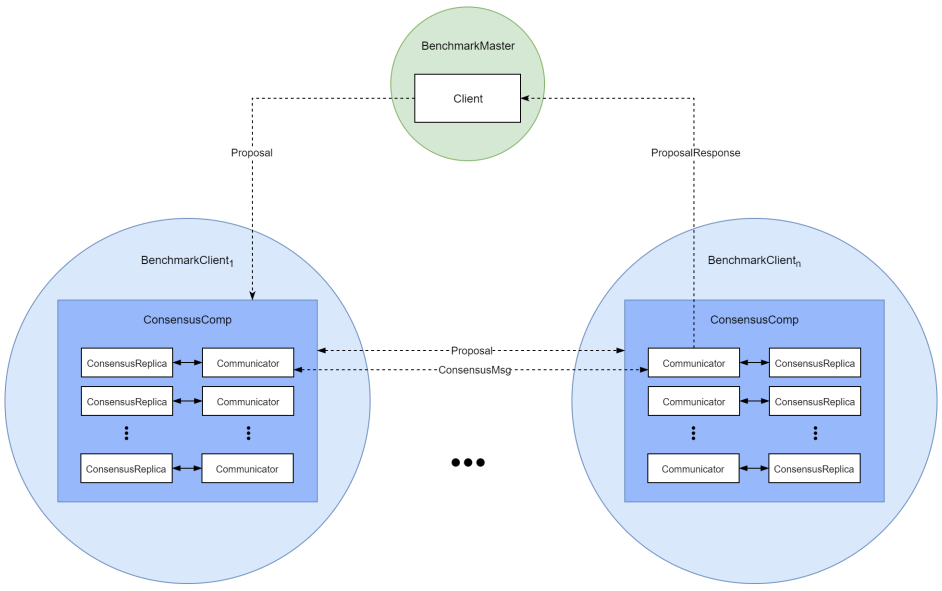


Figure 4: Overview of the system. Proposals from clients are sent to a consensus component that could forward them to its peers.

will simply refer to the shared implementation using the prefix `Consensus`. If a word is prefixed with `Consensus`, it can be replaced with `Raft` or `Paxos`. Furthermore, the word “component” refers to a Kompact component unless stated differently.

A benchmark client consists of a `ConsensusComp` component that can have one or many `ConsensusReplica` child components. The consensus algorithm is implemented in the replica component. Furthermore, a `ConsensusReplica` is connected to a `Communicator` component through a local Kompact channel. The communicator serves as a messenger to its connected replica component, sending and receiving all the in- and outgoing messages via the network. The reason for this design is to separate the work of the consensus algorithm and messaging to gain more local parallelism. The consensus algorithm is likely to be dominated by accessing data structures, while the messaging work includes serialising and deserialising messages. As such, a replica triggers an outgoing message using the communicator port. The communicator sends the network message to the communicator of the intended recipient, which upon receiving the message passes it to its connected `ConsensusReplica`. When a proposal has been decided, the leader replica will trigger a response to the client that is also sent by the `Communicator`. An illustration of the connection between a `ConsensusReplica` and `Communicator` can be found in Figure 5.

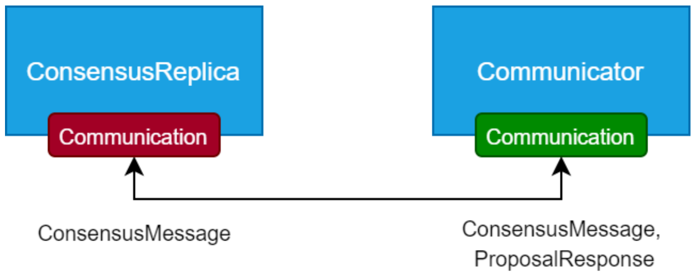


Figure 5: Connection of consensus replica and communicator. The red port represents a required port while the green port represents a provided port.

Proposals from the client are received by a consensus component. The consensus component keeps track of which of its replicas is active and which process is the leader in the active configuration. When a proposal is received, a check is performed to see if its active replica component is the leader. If so, the proposal is deserialised and triggered to that `ConsensusReplica`. Otherwise, the consensus component will forward the proposal without deserialising it to the consensus component of the leader process. To keep track of the current leader, each replica component will notify its supervisor when a change of leader occurs.

4.3 Raft

The Raft algorithm is implemented as a `RaftComp` component that has two child components; a `RaftReplica` that is connected to a `Communicator` component. There is only one replica and communicator per `RaftComp` because the same replica is used even after reconfiguration.

The replica component contains a field called `raw_raft` of type `RawNode`, the TiKV raft-rs data structure representation of a Raft node. To advance the logical clock of `raw_raft`, the replica component uses a periodic Kompact timer that calls the `tick()`. In the same timeout, the latest leader will also be checked. If the leader has changed, it will notify its supervisor.

When a proposal is triggered by the supervisor, the replica component will propose the value of reconfiguration in its `raw_raft`. This will generate outgoing messages and committed entries which are fetched periodically using another separate Kompact timer. Upon each timeout of the timer, the outgoing messages and committed entries are fetched by accessing the ready state of `raw_raft`. The outgoing messages contain the recipient's process id and each of them is triggered on the communication port, where the communicator will send the message to the corresponding process' actor path.

The committed entries require more handling depending on its type. Normal entries are simply triggered on the communication port as a proposal response to the client if the replica is the leader. A reconfiguration entry marks either the beginning or the end of the intermediate joint consensus configuration that includes both the processes of the old and new configuration. When handling these entries, it requires calling the corresponding method on `raw_raft` using `begin`- or `finalize_membership_change()`. When the reconfiguration has been finalized, the new configuration will be active from that point. If the leader was removed, every continued process will create a one-off timer to start campaigning as a leader unless another process already has become the leader. This timer is set to a random value within a certain period to lower the probability of split votes in the election. If a process detects itself was removed, it will stop its timers to stop all `raw_raft` activity.

4.3.1 Storage

The constructor of the `RawNode` data structure has a storage parameter which must be of a type that implements the `Storage` trait [35]. In this project, a provided storage called `MemStorage` [34] from the TiKV raft-rs library was used. `MemStorage` stores the metadata and the log in memory. The log is simply stored as a vector in Rust.

4.3.2 Comparing TiKV raft-rs with LogCabin

To prove that our Raft implementation performed at least as good as LogCabin, LogCabin needed to be tested as well. Two problems had to be solved to facilitate a fair comparison.

Inability to integrate into MPP. As LogCabin could not be integrated into MPP, a benchmark program in the LogCabin repository was used. The benchmark program creates clients that propose to a LogCabin cluster a number of times and records the time it takes to get the response of all proposals. However, the API for proposing in LogCabin in synchronous, meaning that a call to propose only returns when the proposal has been responded. To generate concurrent proposals, more clients have to be created on separate threads. Furthermore, a script was created to automatically bootstrap the LogCabin processes and run the experiment for 30 runs per experiment parameter.

In-memory storage. At the time of writing, LogCabin does not support a functioning in-memory storage. A workaround for this problem was to use a virtual memory file system `tmpfs` that appears as a mounted file system but stores the data in memory [45].

The parameters for the implementations and the experiment were set to the same values. As in MPP, the proposals in LogCabin were set to be 8 bytes in size. Furthermore, the parameters for the Raft algorithm such as timeouts and the maximum batch size were set to the same values in LogCabin and TiKV raft-rs. As in TiKV raft-rs, snapshotting was turned off.

4.4 Paxos

The Paxos algorithm is implemented as a `PaxosComp` with one or multiple `PaxosReplica` child components. Similar to the Raft implementation, a replica is connected to a `Communicator`. Additionally, as each Paxos replica is logically separated with its own leader election, a replica component is connected to a `BallotLeaderComp` as well.

A Paxos replica has a `raw_paxos` field which is the Paxos algorithm implemented as a data structure. The outgoing messages of `raw_paxos` are fetched periodically with a Kompact timer and sent using the required communication port. Unlike Raft, the decided entries are not bundled together with the outgoing messages. Instead, a separate timer is used to read the decided entries. Normal entries are responded to the client by the leader. If the decided entry is a StopSign message, the replica will notify its supervisor with a reconfiguration event that contains the StopSign message, where the final sequence is an immutable shared reference (`Arc` in Rust). The supervisor will then send `ReconfigurationInit` messages to all the new processes in the new configuration c_i. This message contains the configuration id i, processes in the configuration Π_i and metadata of the previous configuration c_{i-1}. The processes of Π_i are also tagged to indicate which of the processes were continued, i.e. were part of c_{i-1} as well. The metadata of the previous configuration includes the configuration id $i-1$ and the length of the final sequence of c_{i-1}. When a new process receives a `ReconfigurationInit` message, it needs to have all the final sequences from previous configurations (i.e. all decided elements) before it can start its replica in c_{i-1}. There are two approaches for a new process to receive these final sequences.

Pull final sequences. The new node can start pulling the final sequence of c_{i-1} from the continued processes by requesting different parts using indices. The sequence requests are received by the `PaxosComp`. If it has the final sequence of the requested configuration id among its previous final sequences, then it will clone the requested part and respond to the requestor. If the configuration id is instead the currently active configuration, it means that the replica has not reached the final sequence, but could possibly still have the requested entries. Hence, the `PaxosComp` will outsource this request to its replica by triggering a local message to it. The replica will transfer the requested entries if they are decided in the sequence or else include a failed tag in the response. A sequence transfer also includes the metadata of the next previous sequence. This allows new processes to pull final sequences of multiple configurations in parallel.

Eagerly transfer final sequences. Another approach is the eager transferring of the final sequence. When a continued `PaxosComp` receives a reconfiguration event from a replica, it will instantly send the `ReconfigurationInit` and a segment of the final sequence to all the new processes. Hence, when using this approach, a new process will simply wait for all the segments to be eagerly transferred to it.

If the whole final sequence has not been received after a certain period, the new process will pull the final sequence from those processes it knows to have reached the final sequence (i.e. all the peers it has received `ReconfigurationInit` from). This applies to both approaches.

4.4.1 Raw Paxos

The raw Paxos data structure is designed similar to TiKV raft-rs, giving users the freedom to use it with the desired network, storage and leader election implementation. Furthermore, it allowed for a more fair comparison between the algorithms. In order to understand the internals of raw Paxos, the API exposed to the user will first be presented. The API for raw Paxos is as follows:

- `with(cid: u32, pid: u64, p: Vec<Peer>, s: Storage<S,P>)`. Constructor for the raw Paxos data structure. The `cid` and `pid` are the configuration id and process id to differentiate replicas of the same process. Each Paxos object in a cluster should have unique `pid`. The peers in the initial configuration are stored as a vector in the argument `p`. The `Peer` type is simply a wrapper for the process id of type `u64`. The storage parameter `s` is of type `Storage` that has two trait arguments; `S: PaxosState` and `P: PaxosSequence`. `PaxosState` is the implementation for storing and accessing the state of Paxos such as the promised and accepted ballot and the length of the decided sequence. `PaxosSequence` implements methods for storing and accessing elements of the sequence. This includes appending and reading elements, and convenient methods specific for Paxos such as getting a suffix or appending on a prefix of the sequence.

- `handle(m: Message)`. Handles a Paxos message sent from another raw Paxos object. A message is tagged with the `pid` of the sender and receiver and could be of any of the Paxos messages as described in Appendix B. Additionally, a message could also be a forwarded proposal, a proposal that was forwarded by a peer if a user tried to propose with a raw Paxos that was not the leader. By handling messages, the internal state of raw Paxos will be changed and outgoing messages will be generated.

- `handle_leader(l: Leader)`. When a user's provided leader election has elected a leader, the user should create a `Leader` object with the elected leader's process id and ballot. This should then be passed as an argument to `handle_leader()`. This will cause the raw Paxos object to change its state to leader and start the prepare phase if the ballot is up-to-date.

- `propose_normal(data: Vec<u8>)`. Proposes a normal proposal with serialised data. If the raw Paxos is not the leader, it will forward the proposal to the leader.

- `propose_reconfiguration(c: Vec<Peer>)`. Proposes a reconfiguration. The new configuration will have configuration id `cid+1`. If the reconfiguration is successful, there will be a StopSign message in the outgoing messages which the user should handle accordingly.

- `get_outgoing_msgs()`. Returns the outgoing messages that this raw Paxos object has produced. The user should then use its network implementation to send the message to the correct peer.

- `get_decided_elements()`. Returns a slice, i.e. a reference to a contiguous sequence of elements [42], of the decided elements since the last `get_decided_elements()` call. If there are no new decided elements since the last call, an empty slice is returned.

- `stop_and_get_sequence()`. This method is a wrapper for the corresponding method for the storage of the raw Paxos object. Stops the raw Paxos by turning the sequence into an immutable shared reference and returns it to the user. This method should only be called when a reconfiguration has been successful, i.e. a StopSign has been handled in `get_decided_elements()`. The raw Paxos can still handle and produce outgoing messages, but cannot extend its sequence anymore. Its purpose is to help its peers reach the final sequence, as described in Section 2.4.3.

The algorithm is mainly implemented according to the pseudo code that can be found in Appendix B. Outgoing messages are put into a vector and on each `get_outgoing_msgs()` call, the vector is returned to the user. Upon deciding an element, raw Paxos simply updates the decided index in its storage. When a user later calls `get_decided_elements()`, the raw Paxos will get a slice of the sequence in its storage using the previously cached decided length and the current decided length. An example of how raw Paxos could be used with a message-passing framework such as Kompact can be found in Algorithm 2.

4.4.2 Storage in Raw Paxos

To maintain fairness, the storage used in raw Paxos is implemented similarly to `Memstorage` used in Raft. The metadata is represented as data structures and the sequence is stored as a vector. As in TiKV raft-rs, the library supports custom implementations of the storage by implementing the `PaxosState` and `PaxosSequence` traits.

4.4.3 Optimisations in Raw Paxos

Batching elements in Accept messages. As previously mentioned, designing raw Paxos as a plain Raft data structure gives the advantage of flexibility to users. However, it also comes with some disadvantages. For instance, having an outgoing message queue that is periodically fetched to then be sent could harm performance when used with message-passing frameworks such as Kompact. It results in causing periods of very heavy workload interchanged with periods of no work at all for the network thread. Hence, it would be beneficial to provide the option of batching for users, which would limit the number of messages to the same process and reduce overhead. A simple way of batching in raw Paxos is to batch the elements of `Accept` and `AcceptSync` messages. This could be implemented in raw Paxos by caching the ballot number and index in the outgoing queue of the latest `Accept` or `AcceptSync` message of each peer. When a new `Accept` message is produced to peer p, the elements can be appended to the cached message of p if the ballot is of the same value. Else, a new message is added to the outgoing queue and the

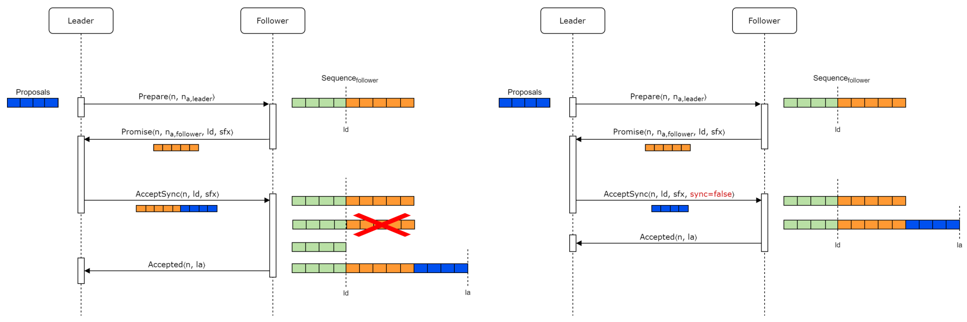
Algorithm 2: PaxosReplica using Raw Paxos

```
1: $cid$;                                          /* configuration id */
2: $pid$;                                                /* process id */
3: $\Pi$ ;                                             /* set of peers */
4: $s$ ;                                           /* desired storage */
5: $rawPaxos \leftarrow RawPaxos.\text{WITH}(cid, pid, \Pi, s)$ ;  /* the raw Paxos
     data structure */
6: $d_{out}$ ;                                       /* outgoing delay */
7: $d_{decide}$ ;                                     /* decided delay */
8: startOutgoingTimer($d_{out}$);    /* schedule periodic timeout
     event in $d_{out}$ timeunits to fetch outgoing messages
     */
9: startDecideTimer($d_{decide}$);    /* schedule periodic timeout
     event in $d_{decide}$ timeunits to check decided elements
     */
--------------------------------------------------------------------
10: Upon ⟨Leader | $(l, n)$⟩
        /* leader event triggered by BLE                          */
11:     $rawPaxos.\text{HANDLE\_LEADER}((l, n))$;
12: Upon ⟨Proposal | $p$⟩ from $client$
13:     if $p.reconfiguration = true$ then
14:         $rawPaxos.\text{PROPOSE\_RECONFIGURATION}(p)$;
        else
15:         $rawPaxos.\text{PROPOSE\_NORMAL}(p)$;
16: Upon ⟨RawPaxosMessage | $m$⟩ from $peer$
17:     $rawPaxos.\text{HANDLE}(m)$;
18: Upon ⟨Outgoing Timeout⟩
19:     $outgoing \leftarrow rawPaxos.\text{GET\_OUTGOING\_MSGS}()$;
20:     foreach $m \in outgoing$ do
21:         send ⟨RawPaxosMessage | $m$⟩ to $m.\text{TO}$;
22: Upon ⟨Decide Timeout⟩
23:     $decided \leftarrow rawPaxos.\text{GET\_DECIDED\_ELEMENTS}()$;
24:     foreach $d \in decided$ do
25:         if $d = StopSign$ then
                $finalSequence \leftarrow rawPaxos.\text{STOP\_AND\_GET\_SEQUENCE}()$;
                /* handle StopSign and final sequence          */
            else
26:             /* handle normal decided element               */
```

cached metadata for p is updated with that ballot number and index in the outgoing message queue. Every time the outgoing message queue is flushed by the user, i.e. when calling `get_outgoing_msgs()`, all the cached metadata for each peer is cleared.

Send only latest decide. Another optimisation related to the periodic flushing of outgoing messages is to only send the latest decide. As a decide message also decides the elements prior to it, only the latest decide message to a peer is needed in the outgoing message queue. This reduces the number of decide messages that are sent, as the ones before the last one are redundant. Using a similar approach to how accept messages are batched, the index of the latest decide and its ballot in the outgoing message queue for each peer can be cached. When a new decide message to peer p is issued, one can check if there already is a cached decide message d with the same ballot to p in the outgoing message queue. If there is no accept message to p behind d, the value of d can be updated to the latest decide value. This check is important to prevent a decide message being handled before the peer has received the accept. All the cached metadata of the latest decided message for each peer is cleared upon the `get_outgoing_msgs()` call.

Max AcceptSync. When a new leader emerges, it will perform a prepare phase to retrieve knowledge of each follower's sequence. Based on the promises received in the prepare phase, the leader will send an `AcceptSync` message to synchronise the sequences of all the processes. The synchronization is based on the follower with the most up-to-date sequence, that is the highest accepted ballot and suffix length in its promise. Additionally, the suffix in `AcceptSync` is also appended with the new elements that the leader proposes. In the pseudo code, the suffix in `AcceptSync` is appended on the last decided element of the follower. This implies that if the `AcceptSync` suffix is based on the suffix of a follower f, then f will drop its elements from the decided index and then append the same elements again together with the new proposals when handling the `AcceptSync`. As one can see in Figure 6, it is redundant for f and all other followers with the same promise as f, to receive the synchronizing suffix, because they already have the same elements in their sequence. Instead, the leader could only send the new proposals to those followers. Hence, we modify the `AcceptSync` message slightly by adding a `sync` flag. To all the followers with the highest promise, i.e. the highest ballot and sequence length, the `AcceptSync` will set the `sync` flag to `false` and only send the new proposals. For the other followers, the `AcceptSync` will remain as before but with `sync = true`. When a follower receives an `Ac-ceptSync`, it will append the suffix to its existing sequence if the `sync` flag


Leader
Follower
Proposals
Sequence$_{follower}$
Prepare(n, n$_{a,leader}$)
Promise(n, n$_{a,follower}$, ld, sfx)
AcceptSync(n, ld, sfx)
Accepted(n, la)
ld
la
Leader
Follower
Proposals
Sequence$_{follower}$
Prepare(n, n$_{a,leader}$)
Promise(n, n$_{a,follower}$, ld, sfx)
AcceptSync(n, ld, sfx, sync=false)
Accepted(n, la)
ld
la

(a) No Max AcceptSync. **(b)** Max AcceptSync.

Figure 6: Prepare phase to a follower with the maximum promise.

is `false`. Otherwise, it will drop the elements since the last decided index and append the suffix.

There are two main concerns of this approach:

1. *Could two followers that promises with the same ballot, have same suffix length but different suffixes?* No, except for one case that must be handled carefully. If the highest promise p_{max} consist of an empty suffix, the leader cannot determine if the followers are perfectly up-to-date or have responded with an empty suffix because its ld is less than ld_{leader}. Hence, two followers with p_{max} could have different sequences that are both a prefix of the leader's sequence at index ld_{leader}. In such case, the sequences of followers will end up unsynchronized after the prepare phase as both only appends the new proposals to its sequence. To prevent this, the max `AcceptSync` should only be sent to followers with $ld >= ld_{leader}$ if the suffix of p_{max} is empty.

   In other cases, if the promises have the same accepted ballot, then they must have appended elements to the sequence issued from the same leader with the same ballot. As the `Accept` messages from the same process are FIFO ordered, they must have the same elements. If one follower would have received more elements than the other in the same ballot, the length of their suffices would not be equal.

2. *Could a follower have appended more elements after promising?* It could, but then the follower must have changed its promise, resulting in the `AcceptSync` being discarded when received. If an `Accept` message arrives late from the old leader, then it will be discarded because the follower has already changed its promise.

This change could possibly reduce the amount of data in `AcceptSync` messages and avoid unnecessary truncating and appending in the sequence of followers.

Head start leader election after reconfiguration. When a new configuration c_i starts after reconfiguration, a leader election will occur. The replicas of the continued processes will likely start faster than the new processes, as they already have the final sequence of c_{i-1} and thus do not need to fetch them. If the continued processes form a majority, c_i will start serving proposals. However, when a new process with a higher id eventually has the final sequence and starts its ballot leader election, it will overtake the leadership due to its higher id. A change of leader is expensive, as it includes a new prepare phase and possibly dropped proposals. Hence, a heuristic to prevent this is to give the leader of c_{i-1} a head start in the ballot leader election of c_i. The head start implies that the initial round of the ballot leader election is set to 1 instead of 0. The leader of c_{i-1} is likely to be the fastest to start in c among the continued processes because it decided the StopSign. Hence, it is likely to be part of the first majority of processes and become the leader, as its initial round is greater than its peers. When new processes with higher ids later join c_i, they will not overtake the leadership as their initial round is less than the current leader's round number in the ballot. This heuristic allows c_i to stabilize faster.

4.4.4 Kompact-Paxos

As previously mentioned, designing the library as a data structure does not take full advantage of the message-passing performance Kompact provides. Furthermore, the original pseudo code of Paxos did not include any batching. Hence, it was decided to also implement Paxos directly in the `PaxosReplica` component. This would allow tracking how each of the optimisations made for the library affected the performance. In this implementation, called *Kompact-Paxos*, there is no outgoing message queue. Instead, each outgoing message is directly triggered to the connected `Communicator`.

4.5 Client

The implementation of the client is central to the experiments, as it essentially forms the benchmarks. Two trivial arguments for the client component are `num_proposals` and `concurrent_proposals`. The former determines how many proposals the client will propose and wait to get responded before the experiment run is completed. The latter determines how many concurrent proposals are proposed at a time, without waiting for a response. This could be seen as how much load is put on the system. The more concurrent proposals, the more heavily loaded the system is.

Furthermore, the MPP runner includes two latches as arguments to the client component to control the execution flow of the benchmarks. The first latch is `leader_election_latch`. This latch signals when the client has acknowledged a successful initial leader election from the cluster. This allows MPP to only measure the execution time from the first proposal to the last response, with a leader already elected. The reason to exclude the first leader election from the execution time is to aid in finding the convergent point of the experiments. If the time to elect a first leader is a significant part of the execution time, one would need to have a long enough experiment, such that what is aimed to be measured becomes the dominant part of the execution time. However, one could argue that the leader election is an important part of the consensus algorithms and therefore should be included in the execution time. Indeed, the leader election is important but then it is preferred to have dedicated experiments to measure it. In this case, we consider the reconfiguration experiments as the main purpose for that.

The client proposes a value by sending the proposal to the last observed leader. Initially, that is the process elected from the first leader election. Later, each response contains the last observed leader from the responding process. The client will hence keep track of which process is the last observed leader and send proposals to it. Furthermore, a response also includes the proposed value. This allows the client to maintain a set of responded values for deduplication.

In both Raft and Paxos, proposals can be dropped. In a real system, the client would set a timer to retry the proposal if a response has not been received after a period of time. In unstable conditions, many retries could be needed before a response is received. To prevent benchmarks running infinitely or having results that differ significantly due to varying amounts of retries in different runs of the same experiment, the client instead treats a timed out proposal as succeeded. The timeout is set to a high value such that timeouts will result in notably higher execution time.

For reconfiguration experiments, the MPP runner will include the new configuration as an argument to the client component. The client will initially execute in the same way as in the normal experiments. Upon receiving `num_proposals`/2 responses, the client will propose the new configuration and then continue as normal. When the client eventually gets a response for the reconfiguration, the latest leader of the response will determine its next action. If there is a leader, the client simply continues with the remaining normal proposals. However, if there was no observed leader at the time the response of the reconfiguration was sent, the client will wait for a successful leader election. Upon getting the first leader of the new configuration, the client will reset all the timers of the pending proposals and retry them. We know for certain that the proposals issued after the reconfiguration proposal were dropped in Paxos, while it could possibly have been dropped in Raft if a change of leadership occurred.

4.5.1 Testing Correctness

For the experiment results to be credible, the algorithms and experiments must be implemented correctly. Thus, a test client that facilitates additional checks was implemented. The test client would perform an experiment in the same manner as the real client, but after receiving all its expected responses, it will query each process involved in the experiment (i.e. both added and removed processes in a reconfiguration experiment) for their sequence. By having the sequence of each process, we could ensure that no process had decided a value that was not proposed and at least a majority contained all the proposed and decided elements. To check that the client got all the proposed proposals is trivial, the experiment would not finish unless it had received all the expected responses.

The test client was implemented in MPP similarly to the real experiment client. It was used to test all the implementations that used MPP and Kompact, that is TiKV raft-rs, raw Paxos and Kompact-Paxos.

Chapter 5

Local Experiments

This chapter presents the results of benchmarks performed on a single host. All the experiments were run over the local loopback network device and therefore without any network latency.

First, the experimental setup is described followed by a comparison between LogCabin and our implementation of Raft using TiKV raft-rs. Furthermore, Kompact-Paxos and raw Paxos are compared before the various optimisation made for raw Paxos is evaluated. Lastly, the performance of our Raft and Paxos implementations are compared.

5.1 Experimental Setup

The implementations were tested by running MPP benchmarks in "fakeRemote" mode on a standalone server machine with 256 GB of memory and an Intel® Xeon® Gold 5218 CPU with 16 cores and 32 virtual threads. The host was running Ubuntu 20.04 LTS with a $5.4.0$ kernel for the x86_64 architecture.

Rust. All experiments were run using version 1.45.0-nightly (2020-05-31).

Kompact. A custom branch of Kompact was used for the experiments[2]. The default buffer size was increased to support large enough messages and Nagle's algorithm was turned off.

Parameters of Raft and Paxos. For the local experiments, the following parameters for the algorithms were used:

[2]Can be found at: `https://github.com/haraldng/kompact/tree/pr-65`

- Election timeout: In Raft, this parameter determines how long a process waits before starting a new election if no heartbeat message from the leader has been received. In Paxos, this corresponds to how long a ballot leader election component waits for heartbeat responses. Set to 5 s.
- Outgoing messages period: How often outgoing messages are flushed. Set to 1ms.
- Get decided elements period (Paxos only): How often the `PaxosReplica` component checks if there are any new decided elements in raw Paxos. Set to 1 ms.
- Tick period (Raft only): How often TiKV raft-rs increments its internal clock. Set to 100 ms.
- Leader heartbeat period (Raft only): How often the leader sends heartbeat messages to maintain as the leader. Set to 100 ms.
- Client proposal timeout: How long the client waits at most for a response. A high value is used to penalize dropped proposals. Set to 20 s.

5.2 LogCabin and TiKV raft-rs

A throughput experiment and a latency experiment with 3 processes were used to compare LogCabin and TiKV raft-rs. The latency experiment consisted of 100k proposal and the throughput experiment had a total of 1 million proposals with 500, 1k, 2k and 4k concurrent proposals. The number of concurrent proposals was set to fairly low values as the number of concurrent proposals corresponds to the number of client threads in the LogCabin experiments.

The results indicated that LogCabin outperforms TiKV raft-rs in terms of latency, with an average of 0.33 ms compared to 2 ms. This is expected as TiKV raft-rs only checks the outgoing messages and decided entries when the Kompact timer trigger every millisecond. LogCabin, on the other hand, directly issues a gRPC call without needing to wait periodically.

As seen in Figure 7, TiKV raft-rs significantly outperformed LogCabin in the throughput experiment. With 500 concurrent proposals, LogCabin recorded a throughput of 17318 operations/s while TiKV raft-rs recorded 116069 operations/s. Although LogCabin got a slight increase in throughput with 1k concurrent proposals, TiKV raft-rs got a much higher increase, resulting in

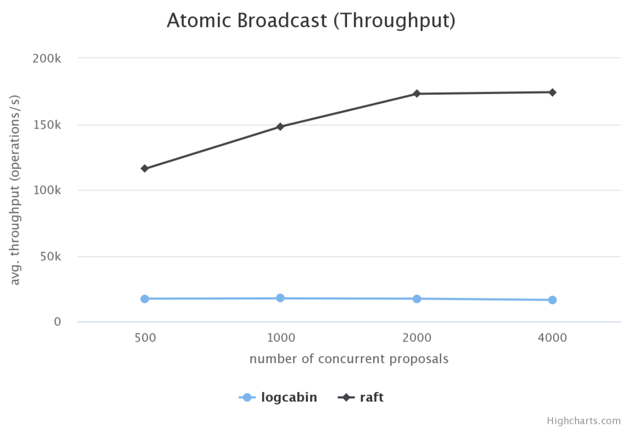


Figure 7: Average throughput of LogCabin and TiKV raft-rs using 3 processes.

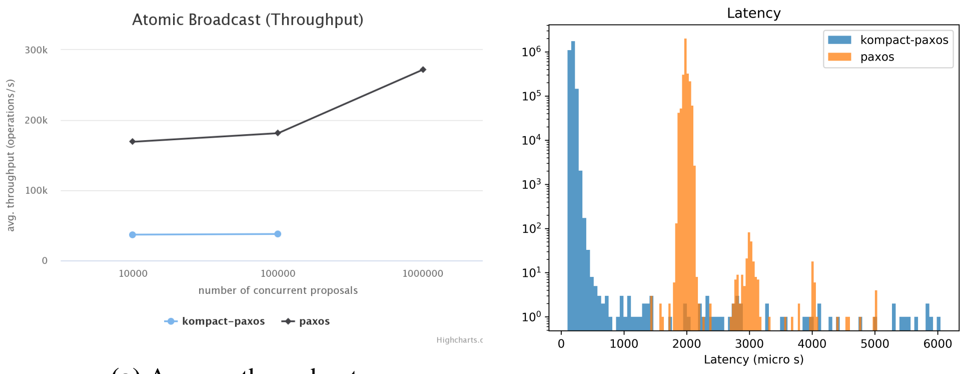
148146 operations/s compared to 17519 operations/s. When increasing the number of concurrent proposals to 2k and 4k, the performance of LogCabin decreased and even recorded lower throughput than the 500 concurrent proposals experiment. This is likely to be caused by context switching with many client threads. In contrast, TiKV raft-rs, got higher throughput as the number of concurrent proposals increased. With 4k concurrent proposals, TiKV raft-rs recorded a throughput of 174213 operations/s, which corresponds to an increase of 10.7x compared to LogCabin.

5.3 Kompact-Paxos and Raw Paxos

To compare Kompact-Paxos and raw Paxos, a normal throughput experiment and a latency experiment with 3 processes were conducted. The normal throughput consisted of 10 million proposals with 10k, 100k and 1 million concurrent proposals. The latency experiment was of 100k proposals.

As seen in Figure 8, raw Paxos has a much higher average throughput than Kompact-Paxos. By batching the Accept elements and only sending the latest decide, it resulted in at least 4.5x higher average throughput across all the different number of concurrent proposals. Furthermore, Kompact-Paxos could not finish the experiment with 1 million concurrent proposals in a reasonable time for 30 runs and was therefore aborted.

However, Kompact-Paxos is far superior in the latency test. Average latency of a proposal is 0.17 ms in comparison to 2 ms in raw Paxos. This is expected as Kompact-Paxos directly sends an outgoing message in Kompact, while raw Paxos has the overhead of the outgoing messages timer and the check



(a) Average throughput.

(b) Histogram of the latencies from all runs.

Figure 8: Comparison of Kompact-Paxos and raw Paxos using 3 processes.

decide timer that are both triggered periodically every millisecond.

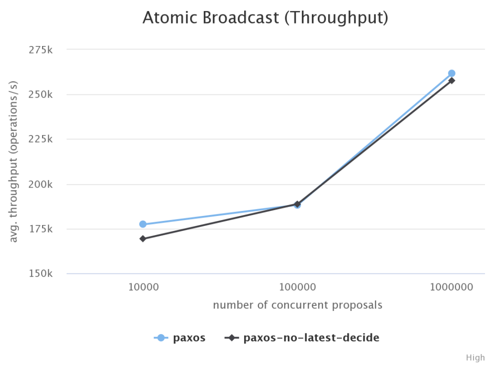
5.4 Optimisations of Raw Paxos

This section presents how the optimisations presented in Section 4.4.3 affected the performance of raw Paxos. The results of these experiments determined which optimisations were used later in distributed environments.

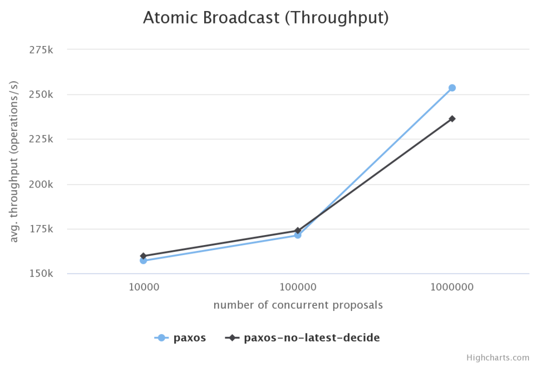
5.4.1 Latest Decide

The latest decide optimisation was tested by throughput experiments of 20 million proposals with 10k, 100k and 1 million concurrent proposals. Configuration sizes of 3 and 5 were used. As Figure 9 indicates, the optimisation does not have a big impact on performance when using 3 processes. When using 5 processes, the only noteworthy result is in the 1 million concurrent proposals experiment. The optimisation yielded a 7% increase in throughput with 253642 operations/s compared to 236457 operations/s in the unoptimised implementation.

The general impression of the results is that the optimisation did not have much effect unless the leader was very heavily-loaded as in the experiment with 5 processes and 1 million concurrent proposals. As the optimisation still showed some improvement, it was decided to be used for the remaining experiments.



(a) 3 processes.



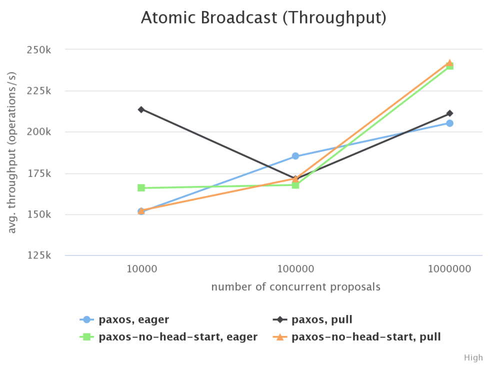
(b) 5 processes.

Figure 9: Average throughput of Paxos with and without the latest decide optimisation.

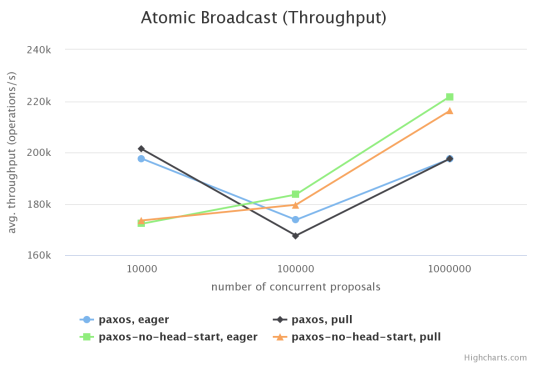
5.4.2 Max AcceptSync

To test the Max AcceptSync optimisation, a separate experiment that only measures the execution time of the prepare phase was used. The leader would measure the time from getting notified as the leader to the time a majority has accepted the initial AcceptSync. Each process starts the experiment with a sequence of 20 million elements and $ld = 10$ million. This implies that if the optimisation is not used, the leader will transfer 10 million elements to each follower. There were considerations to run experiments with various initial states of the followers, such as only a majority of the processes have the whole sequence. However, in such cases, the experiment would be dependent on the timing of which of the followers manage to promise while the leader is still in the prepare phase. It was thus decided to only test the extreme scenarios as that forms bounds for how the other scenarios would perform.

With 3 processes, the optimised version recorded an average execution time of 650 ms and 829 ms without the optimisation. Using 5 processes, the corresponding results were 1019 ms and 1308 ms. The optimised version yielded thus around 22% shorter execution time for the prepare phase in both configuration sizes. Intuitively, one might have expected the optimisation to have greater effect with an increasing number of processes. More followers imply that the leader has to send more of the long AcceptSync messages if the optimisation is not used. However, the execution is likely to be dominated by the time the leader waits to receive a majority of Promise and Accepted messages. This is not significantly affected by the optimisation where the followers only send outgoing messages every millisecond anyway.



(a) 3 processes.



(b) 5 processes.

Figure 10: Average throughput of Paxos reconfiguration to replace one process and 20 million proposals.

5.4.3 Head Start Leader Election

In Figure 10, the average throughput of a reconfiguration experiment, where one process is replaced and the client proposes 20 million proposals in total, can be seen. The relative performance between the implementations with and without the optimisation is similar in both the benchmarks with 3 and 5 processes. When the number of concurrent proposals is low, the implementation with the optimisation performs better than the implementations without the optimisation. This is mainly due to the unoptimised implementation dropping proposals and thus paying a penalty in execution time according to the client timeout. Without the optimisation, the new process with the highest process id is likely to start last in the new configuration and then overtake leadership which could result in dropped proposals. As seen in Table 5.1, the client recorded timeouts in at least $1/3$ of the runs in the 10k concurrent proposals benchmarks. There were no timeouts in the optimised implementation. This resulted in the optimised implementation outperforming the unoptimised implementation by up to 40% when using 3 processes and 17% when using 5 processes.

However, as the number of concurrent proposals is increased to 100k, the difference between the implementations is smaller, despite the unoptimised implementation still dropping proposals. When increased to 1 million concurrent proposals, the unoptimised version does not drop any proposals and even outperforms the optimised version. A reason for this could be that when increasing the number of concurrent proposals, it also increases the time needed for the leader to help the followers of the old configuration to reach the final se-

Table 5.1: Number of timed out proposals in reconfiguration experiments without the head start optimisation.

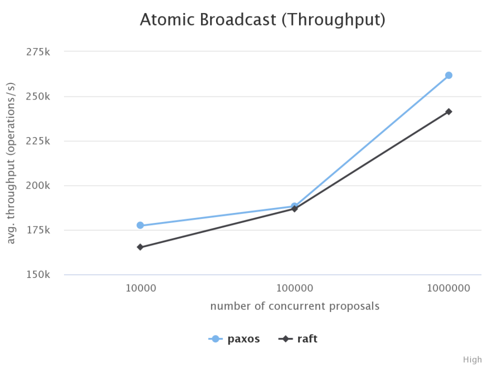
Impl.	Processes	Concurrent Proposals	Runs with Timeouts	Median	Average	Min	Max
Pull	3	10k	27/30	5999	5314	0	9757
Eager	3	10k	16/30	698	2029	0	7104
Pull	3	100k	30/30	42152	39911	5247	98219
Eager	3	100k	11/30	0	13378	0	61047
Pull	5	10k	30/30	5265	4737	301	8621
Eager	5	10k	11/30	0	1597	0	8732
Pull	5	100k	29/30	41851	42917	0	82693
Eager	5	100k	1/30	0	2817	0	84257

quence. And when the new configuration is started, it has to serve as the leader as well, in parallel. This causes the leader process to be heavily loaded and become a bottleneck. As one can see, the gap in relative performance between the implementations for 1 million concurrent proposals is slightly larger when having 3 processes compared to 5 processes, 15% compared to 12%. With 3 processes, the new configuration will become active faster as the majority can consist of the old leader and the new process. When having 5 processes, at least one of the followers from the old configuration must have reached the final sequence before the new configuration can become active. This implies that the old leader at least can help one of the followers reach the final sequence before having to be the leader as well in the new configuration.

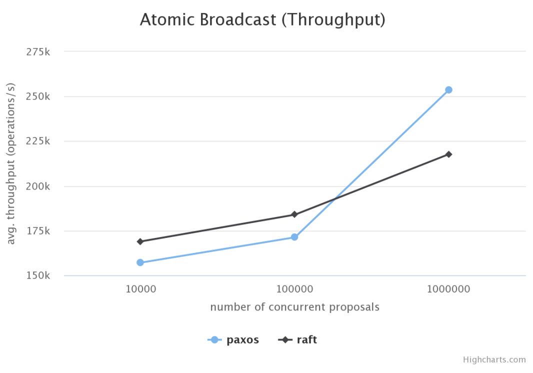
Despite the mixed results of the head start leader election optimisation, it was still decided to be used in the other experiments. The reason was to avoid getting results with client timeouts which would make the results more difficult to interpret. Furthermore, as no clear picture could be made of how the pull and eager approach of transferring the final sequence affects the performance, only the pull approach was decided to be used in the distributed experiments.

5.5 Comparison of Raft and Paxos

In this section, a comparison between our Raft and Paxos implementations in the “fakeRemote” environment is presented. The comparison was made in terms of throughput without reconfiguration, latency and throughput in the presence of reconfiguration.



(a) 3 processes.



(b) 5 processes.

Figure 11: Average throughput of Raft and Paxos.

5.5.1 Throughput

The throughput experiments consisted of 20 million proposals with 10k, 100k and 1 million concurrent proposals. As seen in Figure 11, Paxos outperforms Raft in general when using 3 processes. Paxos recorded an average throughput of 177310 operations/s while Raft got 165190 operations/s. When increasing the number of concurrent proposals to 100k, Raft manages to catch up with Paxos and there was no statistically significant difference between the algorithms. With 1 million concurrent proposals, Paxos had an average throughput of 261590 operations/s compared to 241415 operations/s in Raft, which corresponds to an increase of 8%.

In configurations with 5 processes, Raft outperformed Paxos when having 10k and 100k concurrent proposals with up to 8% higher average throughput. However, with 1 million concurrent proposals, Paxos had a 17% higher average throughput than Raft with 253642 operations/s compared to 217684 operations/s.

The results indicated that Paxos, in general, performs better than Raft with configuration size of 3. However, Raft outperformed Paxos by approximately 8% with 10k and 100k concurrent proposals when using 5 processes. A reason for this could be that Raft includes the commit index in the AppendRPC messages. In Paxos terms, this means that Raft includes the decide index in the Accept messages and hence does not send Decide messages like Paxos.

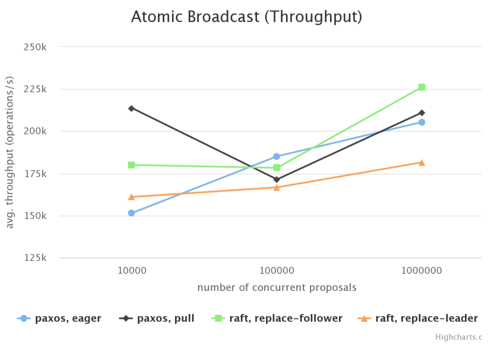
In both configuration sizes, Paxos scored a higher throughput in the experiments with 1 million concurrent proposals by 8% and 17% respectively. An explanation for this is the difference in how batching of elements is implemented in both algorithms. Raft uses a naive implementation for batching, where the whole outgoing message queue is iterated through to find an AppendRPC that has the same receiver and round to append the new element to. This implementation becomes inefficient when the outgoing message queue is filled with many messages, which becomes likely when the number of concurrent proposals is high. Paxos instead caches the index of the latest Accept message to each follower in the outgoing message queue. To batch an element, the leader checks if there is an Accept message in the outgoing message queue that is cached. If that is the case, the leader simply appends that element to the cached message. As accessing a specific index of a vector in Rust uses constant time, the batching in Paxos does not get slower with more messages in the outgoing message queue.

5.5.2 Reconfiguration

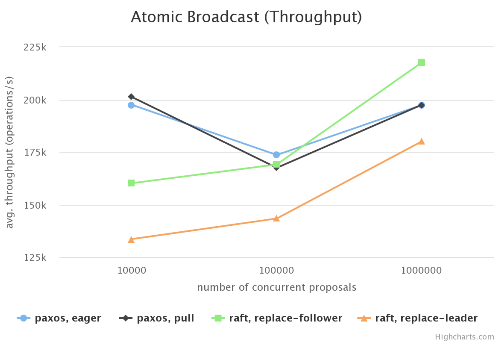
The results of the reconfiguration experiment can be found in Figure 12. As expected, replacing the leader in Raft results in worse performance compared to replacing a follower. Using 3 processes, the difference in average throughput varied from 7% and 25% using 100k and 1 million concurrent proposals respectively. A similar pattern was found using 5 processes, however, the differences were less varying with the number of concurrent proposals. The average throughput was 18% to 21% higher when replacing a follower.

Replacing the leader causes worse performance due to a period of downtime but also because the new leader has to retry replicating entries that the old leader did not manage to commit before the leadership changed. When the leader is changed, the entries that were in flight from the old leader will be discarded by slow followers, as they now have a more updated leader. However, before getting discarded, the messages still need to be handled by getting deserialised and sent between components in Kompact. This results in redundant and duplicated work, as the same entries will be replicated later by the new leader. This explains the reason for the difference in execution time becomes greater between Raft replace-follower and Raft replace-leader as the number of concurrent proposals is increased. With more concurrent proposals, there will be more entries in flight and therefore also result in more redundant and duplicated work.

Furthermore, entries behind the new configuration entry in the log of the old leader that did not get replicated to at least one follower, risk getting timed out by the client. Although the client retries proposals when it has realised a reconfiguration has been successful and the leader has changed, some of the proposals might already have timed out until the old leader has managed



(a) 3 processes.



(b) 5 processes.

Figure 12: Average throughput with reconfiguration.

Table 5.2: Number of timed out proposals per Raft replace-leader reconfiguration experiment.

Processes	Concurrent Proposals	Runs with Timeouts	Median	Average	Min	Max
3	10k	8/30	0	2217	0	10000
3	100k	23/30	91961	77720	0	338342
3	1 million	4/30	0	90957	0	944921
5	10k	13/30	0	3350	0	10000
5	100k	17/30	80741	51263	0	100000
5	1 million	2/30	0	51263	0	910156

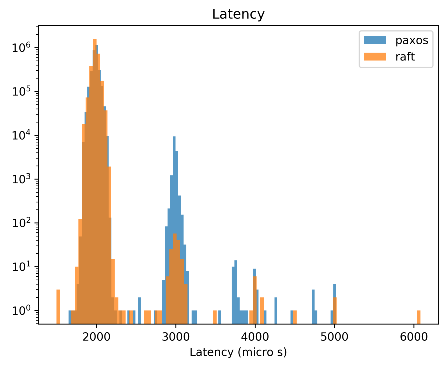
to commit the new configuration and the leader election has occurred. The number of dropped proposals per Raft replace-leader are shown in Table 5.2. A remarkable observation is that the number of experiment runs with timed out proposals increased with an increasing number of concurrent proposals, however, there appears to be a drop off when having 1 million concurrent proposals. A possible explanation is that with 1 million concurrent proposals, there will be more time for the followers and the new process to catch up the log. Because as soon as the joint-consensus configuration $c_{old,new}$ is added to the leader's log, it becomes the active configuration. The leader will start transferring elements to the new process and must get a majority of $c_{old,new}$ to commit entries. This allows the followers to be more up-to-date when c_{new} is committed and therefore result in less work for the new leader later, which lowers the risk of timing out proposals.

When comparing Raft to Paxos, it could be seen that the results were similar using both 3 and 5 processes. Paxos outperformed Raft replace-follower when using 10k concurrent proposals with up to 19% and 26% using 3 and 5 processes respectively. With 100k concurrent proposals, Paxos was still marginally outperforming Raft replace-follower with less than 4%. However, with 1 million concurrent proposals, replacing a follower in Raft performed better than Paxos but replacing a leader performed worse than Paxos. With 3 processes, replacing a follower yielded an average throughput of 226125 operations/s, which is a 7% increase of the 211036 operations/s recorded with Paxos-pull. The difference was even higher with 5 processes, with Raft replace-follower having 10% higher average throughput than Paxos-pull. Replacing the leader yielded a 16% and 10% lower average throughput than Paxos pull with 3 and 5 processes respectively.

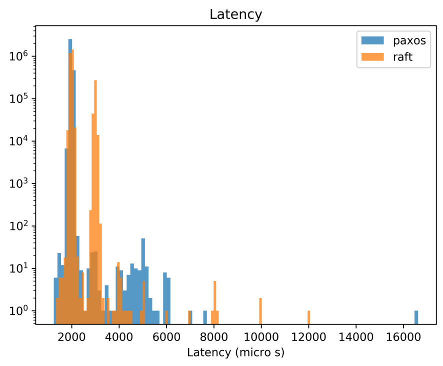
Paxos outperformed Raft replace-follower with a small number of concurrent proposals. The reasons are believed to be short down-time and more efficient transfer of the final sequence. As the number of concurrent proposals is small, followers could quickly reach the final sequence resulting in the new configuration being able to start faster. Furthermore, as the number of concurrent proposals is small, it is likely that even the followers that have not yet reached the final sequence can transfer early parts of the final sequence when getting queried by a new process in Paxos-pull. This allows for an efficient transfer of the final sequence in parallel. In Raft, a new process must catch up the log and it can only be transferred from the leader.

As previously described in Section 5.4.3, with a growing number of concurrent proposals and configuration sizes, Paxos performs worse as it takes longer for the new configuration to start and the leader gets heavily-loaded. In contrast, this is where Raft replace-follower excels, with no down-time and still being able to handle proposals during reconfiguration in the joint-consensus state.

Raft replace-leader performed worse than Paxos by between 10% and 51% across all experiments. As both include a leader election, the difference is likely caused by the less efficient transfer of elements to the new process and the duplicate replication the new leader has to perform. Furthermore, Paxos did not have timed out proposals as Raft replace-leader. The clear separation of replicas in different configurations of a Paxos component is advantageous. Even if a proposal is received after reaching the final sequence but before the new configuration is started, it can be held back until the new configuration start. Then, it could either propose or forward the held back proposals depending on if that process is elected as leader.



(a) 3 processes.



(b) 5 processes.

Figure 13: Histogram of latencies.

5.5.3 Latency

The latency experiment consisted of 100k proposals in total with the client proposing only 1 proposal at a time. The results from the experiment showed that the average and median latency of both algorithms were 2 ms using 3 processes. Using 5 processes, both algorithms recorded a median latency of 2 ms. However, as depicted in Figure 13, there were some extremely rare but noticeable high latencies for some proposals in both algorithms. The maximum latency recorded in Paxos was 16.6 ms and 12 ms in Raft. Raft had more of such spikes that resulted in a slightly higher average latency of 2.1 ms compared to 2 ms in Paxos. It is not clear why these rare spikes exist and further investigation is needed to determine the reason.

5.6 Summary

In this chapter, we have evaluated the performance of our Raft and Paxos implementations on a single host. The results showed that our Raft implementation has a higher latency than LogCabin, but outperforms it significantly in throughput. A similar pattern was found when comparing Kompact-Paxos with raw Paxos. Kompact Paxos had significantly lower latency but also much lower throughput.

The optimisations made in raw Paxos showed promising results with improvements both in terms of performance and less dropped proposals. With the Max AcceptSync optimisation, the prepare phase could be up to 22% lower if all followers already had the longest sequence. However, there were some mixed results for both the Latest Decide and Head Start Leader Election optimisations. The Latest Decide optimisation only showed significant improvement with 7% higher throughput when the leader was heavily loaded. The Head Start Leader Election optimisation behaved as expected with no dropped proposals. However, this proved to be a trade-off with throughput. When the number of concurrent proposals increased, it could cause the leader to become a bottleneck. Despite this, the optimisation was still decided to be used in the remaining experiments as dropped proposals could make results difficult to evaluate and interpret.

Lastly, our Raft and Paxos implementations were compared. Both algorithms recorded similar latency results. In the throughput experiment with 1 million concurrent proposals, Paxos outperformed Raft with 8% and 17% using 3 and 5 processes respectively. The reason is believed to be a more efficient batching implementation in Paxos. However, with 5 processes, Raft outperformed Paxos with a small number of concurrent proposals by around 8%. An explanation for this could be that Raft does not need to send explicit Decide messages that Paxos have to.

In the reconfiguration experiment, Raft showed a clear drop off in performance when replacing the leader instead of a follower. The difference in throughput could be more than 20% and there were also timed out proposals when replacing the leader. The main reasons for this are the down-time from leader election and the possible redundant and duplicated work which the new leader has to perform to complete the replication of entries that the removed leader did not manage to do before stepping down.

Paxos performed better than Raft when the number of concurrent proposals was small due to a short down-time and efficient transfer of the final sequence to new processes that could be made in parallel. In Raft, only the leader can transfer the sequence. However, Raft replace-follower outperformed Paxos as the number of concurrent proposals increased due to its ability to continue handling proposals during reconfiguration and no down-time. Furthermore, Raft replace-follower was outperformed by Paxos. The logically separated replicas in Paxos avoid the redundant replication work Raft possibly needs to do when a new leader is elected. Additionally, proposals can be held back and proposed later to avoid dropping proposals that arrive during the time the old replica has reached the final sequence and the new replica has not started yet.

The main insight from the local experiments is that although the algorithms appear similar in certain aspects, some implementation details could significantly affect the performance. This is highlighted by the Paxos implementation of batching and pulling parts of the final sequence in parallel from followers that might not have reached the final sequence yet.

Furthermore, the design of the Raft algorithm has some advantages and disadvantages over Paxos that also affect the implementation. Reconfiguration in Raft is handled very well when a follower is replaced, the procedure is essentially handled as normal entries. However, the idea of simplicity, where only the leader with the most up to date can be elected leader and therefore can entries only be transferred from leader to follower, really harms performance. It prevents a new process to catch up the log quicker by transferring entries in parallel.

Chapter 6

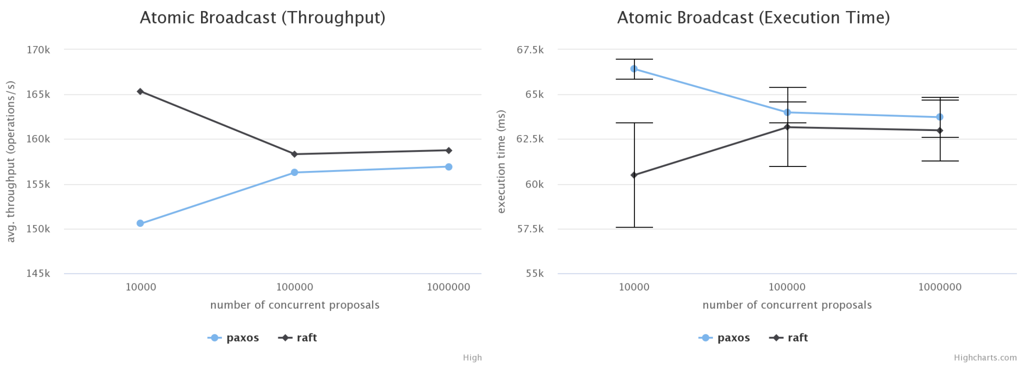
Distributed Experiments

This chapter presents the results of distributed experiments performed on Amazon Web Services using the Elastic Compute Cloud (EC2) service. Each process was started as an EC2 r5.xlarge instance with 4 vCPUs, 32 GB of memory and up to 10 Gbps networking performance.

To limit the execution time and cost of the benchmark, the number of runs per experiment was reduced to 10. Statistics from the raw results were therefore calculated by using student's t-distribution instead, as it is more suitable than normal distribution when the sample size is small. Confidence intervals of 95% were calculated from the results. In case of overlapping confidence intervals between two results, the confidence interval for the difference in mean of the execution times assuming unequal variances was calculated to determine if there is any statistically significant difference between them.

6.1 Single Data Centre

In this setting, all the processes are started in the same data centre. Two experiments with 3 processes were performed; a normal throughput experiment with 20 million and 10k, 100k and 1 million concurrent proposals and a latency experiment with 50k proposals. The purpose of only running these experiments was to get an impression of how the performance differs from the local setting with no latency and the geographically distributed environment with high latency.



(a) Average throughput.

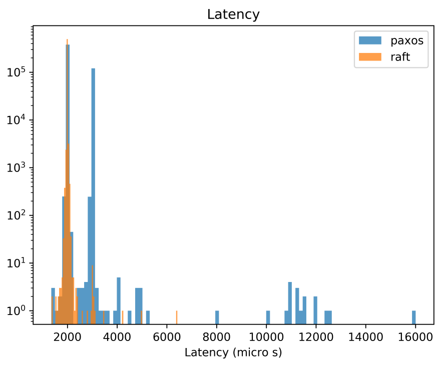
(b) Execution time with 95% confidence interval.

Figure 14: Throughput experiment in a single data centre.

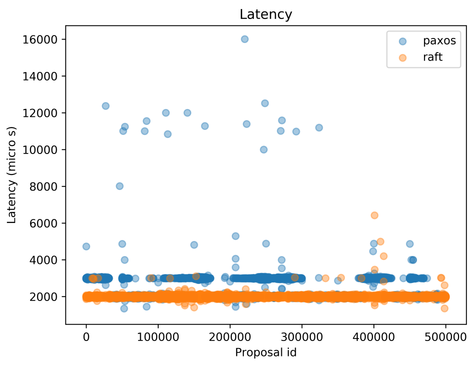
6.1.1 Throughput

The results from the throughput experiment can be found in Figure 14. With 10k concurrent proposals, Raft had around 10% higher average throughput than Paxos, with 165321 operations/s compared to 150555 operations/s. However, as the number of concurrent proposals increased, the difference in throughput was reduced. In both the 100k and 1 million concurrent proposals experiment, Raft had a very slight increase in throughput in comparison to Paxos of less than 1.5%. To determine if there was any statistically significant difference between the algorithms, the confidence interval of the difference in means was calculated. The confidence intervals were $(-1425, 3080)$ and $(-1166, 2635)$. This implies that the true difference in mean between the algorithms is 95% certain to be in the calculated confidence intervals. As the value 0 is included in both the confidence intervals, we conclude that the algorithms are statistically insignificant in the 100k and 1 million concurrent proposals experiments.

In comparison to the local throughput experiment, the increase inthroughput was much smaller when the number of concurrent proposals increased. The most reasonable explanation is that the throughput was limited by the network bandwidth. In the local experiments, there was no limit in the loopback interface. In this setting, however, there is a physical capacity limit on the network links, which for the used AWS EC2 instances were *up to* 10 Gbps. As no details could be found regarding minimum guaranteed bandwidth or fairness in sharing with other AWS users, it is assumed that the bandwidth is shared with other EC2 instances in the same data centre without any guarantees or fairness enforced. There are benchmarks of AWS EC2 instances that suggest



(a) Histogram of the latencies.



(b) Latency of every proposal across all runs.

Figure 15: Latency in a single data centre.

the actual bandwidth could be much lower [12]. Hence, when increasing the number of concurrent proposals, the network bandwidth became a bottleneck for the performance, resulting in a small increase in throughput from 100k to 1 million concurrent proposals.

6.1.2 Latency

Raft performed better than Paxos in the latency test. The average latency of a proposal was 2 ms in Raft and 2.2 ms in Paxos. More detailed results of the latency experiments are shown in Figure 15. Noteworthy is that the median value only differs by 1 microsecond between Paxos and Raft. However, Paxos has rare spikes of very high latency, up to 16 ms, which Raft does not have. There were similar findings in the local latency experiment. The reason for such spikes are unclear and more investigation would be needed to explain them.

6.2 Geographically Distributed

To test Raft and Paxos in a distributed environment with high latency, all processes and the client were started on EC2 instances located in data centres in different regions of the world. As seen in Figure 16, the client was located in Ireland and the consensus processes were located in Northern California, Cape Town and Sydney. The locations were chosen to have roughly equal latency from each other such that there would be high latency between every process and the client.

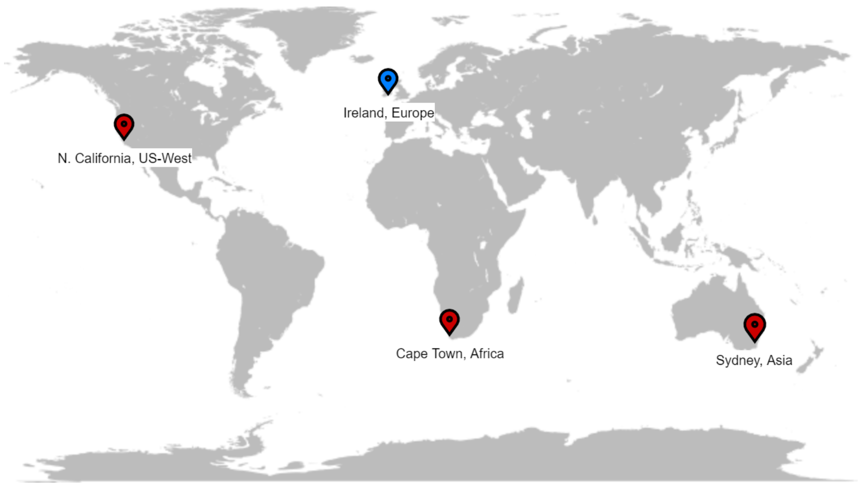


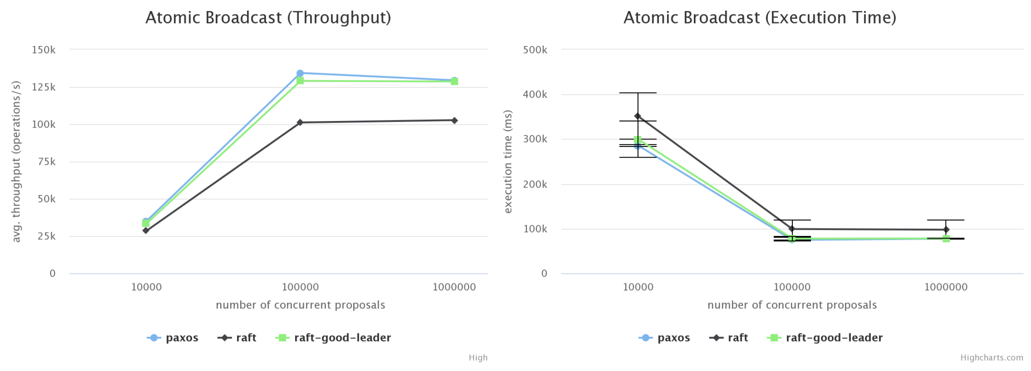
Figure 16: Locations of the AWS EC2 instances in the geographically distributed setting. The blue location represents the client and the red locations represent the consensus processes.

Similar experiments as the ones from the local benchmark were used. However, due to the significant increase in latency, the total number of proposals of the experiments were lowered to prevent very long execution times. In the throughput and reconfiguration experiments, the number of proposals was reduced to 10 million. The latency experiment was reduced to 1000 proposals.

6.2.1 Throughput

As found in Figure 17, Paxos outperforms Raft when using a small number of concurrent proposals. Paxos recorded up to 33% higher average throughput with 10k and 100k concurrent proposals. With 1 million concurrent proposals, Paxos continued to record higher average throughput by 129465 operations/s compared to Raft's 102719 operations/s. As the confidence intervals of the respective algorithms were overlapping, a 95% confidence interval for the difference in mean between Paxos and Raft was calculated. The resulting confidence interval $(-41752, 1527)$ suggests that the difference was not statistically significant in our experiments. However, as the absolute value of the lower limit is clearly greater than the upper limit, there are signs of Paxos would have had a statistically significant less mean if the sample size of the 1 million concurrent proposals experiment was larger.

Raft performs worse than Paxos in the 10k and 100k concurrent proposals due to its unpredictable leader election. The possibility of split votes and retrying election implies that the location of the leader is erratic. The location



(a) Average throughput.

(b) Execution time with 95% confidence intervals.

Figure 17: Throughput experiment in geographically distributed setting.

of the leader is important in this geographically distributed setting as the latency to the client will differ and thus affect performance. In contrast to Raft, Paxos is very predictable in terms of which process will become the leader. The process with the highest process id is most likely to become leader. Additionally, processes that start fast are also advantageous as they are more likely to have a higher round in their ballot than late processes when a majority has started. This naturally favours the process with the lowest latency to the client in MPP, as the experiment client is implemented in the benchmark master and the benchmark master itself starts the benchmark clients via ssh.

The location with the lowest latency to the client in Ireland was Northern California. In all the runs of the experiment, the leader of Paxos was located in Northern California. As seen in Table 6.1, The leader of Raft varied for different runs, which also explains the larger confidence intervals in execution time. In the 10k concurrent proposals experiment, the execution time of Raft differed by up to 35%. By extracting the results where Raft used the same leader as in Paxos, denoted as *raft-good-leader*, the average throughput between the algorithms were much smaller. Paxos still had a marginally higher observed average throughput but there was no statistically significant difference between them. However, one should take into consideration that the sample size of raft-good-leader is very small.

The width of the confidence intervals of the execution times in Raft with the unpredictable leader was noticeably reduced when going from 10k to 100k and 1 million concurrent proposals. This suggests that with a growing number of concurrent proposals, the location of the leader becomes less relevant. As more proposals will be in flight, the effect of latency is not as significant. Dur-

Table 6.1: The number of runs per location of the Raft leader in the geographically distributed throughput experiment.

	10k	**100k**	**1 million**
Cape Town	3	5	2
Sydney	1	3	3
N. California	6	2	5

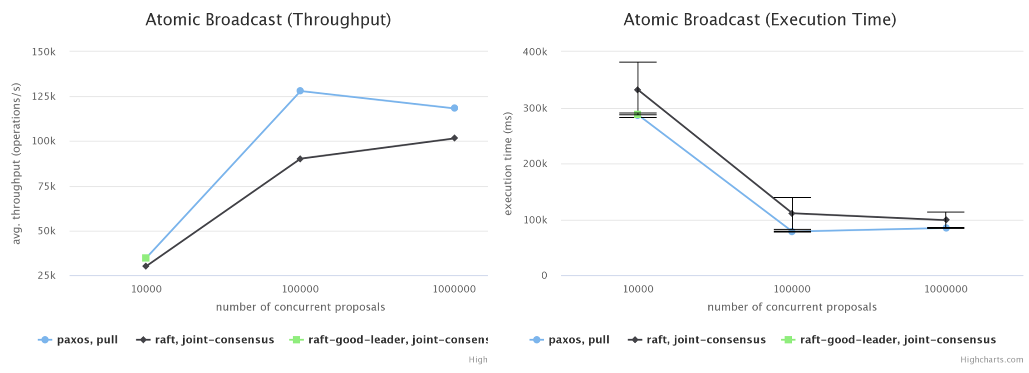
ing the time new proposals from client are sent to the leader, the raft instance is already busy with handling the preceding proposals. The same applies to responses sent from the leader to the client.

6.2.2 Reconfiguration

To test reconfiguration, an additional process was started in the data centre located in Sydney. The new process was set to replace the other process in the same data centre to maintain high latency between the processes. Due to the uncertainty of which process becomes the leader in Raft, the reconfiguration experiment could replace either the leader or a follower.

As seen in Figure 18, Raft is again affected by the latency of the elected leader, resulting in large confidence intervals with decreasing width as the number of concurrent proposals are increased. In the 10k and 1 million concurrent proposals experiment, the confidence intervals are overlapping and the difference in mean resulted in $(-94659, 5385)$ and $(-28955, 1051)$, suggesting that there was no statistically significant difference between the algorithms in our experiments. However, due to the significant larger absolute value of the lower limit in comparison to the upper limit, it appears likely that Paxos would have had a statistically significant shorter execution time than Raft if the sample size was larger.

With 100k concurrent proposals, there was statistically significant difference between the algorithms. Paxos outperformed Raft with an average throughput of 127992 operations/s compared to 90177 operations/s, which corresponds to an increase in 42%. Furthermore, Raft only managed to elect the process in Northern California as the leader in the 10k concurrent proposals experiment in 7 runs. As such, raft-good-leader had an observed average throughput of 5% less than Paxos with 33378 operations/s compared to 35024 operations/s. The calculated confidence interval for the difference in mean was $(-3422.705, -0.927)$. Hence, we can with 95% confidence conclude that



(a) Average throughput. **(b)** Execution time with 95% confidence intervals.

Figure 18: Reconfiguration experiment in the geographically distributed setting.

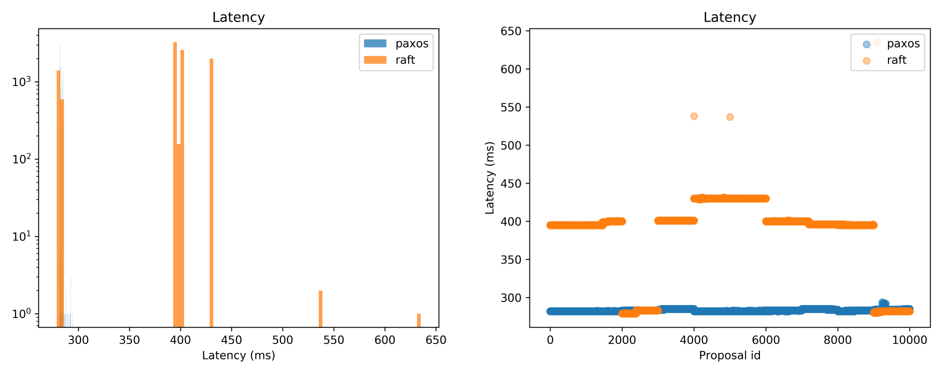
the difference in execution time is between 3422.705 and 0.927 ms shorter in Paxos.

6.2.3 Latency

The results of the latency experiment can be found in Figure 19. The experiment consisted of 1k proposals in total and Paxos recorded a median and an average latency of 283 ms, which were significantly less than the median of 396 ms and average of 381 ms recorded in Raft. As in the previous experiments, this is a result of the varying leader across different runs. By extracting the results from the runs where Raft has the same leader as Paxos, there was no statistically significant difference between the algorithms. The estimated average latency of Raft good-leader was 282 ms.

6.3 Trans-Pacific Data Centres

In the Trans-Pacific setting, instances were started in Northern California and Sydney. The intent was to have a configuration with a majority of processes (including the leader) that have short latency between each other, connected to a minority of followers with high latency. This would allow testing the algorithms in situations where the leader has to handle very slow followers periodically. Hence, in this experiment 5 processes were used, with 3 of them being in Northern California and the rest in Sydney. The client was also placed in Northern California. To force the leader of Paxos to be one of the processes in the majority, one of the processes in Northern California was hard coded

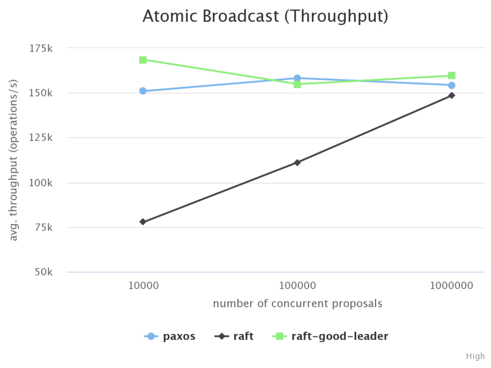


(a) Histogram of the latencies. **(b)** Latency of every proposal across all runs.

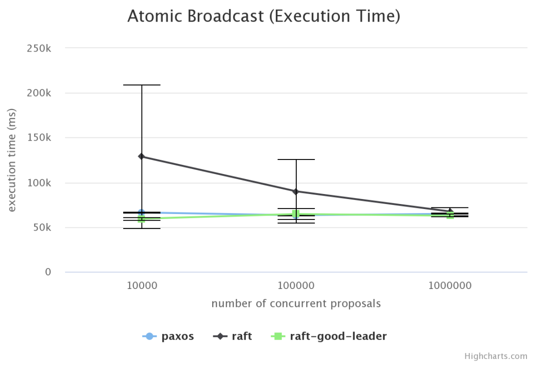
Figure 19: Latency in the geographically distributed setting.

to have $pid = 5$. The experiment consisted of 10 million proposals without reconfiguration.

As Figure 20 illustrates, the results of Raft can vary significantly depending on where the leader is located. Despite observing up to up to 94% higher average throughput for Paxos, there was no statistically significant difference between the algorithms. The confidence intervals for the difference in mean were $(-142325, 17641)$, $(-62391, 8930)$ and $(-6970, 1849)$ for 10k, 100k and 1 million concurrent proposals. However, for the runs where Raft had the leader in the intended data centre, some clearer result could be seen. In the 10k and 1 million concurrent proposals experiments, there was a statistically significant difference between raft-good-leader and Paxos, with raft-good-leader outperforming Paxos in average throughput by 12% and 3% respectively. The reasons for these results are not clear. A noteworthy observation is that despite the leader of raft being in the Northern California data centre in raft-good-leader, the performance varied depending on which AWS EC2 instance was the leader. As seen in Figure 21, this resulted in larger width of the confidence intervals in comparison to Paxos, where the leader was hard coded to the same AWS EC2 instance across all runs. This suggests that different AWS EC2 instance affected the performance, which could be due to the lack of bandwidth guarantees as described in Section 6.1.



(a) Average throughput.



(b) Execution time with 95% confidence intervals.

Figure 20: Throughput experiment in Trans-Pacific setting.

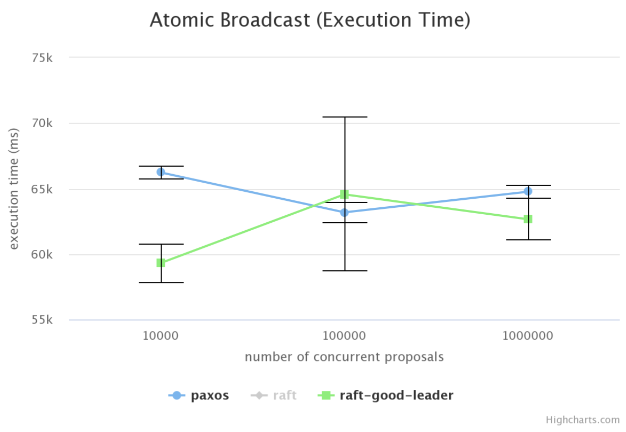


Figure 21: Execution time of paxos and raft-good-leader with 95% confidence intervals in the Trans-Pacific setting.

6.4 Summary

From the results of the single data centre benchmark, it could be seen that when the algorithms are deployed to real distributed environments, external factors that are not specifically related to the algorithm can have a significant impact on the performance. Both Raft and Paxos had similar performance but were limited by the network bandwidth.

When deployed in a geographically distributed setting, the main finding was that the leader election of the algorithms has an important role in where the leader is located, which itself has a major impact on both in terms of throughput and latency. Paxos has a very predictable leader election, which allows for more control in which process to become leader when deploying in practice. This resulted in better observed performance in all experiments, however, the performance were similar when the leader of both algorithms was in the same location. Hence, one should consider to start the processes in a desired order or have a custom implementation that allows for higher predictability of leader election in Raft when deploying in a geographically distributed environment.

A similar pattern was found in the Trans-Pacific setting. Paxos performs better due to its predictability in leader election. However, when the leader was located in the same data centre, the results suggest that Raft is slightly better than Paxos at handling a minority of very slow followers.

Chapter 7

Conclusion

In this work, a performance comparison between the distributed consensus algorithms Raft and Leader-based Sequence Paxos, a practical version of Multi-Paxos, has been made. Existing work focuses on comparing Raft and Multi-Paxos in theory or evaluating the performance based on simulations. To the best of our knowledge, there is no work that compares implementations of the algorithms in real-world scenarios. Prototype systems of the algorithms have therefore been developed and deployed in distributed settings to compare the performance of the algorithms in practice.

In Chapter 3, frameworks and libraries useful for building distributed systems were presented and their inclusion into the implementation was motivated. The Message-Passing Performance Suite (MPP) was decided to be used as the testing framework as it provided convenience for benchmarking distributed systems both on local single host machines and in real distributed settings. Kompact was chosen as the message-passing framework as it provided clear abstractions that facilitate the implementation of distributed systems. Furthermore, Kompact had shown good results when used to build similar systems in previous work.

LogCabin, the original Raft implementation, was considered inappropriate to be compared to our Leader-based Sequence Paxos implementation as the results would be misled by other parts of the system rather than represent the Raft algorithm. Hence, it was decided to implement Raft using TiKV raft-rs library and Kompact. This would allow for a fairer comparison with a Leader-based Sequence Paxos implementation that uses the same storage and network implementation.

Chapter 4 presented and motivated the experiments. The experiments were designed to measure latency and throughput with or without reconfiguration.

A single client was used to issue a certain number of concurrent proposals to produce different levels of workload for the system. Furthermore, a detailed description was given for the implementations of Raft and Leader-based Sequence Paxos in MPP and Kompact. Leader-based Sequence Paxos was designed to be similar to TiKV raft-rs, but also had some possible optimisations that were implemented and later tested.

In Chapter 5 and 6, the results of the local and distributed experiments were presented. The local throughput results indicated that implementation details could significantly affect the performance, despite the similarities of algorithms. This was emphasized by the throughput experiments under high workload and reconfiguration experiments with low workload, where more efficient batching of elements and pulling final sequence in parallel from slow followers in Leader-based Sequence Paxos yielded up to 17% and 26% higher average throughput than Raft. On the other hand, the Raft algorithm has both advantages and disadvantages that affect the implementation. A reconfiguration that replaces a follower is handled seamlessly and similarly to normal proposals. Raft recorded up to 10% higher average throughput than Leader-based Sequence Paxos when replacing a follower in the reconfiguration experiment. However, the motivation of simplicity by enforcing entries to only be transferred from leader to follower prevents efficient implementations for new processes to catch up the log in parallel after being added to a configuration.

The distributed experiments gave insight into how deploying in real-life environments affect performance. As the leader election is predictable in Leader-based Sequence Paxos, where the process with the highest process id is likeliest to become leader, it allows for more consistent performance. In contrast, the leader election in Raft is unpredictable due to random timeouts and split votes. This could affect the performance significantly if the elected leader is in a far away region with high latency. As a result, Raft recorded varying results in the experiments, with Leader-based Sequence Paxos outperforming Raft in average throughput by up to 33%. In the presence of reconfiguration, Leader-based Sequence Paxos recorded up to 42% higher average throughput. However, when the leader was in the same region, the performance in terms of latency and throughput of the algorithms was in general similar.

7.1 Future Work

There exist other aspects of Raft and Leader-based Sequence Paxos that have not been compared in this work. For practical systems, the sequence cannot be growing without bound. Hence, it would be useful to investigate in the performance of different snapshot methods. Moreover, a comparison of Byzantine fault-tolerant versions of the algorithms would also be interesting. By being tolerant of malicious behaviour, consensus could possibly be applied to blockchain technology.

There are also further improvements that could be made to the raw Paxos library:

- **Raw ballot leader election:** At the time of writing, the library relies on the user to provide the leader election abstraction. In our system, the ballot leader election was implemented directly in Kompact. However, a raw ballot leader election data structure could be implemented using a similar design to how TiKV raft-rs where calling a certain method is used to increment the logical clock.

- **Follower only sends the latest Accepted:** Reduce the number of messages sent from the followers to the leader by only sending the latest Accepted index.

- **Functional design to improve latency:** The library could be redesigned using a functional approach. Incoming messages or leader events could be handled as input to methods of raw Paxos that generate output in the form of outgoing messages. With this design, the periodic flushing of outgoing messages could be avoided while still maintaining freedom for the user to use their desired storage and network implementation.

Bibliography

[1] Cosmin Arad. “Programming Model and Protocols for Reconfigurable Distributed Systems”. Doctoral Thesis. Stockholm: KTH Royal Institute of Technology, 2013. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-24202 (visited on 06/13/2020).

[2] Cosmin Arad, Jim Dowling, and Seif Haridi. “Message-Passing Concurrency for Scalable, Stateful, Reconfigurable Middleware”. In: *Proceedings of the 13th International Middleware Conference*. Middleware ’12. Montreal, Quebec, Canada: Springer-Verlag, Dec. 2012, pp. 208–228. ISBN: 978-3-642-35169-3.

[3] *Atomix - A Reactive Java Framework for Building Fault-Tolerant Distributed Systems*. URL: https://atomix.io/ (visited on 06/07/2020).

[4] Jason Baker et al. “Megastore: Providing Scalable, Highly Available Storage for Interactive Services.” In: vol. 11. Jan. 2011, pp. 223–234.

[5] Mike Burrows. “The Chubby Lock Service for Loosely-Coupled Distributed Systems”. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. OSDI ’06. Seattle, Washington: USENIX Association, Nov. 2006, pp. 335–350. ISBN: 1-931971-47-1.

[6] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-15259-7 978-3-642-15260-3. DOI: 10.1007/978-3-642-15260-3. URL: http://link.springer.com/10.1007/978-3-642-15260-3 (visited on 05/29/2020).

[7] *Consul by HashiCorp*. URL: https://www.consul.io/docs/internals/consensus.html (visited on 06/07/2020).

[8] George Coulouris et al. *Distributed Systems: Concepts and Design*. 5th ed. Boston, Massachusetts: Addison-Wesley-Longman, May 2011. ISBN: 978-0-13-214301-1.

[9] Roberto De Prisco, Butler Lampson, and Nancy Lynch. “Revisiting the Paxos Algorithm”. In: *International Workshop on Distributed Algorithms*. Ed. by Marios Mavronicolas and Philippas Tsigas. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1997, pp. 111–125. ISBN: 978-3-540-69600-1. DOI: `10.1007/BFb0030679`.

[10] Victor Dubinin et al. “Implementation of Distributed Semaphores in IEC 61499 with Consensus Protocols”. In: *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. Porto: IEEE, July 2018, pp. 766–771. ISBN: 978-1-5386-4829-2. DOI: `10.1109/INDIN.2018.8472078`. URL: `https://ieeexplore.ieee.org/document/8472078/` (visited on 06/07/2020).

[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the Presence of Partial Synchrony”. In: *Journal of the ACM (JACM)* 35.2 (Apr. 1988), pp. 288–323. ISSN: 0004-5411, 1557-735X. DOI: `10.1145/42282.42283`. URL: `http://dl.acm.org/doi/10.1145/42282.42283` (visited on 05/25/2020).

[12] *EC2 Network Performance Cheat Sheet*. URL: `https://cloudonaut.io/ec2-network-performance-cheat-sheet/` (visited on 08/19/2020).

[13] *Etcd*. URL: `https://etcd.io/` (visited on 05/25/2020).

[14] *Frequently Asked Questions · The Rust Programming Language*. URL: `https://prev.rust-lang.org/en-US/faq.html#project` (visited on 06/10/2020).

[15] Seif Haridi, Lars Kroll, and Paris Carbone. “Lecture Notes on Leader-Based Sequence Paxos – An Understandable Sequence Consensus Algorithm”. In: (Aug. 31, 2020). arXiv: `2008.13456 [cs]`. URL: `http://arxiv.org/abs/2008.13456` (visited on 09/01/2020).

[16] C. Hewitt, P. Bishop, and R. Steiger. “Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute, 1973, p. 235.

[17] Heidi Howard and Richard Mortier. “Paxos vs Raft: Have We Reached Consensus on Distributed Consensus?” In: *Proceedings of the 7thWorkshop on Principles and Practice of Consistency for Distributed Data* (Apr. 2020), pp. 1–9. DOI: `10.1145/3380787.3393681`. URL: `http://arxiv.org/abs/2004.05074` (visited on 05/25/2020).

[18] Patrick Hunt et al. "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems". In: *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (June 2010), p. 14.

[19] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. "Zab: High-Performance Broadcast for Primary-Backup Systems". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. June 2011, pp. 245–256. DOI: `10.1109/DSN.2011.5958223`.

[20] *Kompics/Kompact*. June 2020. URL: `https://github.com/kompics/kompact` (visited on 06/13/2020).

[21] Lars Kroll. "Compile-Time Safety and Runtime Performance in Programming Frameworks for Distributed Systems". Stockholm: KTH Royal Institute of Technology, 2020. URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-267324` (visited on 06/13/2020).

[22] *KTH | ID2203*. URL: `https://www.kth.se/student/kurser/kurs/ID2203?l=en` (visited on 08/24/2020).

[23] *Kubernetes Components*. Nov. 2019. URL: `https://kubernetes.io/docs/concepts/overview/components/#etcd` (visited on 05/25/2020).

[24] Leslie Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (Dec. 2001), pp. 51–58. URL: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[25] Leslie Lamport. "The Part-Time Parliament". In: *ACM Transactions on Computer Systems* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: `10.1145/279227.279229`. URL: `https://doi.org/10.1145/279227.279229` (visited on 05/25/2020).

[26] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1978), p. 8.

[27] Leslie Lamport. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems". In: *ACM Transactions on Programming Languages and Systems* (Apr. 1984), pp. 254–280. URL: `https://www.microsoft.com/en-us/research/publication/using-time-instead-timeout-fault-tolerant-distributed-systems/`.

[28] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. “Reconfiguring a State Machine”. In: *ACM SIGACT News* 41.1 (Mar. 2010), pp. 63–73. ISSN: 0163-5700. DOI: `10.1145/1753171.1753191`. URL: `https://dl.acm.org/doi/10.1145/1753171.1753191` (visited on 08/24/2020).

[29] *Logcabin/Logcabin*. June 2020. URL: `https://github.com/logcabin/logcabin` (visited on 06/14/2020).

[30] Brian M. Oki and Barbara H. Liskov. “Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems”. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC ’88. Toronto, Ontario, Canada: Association for Computing Machinery, Jan. 1988, pp. 8–17. ISBN: 978-0-89791-277-8. DOI: `10.1145/62546.62549`. URL: `https://doi.org/10.1145/62546.62549` (visited on 06/07/2020).

[31] Diego Ongaro. “Consensus: Bridging Theory and Practice”. Doctoral Thesis. Stanford University, Aug. 2014. URL: `https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf`.

[32] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference*. Philadelphia, PA, 2014, p. 16. ISBN: 978-1-931971-10-2.

[33] *Raft - Rust*. URL: `https://docs.rs/raft/0.6.0-alpha/raft/index.html#arbitrary-membership-changes` (visited on 06/20/2020).

[34] *Raft::Storage::MemStorage - Rust*. URL: `https://docs.rs/raft/0.6.0-alpha/raft/storage/struct.MemStorage.html` (visited on 07/15/2020).

[35] *Raft::Storage::Storage - Rust*. URL: `https://docs.rs/raft/0.6.0-alpha/raft/storage/trait.Storage.html` (visited on 07/16/2020).

[36] Raghu Ramakrishnan et al. “Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics”. In: *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD ’17*. Chicago, Illinois, USA: ACM Press, 2017, pp. 51–63. ISBN: 978-1-4503-4197-4. DOI: `10.1145/3035918.3056100`. URL: `http://dl.acm.org/citation.cfm?doid=3035918.3056100` (visited on 06/07/2020).

[37] *Reliable Distributed Algorithms - Part 2*. URL: https://www.edx.org/course/reliable-distributed-algorithms-part-2 (visited on 08/24/2020).

[38] *Replication Layer | CockroachDB Docs*. URL: https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html (visited on 06/07/2020).

[39] Richard D. Schlichting and Fred B. Schneider. "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems". In: *ACM Transactions on Computer Systems (TOCS)* 1.3 (Aug. 1983), pp. 222–238. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/357369.357371. URL: http://dl.acm.org/doi/10.1145/357369.357371 (visited on 05/29/2020).

[40] Fred B. Schneider. "Byzantine Generals in Action: Implementing Fail-Stop Processors". In: *ACM Transactions on Computer Systems (TOCS)* 2.2 (May 1984), pp. 145–154. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/190.357399. URL: http://dl.acm.org/doi/10.1145/190.357399 (visited on 05/29/2020).

[41] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/98163.98167. URL: http://dl.acm.org/doi/10.1145/98163.98167 (visited on 05/26/2020).

[42] *The Slice Type - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch04-03-slices.html (visited on 07/16/2020).

[43] *TiKV*. URL: https://tikv.org/ (visited on 06/15/2020).

[44] *Tikv/Raft-Rs*. June 2020. URL: https://github.com/tikv/raft-rs (visited on 06/15/2020).

[45] *Tmpfs(5) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man5/tmpfs.5.html (visited on 08/13/2020).

[46] Abhishek Verma et al. "Large-Scale Cluster Management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*. Bordeaux, France: ACM Press, 2015, pp. 1–17. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964. URL: http://dl.acm.org/citation.cfm?doid=2741948.2741964 (visited on 06/07/2020).

[47] Zhaoguo Wang et al. “On the Parallels between Paxos and Raft, and How to Port Optimizations”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. Toronto ON Canada: ACM, July 2019, pp. 445–454. ISBN: 978-1-4503-6217-7. DOI: `10.1145/3293611.3331595`. URL: `https://dl.acm.org/doi/10.1145/3293611.3331595` (visited on 06/07/2020).

[48] Jianjun Zheng et al. “PaxosStore: High-Availability Storage Made Practical in WeChat”. In: *Proceedings of the VLDB Endowment* 10.12 (Aug. 2017), pp. 1730–1741. ISSN: 21508097. DOI: `10.14778/3137765.3137778`. URL: `http://dl.acm.org/citation.cfm?doid=3137765.3137778` (visited on 06/07/2020).

Appendix A

BLE Pseudo Code

Algorithm 3: Gossip Leader Election

```
Implements: Ballot Leader Election
Requires: Perfect Link
Algorithm:
1:  $\Pi$;                                   /* Process set */
2:  $round \leftarrow 0$ ;                   /* round number */
3:  $ballots \leftarrow \emptyset$;
4:  $n \leftarrow (0, pid)$ ;                /* ballot number */
5:  $L \leftarrow \bot$ ;                    /* leader */
6:  $n_{max} \leftarrow n$ ;                 /* largest ballot number seen */
7:  $d \leftarrow \Delta$ ;                  /* heartbeat delay */
8:  startTimer($d$);       /* schedule a timeout event in $d$
      timeunits */

9:  Fun checkLeader()
10:     $top = (topProcess, topN) \leftarrow$ maxByBallot($ballots \cup \{(self, n)\}$);
        if $topN < n_{max}$ then
            while $n \leq n_{max}$ do
11:             $n \leftarrow$ increment($n$);
12:         $L \leftarrow \bot$;
        else
            if $top \neq L$ then
13:             $n_{max} \leftarrow topN$;
14:             $L = top$;
15:             trigger $\langle$Leader $\mid topProcess, topN\rangle$;

16: Upon $\langle$Timeout$\rangle$
        if $|ballots| + 1 > \left\lceil \frac{|\Pi|+1}{2} \right\rceil$ then
17:         checkLeader();
18:     $ballots \leftarrow \emptyset$;
19:     $round \leftarrow round + 1$;
        foreach $p \in \Pi$ s.t. $p \neq self$ do
20:         send $\langle$HeartbeatRequest $\mid round, n_{max}\rangle$ to $p$;
21:     startTimer($d$);

22: Upon $\langle$HeartbeatRequest $\mid r, b_{max}\rangle$ from $p$
        if $b_{max} > n_{max}$ then
23:         $n_{max} \leftarrow b_{max}$;
24:     send $\langle$HeartbeatReply $\mid r, n\rangle$ to $p$;

25: Upon $\langle$HeartbeatReply $\mid r, b\rangle$ from $p$
        if $r = round$ then
26:         $ballots \leftarrow ballots \cup \{(p, b)\}$;
        else
27:         $d \leftarrow d + \Delta$ ;   /* Increase delay to make sure all
              replies from the current round are received
              within the time window. */
```

Appendix B

Paxos Pseudo Code

Algorithm 4: Leader-based Sequence Paxos – State&General

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

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

```
    /* General Code                                        */
17: Fun STOPPED()
        return LAST($v_a$) $= SS_i$;
18: Upon $\langle$LEADER $\mid L, b\rangle$
19:     $n \leftarrow (i, b)$;
        if $self = L \land n > n_L \land n > n_{prom}$ then
20:         $(n_L, n_{prom}) \leftarrow (n, n)$;
21:         $promises \leftarrow \{(rself, n_a, \text{SUFFIX}(v_a, l_d))\}$;
22:         $las \leftarrow [|\sigma_{i-1}|]^{|R|}$;
23:         $lds \leftarrow [\bot]^{|R|}$; $lds[rself] \leftarrow l_d$;
24:         $l_c \leftarrow |\sigma_{i-1}|$;
25:         $state \leftarrow (\text{LEADER}, \text{PREPARE})$;
            foreach $r \in R_o$ do
26:             send $\langle$PREPARE $\mid n_L, l_d, n_a\rangle$ to $r$;
        else
27:         $state \leftarrow (\text{FOLLOWER}, state.2)$;
```

Algorithm 4: Leader-based Sequence Paxos – Leader (1)

```
    /* Leader Code                                                  */
1:  Upon ⟨PROPOSE | $C$⟩ s.t. $state = (\text{LEADER}, \text{PREPARE})$
2:      $propCmds \leftarrow propCmds \cup \{C\}$;
3:  Upon ⟨PROPOSE | $C$⟩ s.t. $state = (\text{LEADER}, \text{ACCEPT}) \wedge \neg\text{STOPPED}()$
4:      $v_a \leftarrow v_a \oplus C$;
5:      $las[rself] \leftarrow |v_a|$;
        foreach $p \in \{r \in R_0 \mid lds[r] \neq \bot\}$ do
6:          send ⟨ACCEPT | $n_L, C$⟩ to $r$;
7:  Upon ⟨PROMISE | $n, n', suffix_a, ld_a$⟩ from $a$ s.t. $n = n_L \wedge state =$
    $(\text{LEADER}, \text{PREPARE})$
8:      $promises \leftarrow promises \cup \{(a, n', suffix_a)\}$;
9:      $lds[a] \leftarrow ld_a$;
10:     if $|promises| = \left\lceil \frac{|\Pi|+1}{2} \right\rceil$ then
11:         $suffix \leftarrow \text{MAXVALUE}(promises)$;   /* suffix with max $n$,
             longest if equal */
            /* adopt $v_d +\!+\, suffix$ and append commands          */
12:         $v_a \leftarrow \text{PREFIX}(v_a, l_d) +\!+\, suffix$;
            if $SS_i = \text{LAST}(v_a)$ then
13:             $propCmds \leftarrow \emptyset$;        /* commands will never be
                 decided */
            else
                if $SS_i \in propCmds$ then
                    /* Could also just drop other
                       outstanding commands instead of
                       ordering them before $SS_i$                  */
14:                 $v_a \leftarrow v_a \oplus C$ forall $C \in propCmds - \{SS_i\}$;
15:                 $v_a \leftarrow v_a \oplus SS_i$;
                else
16:                 $v_a \leftarrow v_a \oplus C$ forall $C \in propCmds$;
17:         $las[self] \leftarrow |v_a|$;
18:         $state \leftarrow (\text{LEADER}, \text{ACCEPT})$;
            foreach $r \in \{r \in R_0 \mid lds[r] \neq \bot\}$ do
19:             send ⟨ACCEPTSYNC | $n_L, \text{SUFFIX}(v_a, lds[r]), lds[r]$⟩ to $r$;
```

Algorithm 4: Leader-based Sequence Paxos – Leader (2)

```
    /* Leader Code (continued)                                    */
1:  Upon ⟨Promise | $n, n', suffix_a, ld_a$⟩ from $a$ s.t. $n = n_L \wedge state =$ (leader, accept)
2:      $lds[a] \leftarrow ld_a$;
3:      send ⟨AcceptSync | $n_L$, suffix($v_a$, $lds[a]$), $lds[a]$⟩ to $a$;
4:      if $l_c \neq |\sigma_{i-1}|$ then
5:          send ⟨Decide | $l_c, n_L$⟩ to $a$;   /* also inform what got decided already */
6:  Upon ⟨Accepted | $n, l_a$⟩ from $a$ s.t. $n = n_L \wedge state =$ (leader, accept)
7:      $las[a] \leftarrow l_a$;
8:      $M \leftarrow \{p \in \Pi \mid las[p] \neq \bot \wedge las[p] \geq l_a\}$;   /* support set for $l_a$ */
9:      if $l_a > l_c \wedge |M| \geq \left\lceil \frac{|\Pi|+1}{2} \right\rceil$ then
10:         $l_c \leftarrow l_a$;
            foreach $p \in \{p \in \Pi_0 \mid lds[p] \neq \bot\}$ do
11:             send ⟨Decide | $l_c, n_L$⟩ to $p$;   /* send length of chosen sequence */
```

Algorithm 4: Leader-based Sequence Paxos – Acceptor & Learner

```
/* Acceptor Code                                                    */
Upon ⟨PREPARE | $n, ld, na_L$⟩ from $p$
    if $n_{prom} < n$ then
        $n_{prom} \leftarrow n$;
        $state \leftarrow$ (FOLLOWER, PREPARE);
        $suffix \leftarrow$ if $n_a \geq na_L$ then SUFFIX$(v_a, ld)$ else $\langle\rangle$;
        send ⟨PROMISE | $n, n_a, suffix, l_d$⟩ to $p$;
Upon
 ⟨ACCEPTSYNC | $n, suffix, ld$⟩ from $p$ s.t. $state =$ (FOLLOWER, PREPARE)
    if $n_{prom} = n$ then
        $n_a \leftarrow n$;
        $v_a \leftarrow$ PREFIX$(v_a, ld)$ ++ $suffix$;
        $state \leftarrow$ (FOLLOWER, ACCEPT);
        send ⟨ACCEPTED | $n, |v_a|$⟩ to $p$;
Upon ⟨ACCEPT | $n, C$⟩ from $p$ s.t. $state =$ (FOLLOWER, ACCEPT)
    if $n_{prom} = n$ then
        $v_a \leftarrow v_a \oplus C$;
        send ⟨ACCEPTED | $n, |v_a|$⟩ to $p$;
--------------------------------------------------------------------
/* Learner Code                                                     */
Upon ⟨DECIDE | $l, n$⟩ s.t. $n = n_{prom}$
    while $l_d < l$ do
        $C \leftarrow v_a[l_d]$;          /* assuming 0-based indexing */
        trigger ⟨DECIDE | $C$⟩;
        $l_d \leftarrow l_d + 1$;
```