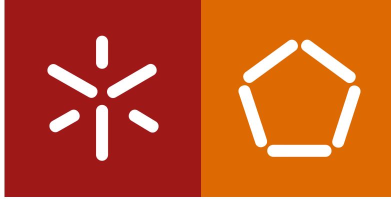
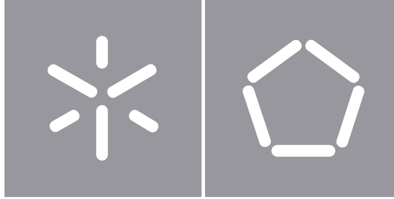


University of Minho
School of Engineering

André da Silva Gonçalves

Driftwood: Decentralized Raft Consensus

december 2023



University of Minho
School of Engineering

André da Silva Gonçalves

Driftwood: Decentralized Raft Consensus

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
José Orlando Roque Nascimento Pereira
Ana Luísa Parreira Nunes Alonso

december 2023

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:

CC BY-SA

https://creativecommons.org/licenses/by-sa/4.0/

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, december 2023

André da Silva Gonçalves

Abstract

The Raft consensus algorithm allows multiple state machine replicas to behave as a single and robust system, capable of tolerating various faults, offering more resilience when compared to a monolithic system. Raft is known for its ease of understanding and practical implementation, due to its utilization of strong leadership and division into three relatively independent sub-problems: log replication, leader election and safety. The combination of its simplicity, strong consistency, and fault tolerance guarantees in data replication has solidified Raft as one of the most popular algorithms since its creation a decade ago. It has been widely adopted in production by various systems, including Etcd, CockroachDB, MongoDB, and Neo4j.

The leader in a Raft cluster has the active role of replicating its log entries on a process-by-process basis. In contrast, the other processes play a passive role, waiting for the leader's requests and responding accordingly. This approach limits Raft in terms of scalability and performance: the larger number of processes in the system, the higher the leader's replication workload, causing the leader to become the system bottleneck. Therefore, the overall performance of a Raft cluster is heavily dependent on the efficiency of the leader process.

In terms of fault tolerance, a Raft cluster with $2f+1$ processes can tolerate f processes to crash, requiring only a majority of correct processes to ensure availability. Furthermore, Raft allows for messages to be delayed or lost and can tolerate total network partitions. However, it assumes that messages are eventually delivered, as the leader needs to maintain constant communication with all processes to retain its leadership status. If a process does not receive any communication from the leader within a specified timeout period, it will assume that the leader failed and initiate an election. Therefore, Raft needs a transitive network to operate efficiently. If there are processes that do not receive messages from the leader, it increases the likelihood of elections being triggered constantly, leading to a loss of system performance. A consequence of such an occurrence was the outage experienced by Cloudflare on November 2, 2020, which lasted for six and a half hours [30].

In this thesis, we present Driftwood, a novel algorithm that expands Raft by incorporating gossip mech-

anisms to decentralize the leader replication effort. Gossip protocols allow a process to deliver a message to all processes, even in non-transitive networks, with a high probability. The Driftwood's leader, instead of sending its log entries one-by-one, will initiate gossip rounds with a message containing uncommitted entries so that the processes propagate these entries among themselves and replicate the leader's log. Furthermore, this message serves as the leader's heartbeat when first received, avoiding unnecessary elections in non-transitive networks. However, the leader still has to receive replication confirmations from the processes in order to commit entries. So, we also propose new replicated data structures, shared in the gossip rounds, to discover new committed log entries.

Driftwood is experimentally evaluated and compared with Raft. To this end, we implemented three algorithms in the Paxi framework: Raft, which serves as the baseline comparison, and two versions of Driftwood, one that uses the new replicated data structures while the other doesn't. Through our evaluation, we determine the differences in performance, scalability, and distributed resource usage between Raft and Driftwood. We also analyse the behaviour of these algorithms in simulated non-transitive network scenarios.

Keywords Distributed consensus, Raft algorithm, Gossip

Resumo

O algoritmo de acordo Raft permite que múltiplas réplicas de uma máquina de estado se comportem como um único sistema robusto, capaz de tolerar diversas faltas, oferecendo mais resiliência comparado a um sistema monolítico. Raft é reconhecido pela sua facilidade de compreensão e implementação prática, devido ao uso de liderança forte e divisão em três sub-problemas relativamente independentes: replicação do registo, eleição do líder e segurança. A combinação de simplicidade, e garantias de coerência forte e tolerância a faltas na replicação de dados solidificou o Raft como um dos algoritmos mais populares desde a sua criação há uma década, sendo adotado em produção por diversos sistemas, incluindo Etcd, CockroachDB, MongoDB e Neo4j.

O líder dum sistema de Raft tem o papel ativo de replicar as entradas do seu registo processo a processo. Em contrapartida, os outros processos têm um papel passivo no sistema, esperando apenas pelos pedidos do líder e respondendo de acordo. Esta abordagem limita o Raft em termos de escalabilidade e desempenho. Com um maior número de processos no sistema, maior será a carga de trabalho de replicação para o líder, fazendo com que o líder se torne o gargalo do sistema. Assim, o desempenho dum sistema de Raft é dependente da eficiência do processo líder.

Em termos de tolerância a faltas, um sistema de Raft com $2f+1$ processos consegue tolerar f *crashes* de processos, necessitando apenas uma maioria de processos corretos para assegurar disponibilidade. Além disto, Raft permite perda ou atraso de entrega de mensagens e tolera partições totais de rede. Contudo, é assumido que mensagens são entregues inevitavelmente, uma vez que o líder tem que manter constante comunicação com todos os processos para manter a liderança. Se um processo não recebe nenhuma comunicação do líder até um tempo limite, este assume que o líder falhou e inicia uma eleição. Desta forma, Raft necessita de uma rede transitiva para operar eficientemente. Caso o líder não consiga comunicar com alguns processos, maior será a possibilidade que eleições aconteçam constantemente, resultando na perda de desempenho do sistema. Uma consequência desta ocorrência foi uma interrupção de serviços da Cloudfare a 2 de novembro de 2020 que durou seis horas e meia [30].

Nesta tese apresentamos Driftwood, um novo algoritmo que expande o Raft com a incorporação de

mecanismos de propagação epidémica para descentralizar o esforço da replicação pelo líder. Protocolos de propagação epidémica permitem que um processo entregue uma mensagem a todos os processos, mesmo em redes não transitivas, com elevada probabilidade. O líder, no Driftwood, em vez de enviar as entradas do seu registo um-a-um, vai iniciar rondas de propagação epidémica com uma mensagem contendo entradas não confirmadas para que os processos propagarem estas entradas entre si e replicarem o registo do líder. Além disso, esta mensagem serve como *heartbeat* do líder quando recebida pela primeira vez, evitando eleições desnecessárias em redes não transitivas. Contudo, o líder continua a ter que receber as confirmações de replicação dos processos para descobrir que entradas estão confirmadas. Assim, propomos novas estruturas de dados partilhadas na propagação epidémica que permitirão aos processos descobrir as entradas do seu registo que são confirmadas.

Driftwood é avaliado experimentalmente e comparado com Raft. Para tal, implementamos três algoritmos na *framework* Paxi: Raft que serve de base e duas versões de Driftwood, que diferem no uso de novas estruturas de dados para avanço das entradas confirmadas. Através da nossa avaliação pretendemos determinar a diferença de desempenho, escalabilidade e no uso de recursos distribuídos entre Raft e Driftwood. Analisamos também o comportamento destes algoritmos em cenários de rede não transitiva simulada.

Palavras-chave Acordo distribuído, Raft, Propagação epidémica

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Objectives 2

2 Background and Related Work **4**

2.1 State Machine Replication 4

2.2 Distributed Consensus 5

2.3 Raft Consensus Algorithm 6

2.3.1 System Assumptions 7

2.3.2 Overview 7

2.3.3 Leader Election 9

2.3.4 Log Replication 11

2.3.5 Limitations of Raft 13

2.4 Gossip-Based Protocols 15

2.5 Relevant algorithms 16

3 Driftwood **19**

3.1 Gossip of `AppendEntries` Requests 19

3.2 Adding Replicated Data Structures 22

3.2.1 Function `Update` 23

3.2.2 Function `Merge` 26

3.2.3 Overview of Version 2 of Driftwood 28

3.3 Correction Argument 29

4 Implementation **32**

4.1 Paxi Framework 32

4.2 Raft Implementation 35
4.2.1 Messages Module 35
4.2.2 Replica Module 36
4.3 Driftwood 37
4.3.1 Version 1: Addition of Gossip 38
4.3.2 Version 2: New Replicated Data Structures 38
4.4 Parameters 39

5 Experimental Evaluation **40**
5.1 Experimental Setup 40
5.2 Servers Parameters 42
5.3 Replication Scalability 43
5.4 Performance 45
5.5 Impact of Concurrent Clients 45
5.6 Use of Resources 46
5.7 Server Commitment Discovery 48
5.8 Performance with Partial Network Partitions 48

6 Conclusion **52**
6.1 Driftwood vs. Raft 52
6.2 Driftwood: Aspects to Improve 54
6.3 Final Remarks 56

A Details of Results **63**
A.1 Parameters 64
A.2 Replication Scalability 64
A.3 Performance 66
A.4 Impact of Concurrent Clients 67
A.5 Use of Resources 68
A.6 Performance in Network Partitions 70

List of Figures

1	Replicated state machine architecture with log approach	8
2	Transition relationship between Raft states	9
3	Example of a terms' view across time	9
4	Example of Raft logs	12
5	No stable leader	14
6	Comparison of Message Broadcasting Methods	15
7	Leader log replication in Raft and Driftwood	21
8	A condensed summary of the new mechanisms that make the first version of the Driftwood consensus algorithm an extension of Raft. Driftwood maintains most of the Raft logic, so we exclude the `RequestVote` RPC and the state variables already summarized in [32].	22
9	Examples of processes' states	25
10	Possible logs for a five-process cluster	26
11	Examples of the `Merge` function	28
12	A condensed summary of the new mechanisms that make the second version of Driftwood. We omit again the overlapping logic with Raft as well as the detailed explanations of the functions to update and merge the new replicated data structures, which are explained in 3.2.1 and 3.2.2, respectively.	29
13	Paxi modules [12]	33
14	Throughput (a) and Latency (b) with increasing number of replicas	44
15	Mean response time according to reached throughput	45
16	Throughput (a) and latency (b) with increasing number of concurrent clients	46
17	Use of CPU with increasing number of replicas	47
18	Use of CPU with increasing client request throttle	48

19 CDF of the time interval between a client request received by the leader and committed in leader/follower 49
20 Network topologies simulated 50
21 Performance of systems with 5, 53 and 125 replicas under varying link failures 51

List of Tables

1 Paxi benchmark parameters 34

2 CPU architecture information 41

3 Best-performing parameters for Version 1 42

4 Best-performing parameters for Version 2 43

Chapter 1

Introduction

Distributed systems are essential to society today, where areas like healthcare, transportation, finance and telecommunications rely on them to function efficiently and effectively. However, lots of things can go wrong in a distributed system: packets can be lost, reordered, duplicated, or arbitrarily delayed in the network and servers might pause or crash, and these issues can lead to data loss, services outage or decreased performance. Therefore, it is imperative to design and implement distributed systems that are robust and reliable, even in the presence of faults.

There are many algorithms and protocols for building fault-tolerant systems, with different assumptions and guarantees. Some of these protocols are based in the consensus abstraction, that is getting processes to agree on something, and is one of the most important and fundamental problems in distributed computing, that has been studied for decades.

One of the first and most important consensus algorithms is Paxos [27], proposed by Leslie Lamport in 1998, that allows consensus over a value under unreliable communication and process crash. In addition to Paxos, Leslie Lamport also proposes Multi-Paxos [27], a variation of Paxos that considers multiple concurrent runs of Paxos for reaching consensus on a sequence of values, that is commonly used to implement a replicated state machine. However, Paxos is known to be very hard to understand due to its non-intuitive approach and under-specification. This led to the development of the Raft [32] consensus algorithm by Diego Ongaro and John Ousterhout in 2014, which provides the same guarantees under the same assumptions as Paxos, but with the key goals being its ease of understanding and implementation. Raft rapidly became popular, and many systems today are divided between those which use Paxos [14, 20, 34, 38, 40, 41] and those which use Raft [3, 4, 5, 6, 8, 9, 11].

Raft is a consensus algorithm used for replicating (deterministic) state machines. State machine replication [36] is a popular replication technique that composes a set of unreliable hosts into a single reliable service that can provide strong consistency guarantees, including linearizability [24]. It requires that state machines replicated throughout various hosts are all initiated in the same state and in each of

them the same operations are applied in the same order, assuring all replicas states are, eventually, the same. Raft achieves this by managing a replicated log of operations in all servers.

1.1 Problem Statement

Raft is a leader-based protocol, where an elected leader process coordinates the other processes and communicates with clients. This approach limits the algorithm availability and scalability, as the leader performs more work than the other processes. With increasing number of processes or load to the system, the leader quickly reaches the limit of its resources. This happens because the leader is the one replicating its log to all processes. Whenever the leader receives a client request, it appends the request operation to its log as a new entry, sends the new entries to all processes and when it knows that a majority of processes have replicated the log correctly to a certain point, the leader applies to its state machine the new operations up to that point and replies properly to the respective clients.

On November 2nd 2020, Cloudfare experienced a failure for six minutes in a load-balanced switch, during which time their *etcd* cluster became unavailable [30]. This caused an outage of around six and a half hours. The problem in the *etcd* cluster was caused by a partial network partition in the three-node Raft cluster.

Raft relies on timeouts to detect leader crashes. However, in environments with highly variable network delays, processes might (falsely) believe that the leader has crashed due to transient network issues, and start a new leader election. These kinds of errors do not affect safety properties, however frequent leader election leads to bad performance because the system can end up more time choosing a leader than doing any useful work, resulting in loss of liveness [15].

Raft is particularly sensitive to network partitions. Leaders use heartbeats to maintain their leadership, however consistently unreliable network links between processes can cause situations where leadership continually bounces between processes, or the current leader is continually forced to resign, and therefore the system makes no or little progress.

1.2 Objectives

In this thesis, we propose the integration of new gossip-based mechanisms [31] into Raft, presenting a novel algorithm that we call Driftwood, capable of overcoming Raft limitations mentioned before. Our goal is to reduce centralization of load on the leader and enhance the resilience of the system to unreliable networks, more precisely being capable of tolerating network partitions.

We leverage gossip as the means to overcome network limitations and improve the use of distributed resources. Gossip protocols [31] are used in group communication due to high scalability and reliability properties. With gossip, processes communicate by forwarding messages to their neighbours. This way, processes can communicate even if they are not directly connected. Our approach consists in introducing on the leader periodic gossip rounds, sent in a permutation [33], for log replication and thus avoiding elections caused by network partitions. Additionally, we present novel concurrent data-structures shared through gossip, that allow any process to find out the next log index up to which point entries were correctly replicated to a majority of processes (i.e. committed) and may be applied.

We implement Raft and Driftwood on Paxi [10], an open-source framework for prototyping and evaluation of replication algorithms in Go [21]. The Raft implementation serves as a baseline comparison for Driftwood.

Chapter 2

Background and Related Work

This chapter presents and discusses the relevant background and related work. Section 2.1 introduces the state machine replication model, a method used to achieve consistent replicated states and fault-tolerance that requires coordination through the use of consensus algorithms. Section 2.2 describes the consensus problem in distributed systems and explains some approaches used to solve it. Section 2.3 presents the Raft consensus algorithm, an algorithm used to coordinate replicated state machines and created with the purpose of understandability. Section 2.4 explains how gossip-based protocols work. Section 2.5 summarizes some algorithms that have relevance to our work.

2.1 State Machine Replication

State replication is a method used in distributed systems for achieving fault tolerance. It involves maintaining consistent replicated states, allowing a fraction of the processes to fail. There are two fundamental approaches: primary-backup (or passive) replication [13] and state machine (or active) replication [26]. In the primary-backup replication model, one process is considered the primary that executes all operations issued by the clients and, regularly, pushes its state to the other processes considered the backups. Furthermore, in case the primary fails, a correct backup will take its role, ensuring availability. In the state machine replication model, clients operations are executed in all processes' (deterministic) states in a coordinated way.

State machine replication can be abstracted as a set of state machine replicas that receive operations from clients and through some coordination technique behave as a single service. A *state machine* consists of *state variables*, which encode its state, and *operations*, which transforms its state. A given operation applied to a given state is always deterministic and produces a new state and/or output, meaning that if any two states are the same and apply the same operation, then they will have the same result. Replicated state machines requires three properties to hold [37]:

1. **Initial state:** All correct processes are initiated with the same state;
2. **Determinism:** All correct processes receiving the same input on the same state produce the same output and resulting state;
3. **Coordination:** All correct processes apply the same sequence of operations.

Assuming a system where only crash-failures happen, we can easily understand how replicated state machines provide service, as long as at least one process is correct. Provided that each process starts in the same state and execute the same operations in the same order, then each will do the same and produce the same outputs; however, such a system is not realistic. In practice, assuring these three requirements is challenging. Processes can crash and later recover that might result in outdated states compared to the other processes. Moreover, determinism can introduce some constraints in the system, as it complicates the use of paradigms such as concurrency and parallelism.

Coordination is the fundamental problem in replicated state machines. To resolve it is necessary to implement *total order broadcast* and/or a *consensus algorithm* [22]. Replicated state machines must reach *consensus* about the contents and ordering of operations.

2.2 Distributed Consensus

Distributed consensus is the problem of achieving agreement in asynchronous distributed systems. Traditional consensus requires that a set of processes agree on a value or decision, and must hold the following safety properties [29]:

1. **Non-triviality:** A value can only be chosen if it was proposed by a correct process;
2. **Stability:** Once a value has been chosen, no process can revert its decision;
3. **Consistency:** All correct processes decide the same value.

An additional liveness property is usually considered, denominated the **Termination** property, which states that eventually all correct processes decide on a value.

There are many variations of the consensus problem depending on the system and failure models. Most real systems need to make a series of decisions over time, instead of making a single decision once. A common solution to solve this problem is to append each decisions' results to a replicated log in an orderly way. Each process has a copy of the log, and log entries are appended at each process according to a consensus algorithm that ensures eventual consistency between the replicated logs. Most consensus

algorithms keep track of a log's index such that any entry before that index has been agreed upon, allowing the replicated logs' entries to differ past that index, which might be agreed upon or discarded in the future.

Many consensus algorithms have been proposed in the literature; one of the first and most influential is Paxos [27]. Usually, consensus algorithms can be divided in two types

- **Symmetric (leader-less):** All processes have equal roles, and clients can send requests to any process;
- **Asymmetric (leader-based):** At least one process is a leader, making all decisions on behalf of the system.

The leader-based approach helps processes reach agreement by letting the leader decide, that is usually faster compared to the leader-less approach. However, in the presence of faults, the leader can become unreachable to the system. Therefore a fault detector and a mechanism to replace the leader become necessary, which impacts performance.

A key idea in distributed consensus is *voting* to reach a decision. The system reaches a decision when a majority of processes vote for it. If more than one value is proposed, a majority of processes is required to vote for one value for a decision. This allows some processes to fail, as it is only necessary for a majority of correct processes to reach agreement.

2.3 Raft Consensus Algorithm

Raft [32] is a leader-based consensus algorithm that achieves strong consistency on replicated state-machines by managing a replicated log of operations.

As explained in Section 2.1, replicated state machines need to apply the same operation in the same order to be consistent. In Raft, an elected *leader* appends clients operation requests to its log, in order, and replicates its log to the other processes, *followers*. When the leader knows a majority of followers have copied the log up to a certain index, the leader can then commit the uncommitted operations up to that index and respond to the respective clients. The leader informs the followers of the last committed index so that they can also progress. Raft ensures that in case of leader failure, only a process with an up-to-date log, i.e. that as copied the previous leader log to the point of the that leader's committed index, can become leader. Therefore committed entries are never deleted.

Raft is equivalent in performance and fault-tolerance to (Multi-)Paxos [28], one of the most used and studied consensus algorithms, but Raft improves it in terms of understandability. Because of that, Raft is

becoming the preferred consensus algorithm to be taught by distributed systems related lecturers. This also led to many real-world distributed systems using it such as etcd [6], CockroachDB [1], TiDB [2] and more.

2.3.1 System Assumptions

Raft operates under the following assumptions:

1. the cluster of processes is an asynchronous distributed system, i.e., no upper bound in computation and message delivery as well as no global clock available;
2. network communication has to be fully connected, but can be unreliable, allowing network delays, total partitions, as well as packet loss, duplication, and re-ordering;
3. Byzantine failures will not be tolerated;
4. operations requests are sent to the leader only (clients know which process is the leader);
5. the protocol has access to infinitely large, monotonically increasing values;
6. the state machines running on each process are deterministic and all start in the same state;
7. processes have access to infinite persistent storage that cannot be corrupted, and any write to persistent storage will be completed before crashing (i.e. using write-ahead logging); and
8. processes are statically configured with a knowledge of all other process in the cluster.

Some of these points can be relaxed by extending Raft.

2.3.2 Overview

Raft and many others consensus algorithms usually describe the system as shown in Figure 1. In each replicated process exists a clear distinction between the *state machine*, the *replicated log* and the *consensus module*.

The consensus module is handled by the consensus algorithm, in our case Raft. It receives operations from clients and messages from other processes and makes all decisions, such as, appending and deleting entries from the log or when to apply operations to the state machine.

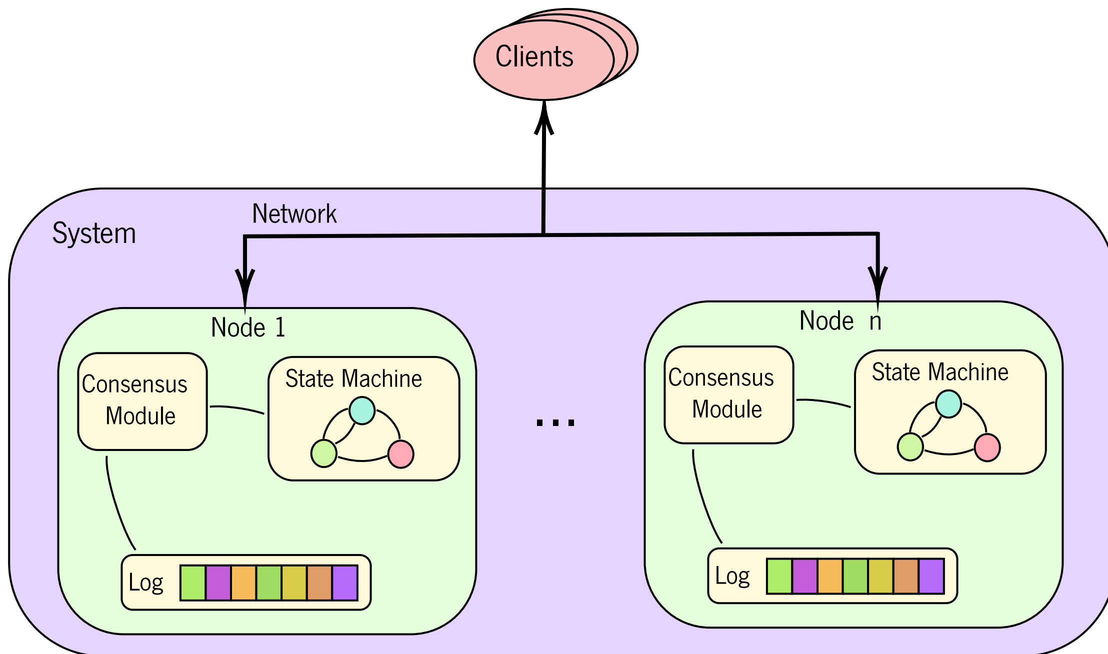


Figure 1: Replicated state machine architecture with log approach

In a Raft cluster each process, more precisely the process's consensus module, can be in one of the following three states, each with its role:

- **Follower**: all processes start as followers; passively receives requests from the leader and replies accordingly, never initiating communication.
- **Candidate**: intermediate state between follower and leader; actively requests votes to become leader.
- **Leader**: responsible for log replication and communicating with clients; no more than one leader exists at the same term.

Processes transition between states in accordance with certain events, shown in Figure 2. These events are either temporal, such as timeouts happening; or spatial, such as receiving RPCs.

Time in Raft is logically divided into **terms**. A term is a monotonically increasing natural number, that acts as a logical clock to achieve global partial ordering of events. A process's term is only updated when it starts (or restarts) an election, or when it learns from another node that its term is out of date, i.e., is smaller. All messages exchanged between processes include the senders' term. The receiver will compare its term with the message's term. If its term is larger, a negative reply is sent, otherwise its term, and possibly state, will be updated. In each term, only one leader can be elected, and it serves as a leader until the end of the term. An example of a process's vision of terms across time can be visualized in Figure 3.

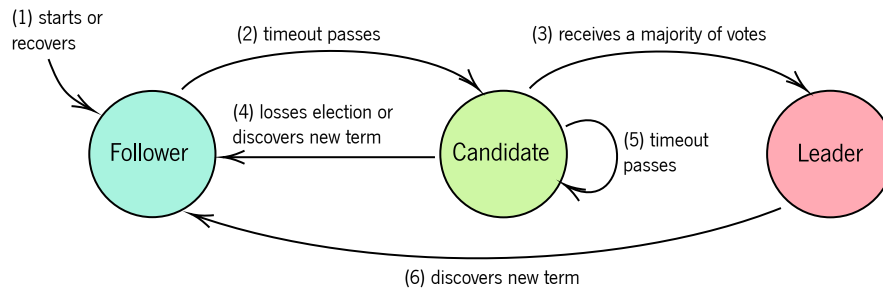


Figure 2: Transition relationship between Raft states

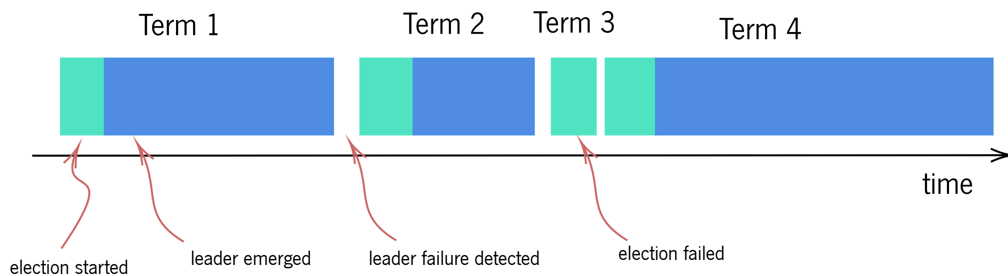


Figure 3: Example of a terms' view across time

Communication between processes is done through remote procedure calls (RPCs). Raft requires only two types of RPCs:

- `RequestVote` RPCs: initiated by candidates during elections; and
- `AppendEntries` RPCs: initiated by the leader to replicate its log and provide a form of heartbeat.

The leader, in order to replicate its log, sends `AppendEntries` carrying log entries for followers to copy. Additionally, the leader sends `AppendEntries`, without log entries, periodically to followers as a form of heartbeat. If a follower doesn't receive an `AppendEntries` RPC during a period of time, called `Election Timeout`, it assumes there is no viable leader, so it transitions to candidate and starts an election.

For simplicity, the Raft algorithm is divided in three relatively independent sub-problems: leader election, log replication and safety. Candidates send `RequestVote` RPCs to gather votes from other processes. In the Section 2.3.3 we further describe the leader election process.

2.3.3 Leader Election

Leader election is triggered by a follower, when a period of time (`Election Timeout`) elapses without receiving any `AppendEntries`. Leaders send periodic heartbeats (`AppendEntries` that carry no log

entries) to all followers to maintain their leadership. Evidently, if the leader crashes it will stop emitting RPCs to followers, and a new election will be eventually started.

On startup, all processes start as followers. When a follower's `Election Timeout` elapses, it transitions to the candidate state and emits *RequestVote* RPCs to all other processes to acquire votes in order to become leader. A candidate stays in that state until one of the following happens:

a) the candidate wins the election;

b) another process establishes itself as leader; or

c) a period of time passes without a leader appearing.

A candidate wins an election if it receives votes from a majority of processes with the same term as the candidate (candidates vote for themselves). Each process votes once in a given term, for the first candidate with a log up to date whose vote request was received first (explained in Section 2.3.4). The majority rule ensures only one leader can be elected at any term. Once a candidate wins an election, it transitions to the leader state and starts sending heartbeats to advertise its election victory and maintain leadership.

In case the candidate receives an `AppendEntries` from another process claiming to be the leader, if the leader's term (included in the request) is equal or larger than candidate's term, then if recognizes the leader as legitimate and transitions to follower, else if the leader's term is smaller than the candidate's, it replies that its term is higher and continues the election.

During a candidate's election, it's possible that a leader is not elected, when votes are split between many candidates. The candidate assumes this happened when the `election timeout` elapses again. When this happens, the candidate will start a new election by incrementing its term and emits another round of `RequestVote` RPCs.

A *RequestVote* message includes the following parameters:

- **`Term`**: candidate's current term, i.e. election term;
- **`LastLogIndex`**: index of candidate's last log entry; and
- **`LastLogTerm`**: term of candidate's last log entry.

Raft uses randomization to avoid split votes to repeat indefinitely. Each process chooses an *election timeout* randomly from a fixed interval (for example 150 to 300 ms for a broadcast time of 15 ms) every time it starts an election. Therefore, in most cases only a single process will time out, win the election

and start sending heartbeats before the others processes time out. If a candidate starts an election but can't become leader, another random process will, eventually, time out and start a new election.

Followers and candidates, if the `Election Timeout` elapses become/remain candidates and start election by doing the following:

1. Increment their current term;
2. Vote for itself;
3. Emit `RequestVote` RPCs to all other processes.

Processes, upon receiving a `RequestVote` message, reply with their current term and whether they grant their vote to the candidate. A process grants its vote if its current term is smaller or equal to `RequestVote's Term`, it hasn't voted in anyone yet in the `RequestVote`'s Term and candidate's log is up-to-date with its log. In the following section we explain Raft's log properties and how it's replicated by the leader.

2.3.4 Log Replication

Once a process is elected leader, as well as sending heartbeats, it starts receiving requests from clients and replicating its log. Clients send operations requests to be applied to the replicated state machine. On reception, the leader appends the operation as a new entry to its log. Then it issues `AppendEntries` RPCs to the followers to replicate the newly appended entry. Once the leader knows that the entry has been safely replicated to a majority of processes, the leader applies the entry's operation to its state machine and responds the execution result to the respective client. The leader ensures its log is replicated by sending `AppendEntries` indefinitely until the log is fully replicated to the respective follower.

Each log entry stores one client operation and term of reception. The entry's log index and stored term are necessary to detect inconsistencies between logs. Log entries are called `committed` when the leader decides they are safe to apply to the state machines. Figure 4 shows an example of logs states in a raft cluster with 5 processes, where operations are *writes* to a key-value store state machine, e.g. in the entry in index 4 of leader's log has associated the operation $x \leftarrow 5$ (assign 5 to x) and was appended to the log by the leader of term 2.

Followers keep a local monotonically increasing value, the `CommitIndex`, that is the index of the last entry committed. Leader informs other processes through `AppendEntries` of the last index that can

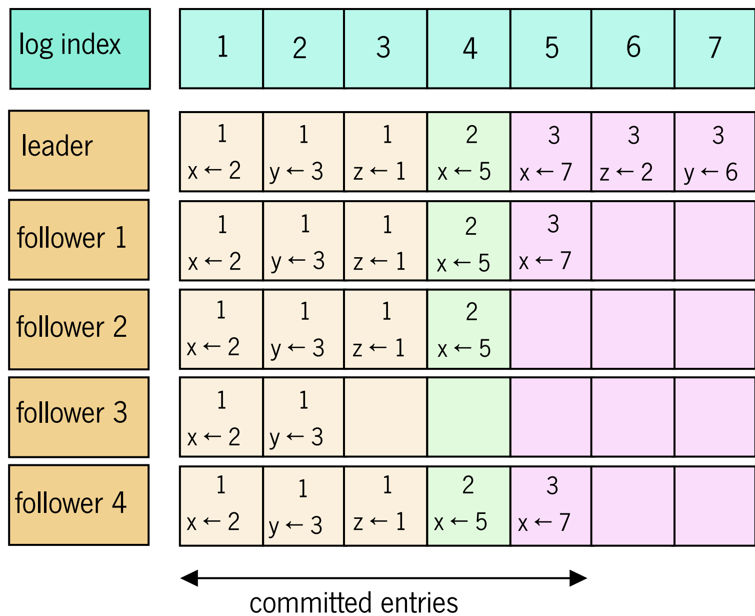


Figure 4: Example of Raft logs

be committed, i.e. leader's `CommitIndex`. When a follower receives a leader's `CommitIndex` higher than its own, the follower updates it and can therefore apply the uncommitted entries in order up until the updated `CommitIndex`.

Raft ensures that if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index, known as the **Log Matching Property**.

An `AppendEntries` request includes the following parameters:

- **`Term`**: the term of the current leader (i.e. sender's term);
- **`PreviousLogIndex`**: the index of the log entry preceding new ones (i.e. entries sent to be replicated);
- **`PreviousLogTerm`**: the term of the log entry preceding new ones;
- **`LogEntries`**: partition of leader's log sent to be replicated; and
- **`LeaderCommit`**: leader's entry index of last committed entry (i.e. leader's `CommitIndex`).

Each process answers an `AppendEntries` request with its term and if its log matches the leader, i.e. the term of the entry with index `PrevLogIndex` in its log is equal to `PrevLogTerm` (Log Matching Property). Once a majority of processes answers with success, the leader can commit the new entries.

Followers upon receiving an `AppendEntries` message do the following:

1. If follower's current term is higher than `AppendEntries` term, returns `false`.

2. If logs don't match up until `PrevLogIndex`, returns `false`.

3. If there are any entries after `PrevLogIndex` then they are deleted.

4. Append `LogEntries` to its log.

5. If the follower's current `CommitIndex` is smaller than `LeaderCommit`, then it updates its commit index to the minimum of `LeaderCommit` and index of last entry in log; then it can apply, in order, the entries with an index larger than the last `CommitIndex` and smaller than or equal to the current `CommitIndex`.

6. Return `true`.

Leaders, upon receiving an `AppendEntries` reply message must consider the following two cases:

a) If the reply's term is higher than its own, it realizes it's no longer the leader and becomes a follower;

b) Else if it receives false, then the sender's log is outdated and needs updating, but the entries previously sent aren't enough, so it re-sends an `AppendEntries` request to that process, with log entries starting from an earlier point in the log. This way, it may get false several times, until it reaches a point where leader's log and that process' log match.

Raft ensures any process elected as leader has all the previous leader's committed log entries, because a majority of processes has the leader's committed log entries replicated and processes can only vote to processes with an up-to-date log. A candidate's log is more up-to-date than another log if the candidate's last log entry's term is higher than the other's last log entry's term; or if the candidate's last log entry's term is equal to the other log last entry's term, the longest is considered more up-to-date.

To ensure safety, Raft also adds a restriction for committing entries from older terms. A leader recognizes that an entry is committed only when an entry with higher index and term equal to the current term was stored in a majority of processes. The leader cannot conclude that an entry from an older term was committed if it was stored in a majority of processes because it's possible for a future leader to delete it.

2.3.5 Limitations of Raft

Leader bottleneck

Leader-based protocols, such as Raft, assign a leader that has a central role in the system usually performing more work than the non-leader ones, and this can cause the system to scale poorly. Raft's leader

handles all communication with clients and has an active role compared to the followers, since it makes decisions and sends `AppendEntries` to replicate its log and serve as heartbeat. Because of this, Raft is an algorithm suited for systems with few replicas [23].

As the leader uses more resources than followers, it can become a bottleneck to the system. This can happen with respect to system resources such as CPU and network bandwidth, since the leader handles more messages, due to having to broadcast `AppendEntries` to followers every client request, so the leaders' network bandwidth can become saturated, or the CPU might reach its limit first if the average request is small.

Network partitions

Raft assumes the network is transitive, messages may arbitrarily drop or delay (omission faults), but messages sent from a point will eventually be delivered to another point; however this is not viable for certain networks, like SDNs [18], where network partitions can exist. A real world example of an outage caused by this failure is the Cloudflare's November 2020 outage in their *etcd* cluster [30].

We now describe a scenario with network partitions where there is no stable leader. In the example on Figure 5 all processes are up-to-date, R1 (in red) is the leader and the links (R1, R3), (R1, R4) and (R2, R5) fail. R3 and R4 are not connected to the leader, so one (or both) will time out and become a candidate incrementing their term and sending *RequestVote* RPC, to their neighbours that will inform R1 and make it step down to follower. Either R3 or R4 will become leader, however they can't communicate with R1, so it will time out and become candidate and possibly leader again, then we are back at the initial setup again. In this setup, no process can become a stable leader because there is always a process that can't receive RPCs from the respective leader. This demonstrates how network partitions can cause loss of liveness in Raft.

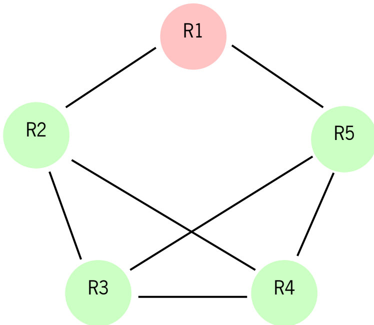


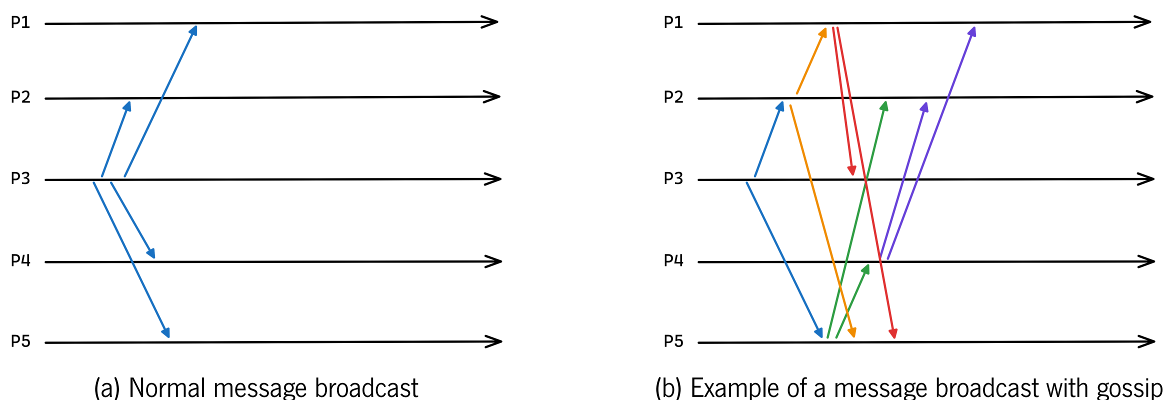
Figure 5: No stable leader

2.4 Gossip-Based Protocols

Gossip protocols [31], also known as epidemic protocols, are used in group communication due to their high scalability and reliability. They can be used to detect failures, aggregate or replicate data, reach consensus, etc.

The idea underlying most gossip protocols is that when a process intends to deliver a message m to all processes, it sends m to t random nodes (t is a configuration parameter called *fanout*); processes upon receiving m for the first time do the same [25]. This way m will, with high probability, be delivered to all processes at least once.

Figure 6 illustrates the difference between conventional one-to-one message broadcasting and the gossip-based approach in a system with five processes. On the left, Process P3 sends a message directly to each recipient, whereas with gossip, P3 sends the message to two randomly selected processes (fanout), which then forward the message upon receipt.



(a) Normal message broadcast

(b) Example of a message broadcast with gossip

Figure 6: Comparison of Message Broadcasting Methods

Gossip-based protocols are decentralized, since they rely on multiple processes participating in the dissemination of information, so the load is uniformly distributed. This approach can provide in some cases more efficient use of computing resources. Additionally, gossip-based protocols are reliable because of their ability to tolerate failures. Even if a process fails or some messages are omitted, as long as one process receives the message, the message can still be disseminated to the rest.

The simplistic approach of gossip-based protocols makes them easy to implement in practice. However, they can be less efficient and slower than other protocols due to the inherent redundancy in the dissemination of messages and because messages have to go through multiple processes before reaching all processes (more hops).

Parameters

There are two fundamental parameters that must be considered carefully in any gossip-based protocol:

- **Fanout:** number of random processes selected by a process to disseminate a received message for the first time. Higher values increase fault tolerance and broadcast reliability, however also increases redundant network traffic.
- **Maximum rounds:** maximum number of times a message is retransmitted by processes. Each message has an associated *round value* with initial value zero and increases monotonically each time a process retransmit the message. Processes only retransmit the message if its *round value* is smaller than the *maximum rounds* parameter. Some gossip protocols don't define this parameter, and the message dissemination terminates once all retransmitted messages reach processes that have received the message once. Defining *maximum rounds* can decrease message redundancy and keep high probability of achieving atomic delivery.

Strategies

There are several different approaches to implementing gossip-based protocols. We distinguish the following three basic approaches:

- **Eager push approach:** Processes, as soon as they receive a message for the first time retransmit, it to randomly selected peers;
- **Pull approach:** Periodically, processes request information from randomly selected peers about recently received or available messages. When a process becomes aware of a message it did not receive from a peer, it explicitly requests that peer for the message. This is a strategy that works best as a complement to a best-effort broadcast mechanism (e.g., IP Multicast [19]);
- **Lazy push approach:** When a process receives a message for the first time, it transmits only the message identifier. Receivers that have not yet received the message, make an explicit request for the message to the sender.

2.5 Relevant algorithms

In this chapter, we summarize some related algorithms involving the consensus and gossip paradigms. We start by introducing the mutable consensus algorithm, which utilizes strong failure detectors and stub-

born channels that can be easily mutated, with an interesting mutation being gossip-based. Then we explain Paxos atop Gossip, that simply implements Paxos atop a gossip communication layer, that can be approached using any other consensus algorithm besides Paxos and introduces the idea of semantic gossip. Afterwards we discuss S-Paxos, a Paxos variant that decentralizes the leader, decreasing the leader's bottleneck problem. Finally, we discuss Mandator-Sporades which joins Sporades, an omission fault-tolerant consensus algorithm and Mandator, an asynchronous and consensus-agnostic client request dissemination protocol.

Mutable Consensus

The Mutable Consensus protocol [33] achieves consensus in asynchronous distributed systems where processes are considered to be fully connected using stubborn channels and may crash, so eventual strong failure detectors, $\diamond S$, are assumed. Stubborn channels satisfy that messages received were previously sent (No-Creation property) and the last message sent by a process is eventually received if no following message is sent (Stubborn property). The protocol proceeds in asynchronous rounds, each with its designated coordinator. Each round has two phases where the coordinator tries to impose its decision. In the first phase, the coordinator's proposal is approved if a majority of the processes endorses it. When a process suspects that the coordinator has failed, then it enters the second phase and requests other processes to do the same, and as soon a majority does so, they proceed to the next round. Based on the implementation of the stubborn channel several mutations of the protocol are possible, being the first four mutations proposed: early, centralized, ring and permutation gossip. The correction of the protocol doesn't depend on the mutation. With great importance to us is the gossip permutation, in this implementation messages are immediately sent to f processes (gossip fanout) and delays it to the remaining processes. The permutation gossip mutation allows the protocol to scale to numerous processes.

Paxos atop Gossip (Semantic Gossip)

Gossip communication can complement consensus algorithms. In [17] a gossip communication layer is added to the Paxos consensus algorithm running over partially connected networks. The implementation of Paxos does not require any change. The paper introduces Semantic Gossip, which optimizes the classical gossip approach for consensus algorithms through two techniques, *semantic filtering* and *semantic aggregation*. Semantic filtering involves the use of a set of rules in order to decide if a message should be sent to a peer or if it can be discarded, according to the consensus semantics. Semantic aggregation involves aggregating multiple pending messages to be sent to a peer in accordance with a set of rules that

identify those that are prone to aggregation and define how messages are aggregated. These techniques reduce the number of messages exchanged without sacrificing the protocol and gossip properties.

S-Paxos

S(calable)-Paxos [16] is a variant of Paxos, sharing the same system assumptions, that tries to resolve the limitations of leader-centric protocols. In Paxos the leader is responsible for receiving requests from clients and distributing them to all processes so to decrease the leader's workload S-Paxos distributes this task to all processes. In S-Paxos any process can receive a request, with a unique global identifier, and the receiver is responsible for disseminating the request to the other processes. The leader utilizes the request identifiers instead of full requests to order them through the Paxos protocol. Each client's request is properly replied to by the receiver after executing it. This algorithm allows the system to scale, as adding more processes for fault-tolerance can improve performance.

Mandator-Sporades

Mandator-Sporades [39] is a modular state machine replication algorithm that enables high performance and resilience in the wide-area setting. Like Raft, the algorithm is tolerant to crashes and omission faults. Mandator and Sporades are the two building blocks of the algorithm.

Mandator is a protocol for disseminating clients requests, that is asynchronous and has no codependency with the consensus protocol. Mandator can reliably and efficiently replicate clients requests in batches over a wide-area network, and the workload for all processes is the same. The consensus layer can then refer to request batches by their unique identifiers, making communication lighter and faster.

Sporades is an omission fault-tolerant consensus algorithm, that consists of two modes of operations - synchronous and asynchronous - that always ensures liveness. The synchronous mode offers high performance under synchronous networks, that commits a batch of client requests in a single network round-trip. The asynchronous mode offers high resilience and liveness under asynchronous networks and process failures, that commits client requests with quadratic message complexity. Sporades dynamically switches between the synchronous and the asynchronous modes depending on the network condition.

Chapter 3

Driftwood

In this chapter, we present the novel mechanisms that will extend Raft creating Driftwood. Driftwood aims to improve the performance and scalability of Raft by decentralizing log replication, maintaining the leader election procedure, and safety properties of Raft. The leader in Driftwood uses gossip to replicate its log in a decentralized way: specifically, `AppendEntries` requests are gossiped instead of emitting RPCs to every follower (Section 3.1). However, we felt that we could further decentralize log replication by adding new replicated data structures that make it possible for followers to progress the `CommitIndex`, further decentralizing the system (Section 3.2). We distinguish between Driftwood with and without the replicated data structures by referring to the base as Version 1 and the version with replicated data structures as Version 2. Section 3.3 presents the correction argument for the replication procedure of Driftwood by referencing the safety properties of Raft.

3.1 Gossip of `AppendEntries` Requests

Raft uses RPCs for communication between processes to directly replicate missing log entries in the receiver, as exemplified in Figure 7a. While this approach makes it easier to reason about the algorithm, it is also one of its limitations. As the number of replicas increases, the leader will need to handle more RPCs, and therefore more messages, to replicate its log.

To ensure the followers consistently replicate the leader's log, a partition of the leader's log with new entries matching each follower's log must eventually be delivered. Raft uses RPCs because it is an efficient way to replicate the missing entries. However, Raft allows redundant entries to be included in the `AppendEntries` request, allowing the leader to broadcast a single `AppendEntries` request containing the largest log partition necessary to replicate the log between all the followers.

Gossip is a decentralized broadcast mechanism designed to deliver messages to all participants with high probability and reliability. In order to use this mechanism for replication, we need to consider would

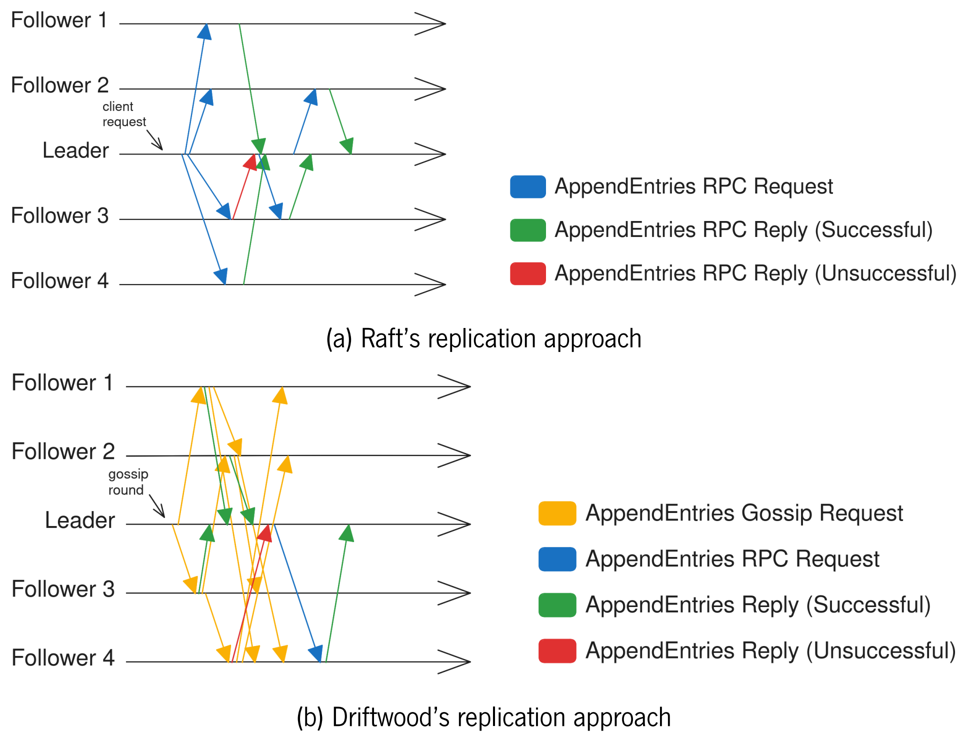
be the leader's log partition that would maximize the number of entries not yet replicated while minimizing the number of redundant entries (entries that have already been replicated). If the gossiped message is successfully delivered to all participants, then sending the uncommitted entries will allow at least the majority of processes to replicate the log, allowing the system to progress. However, followers with logs missing committed entries will not be able to replicate the leader's log by receiving only the uncommitted entries. In this case, the leader needs to gossip a message containing entries from an earlier point or directly replicate the missing entries, following the original algorithm.

In Driftwood, the leader initiates gossip rounds to broadcast uncommitted log entries through a single `AppendEntries` request, as exemplified in Figure 7b. Followers, upon receiving a request for the first time, reply to the leader about the replication's success. Occasionally, followers might fail to replicate some entries due to network faults or crashes, causing their log to be behind the leader's `CommitIndex`. In these instances, these followers will reply that the replication was unsuccessful. In response, the leader will initiate the usual `AppendEntries` RPC, as in Raft, to ensure that the missing entries in that follower's log are replicated and its log is up-to-date with the entries broadcasted in the subsequent gossip rounds. If the reply doesn't reach the leader, the follower will reply unsuccessfully again to the next `AppendEntries` request.

The leader initiates periodic gossip rounds with `AppendEntries` requests containing uncommitted entries. This is advantageous as it gives time for the gossiped request to be delivered and for followers to reply before the next gossip round initiates and permits temporal batching of log entries. We refer to this as `AppendEntries` Gossip, and, unlike the RPC mechanism, the leader in this case doesn't wait for followers' replies, and the followers only reply to newer `AppendEntries` requests (i.e., from newer gossip rounds).

`AppendEntries` gossip rounds employ a permutation-based method to select the gossip targets [33]. A process, upon becoming leader, calculates a permutation of followers, which is circularly traversed in each gossip round, as specified in Algorithm 1.

Figure 8 provides a concise overview of the new mechanisms that make Driftwood. Within each process's state, we introduce a new integer variable, `RoundLC`, which helps followers distinguish between `AppendEntries` messages disseminated in each gossip round. `RoundLC` is set to zero at the start of a new term and is only incremented by the leader when it initiates a new gossip round. When a follower receives an `AppendEntries` gossip request, it checks the value of `RoundLC` contained within the request. If the received value is higher than its own copy, the follower knows that it is the first time



(a) Raft's replication approach

(b) Driftwood's replication approach

Figure 7: Leader log replication in Raft and Driftwood

Algorithm 1: Permutation gossip round for the process $P_i, i \in 0..n-1$

State:
F ← fanout value
c ← 0
u ← random permutation from 0 to $n-1$ except i

Function `Round`(*m*)**:**
  for $i \leftarrow 0$ **to** F **do**
    send *m* to u[(c + i) mod F];
  end
  c ← c + F;

---

receiving this message and proceeds to process it accordingly. By consistently updating `RoundLC` with the highest received values, the follower effectively filters out messages from older rounds.

To differentiate between RPC and gossip `AppendEntries` requests, the leader includes a boolean variable within the request. When set to `true`, the request is a RPC, indicating that the receiver should process the message and respond accordingly without propagating it.

According to Raft, the leader also uses two variables to keep track of the followers' log states: the `MatchIndex` which states, for each process, the index of the highest log entry known to have been successfully replicated, and the `NextIndex`, which specifies, for each server, the index of the next log entry to be sent to that process. To assist the leader in updating these variables, the follower includes in the `AppendEntries` reply the lowest of the two indices: the index of the last entry in its log and the index

immediately preceding the entries sent in the `AppendEntries` request. Consequently, when the leader receives an unsuccessful `AppendEntries` reply, it starts an `AppendEntries` RPC that continues until the entries being sent match the respective follower's log, successfully replicating the leader's log.

State

The state is the same as Raft's with the following additional variable

New volatile state on all servers:

roundLC: Initialized to 0 when a new term starts and incremented at each gossip round by the leader; follower's highest roundLC received in current term from AppendEntries gossip, reset to 0 on new term discovery

Rules for Servers

New rules for servers

All Servers:

- On new term set roundLC to 0

Follower:

- Responds to RequestVote RPCs and AppendEntries requests with leaderRound > roundLC or isRPC is true
- Gossips AppendEntries request if term ≥ currentTerm, leaderRound > roundLC and isRPC is false
- If election timeout elapses without receiving new valid AppendEntries request (with higher roundLC or isRPC is true, and term ≥ currentTerm) or granting vote to candidate: convert to candidate

Leader:

- Upon election over or idle period: send gossip round of empty AppendEntries (heartbeats)
- While commitIndex < last log entry index periodically send AppendEntries gossip round
- Upon receiving AppendEntries reply from *i*, if successful and lastLogIndex > matchIndex[*i*] set nextIndex[*i*] = lastLogIndex+1 and matchIndex[*i*] = lastLogIndex else if unsuccessful set nextIndex[*i*] = lastLogIndex and send AppendEntries RPC to *i*

AppendEntries RPC/Gossip

Arguments:

Same as Raft's AppendEntries RPC arguments with the following additional values

leaderRound: leader's roundLC

isRPC: true if message is RPC

Result:

Sent to the leader if is RPC or leaderRound higher

term: currentTerm

success: true if follower contained matching entries

lastLogIndex: minimum of last entry index in log and prevLogIndex-1, in order to update nextIndex[] and matchIndex[]

Receiver Implementationt:

1. Reply false if term < currentTerm
2. Return if leaderRound ≤ roundLC and isRPC is false
3. If log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm delete entries starting from prevLogIndex and reply false
4. Replace conflicting entries and append new ones
5. Set commitIndex = min(leaderCommit, last new entry index)
6. If not RPC then set roundLC = leaderRound and gossip message

Figure 8: A condensed summary of the new mechanisms that make the first version of the Driftwood consensus algorithm an extension of Raft. Driftwood maintains most of the Raft logic, so we exclude the `RequestVote` RPC and the state variables already summarized in [32].

3.2 Adding Replicated Data Structures

Driftwood's gossip mechanism reduces the number of messages that a Raft leader would have to handle by sending uncommitted entries in batches and delegating the dissemination of these entries go to the followers. However, we maintain the approach where the leader advances the `CommitIndex` only after receiving acknowledgments from a majority of followers indicating the successful replication of the log entries.

We further extend Driftwood by adding new replicated data structures, referred to as `CommitStatus`, that synergize with the gossip mechanism, allowing the `CommitIndex` to progress in a decentralized way. The processes use gossip to propagate their data structures so that they can progress.

Each process maintains the `CommitStatus` in their local state, which consists of the following variables:

- **`Bitmap`**: an array of bits that keeps track of the processes known to have successfully replicated their local log up to the index `NextCommit`. Each index in the array corresponds to a specific process, and that process sets its value to "one" when the posterior condition is verified;
- **`MaxCommit`**: maximum value that can be assigned to the `CommitIndex`. It is the highest log index that can be committed, indicating that the log entries up to this index have been successfully replicated to a quorum of processes;
- **`NextCommit`**: log index being considered as the next `MaxCommit` value in the current term.

Processes share their `CommitStatus` in `AppendEntries` requests, ensuring the convergence and progression of the values. The `Bitmap` is used to find out when a quorum of processes has replicated their log successfully up to the `NextIndex`. Each process must assign its value in the `Bitmap` to "one" if the index `NextIndex` in its log has the same term as its current term. According to the log matching property, this condition signifies that the log is an accurate replica of the leader's log up to an index higher or equal to `NextIndex`.

We define two functions to advance the `CommitStatus`:

- **`Update`**: to progress the values when a majority is reached in the `Bitmap`;
- **`Merge`**: to converge the values with those received from the other processes.

3.2.1 Function `Update`

Function `Update`, according to Algorithm 2, advances the values of `MaxCommit` and `NextCommit`, and resets the `Bitmap` when a majority is reached (line 1). This means that a majority of processes set their value in the `Bitmap` of the current `NextIndex` and `Term`, i.e., these processes successfully replicated the current leader's log up to the `NextCommit`, and therefore `NextCommit` and `MaxCommit` can advance. When this happens, `MaxCommit` is set to `NextCommit` and the `Bitmap` is reset with "zeros" (lines 2 and 3). Then, `NextCommit` advances according to the current state of the local log: if `NextCommit` is in a point more advanced than the log or the log doesn't have an entry with its term equal to the current term (line 4), then `NextCommit` is incremented by one; else the log has an entry with index higher than the current `NextCommit` with term equal to the current term, so the value of `NextCommit` is set to the higher value that the process knows will be replicated in the current term, i.e., the highest entry

index in the local log (line 7) and, under these conditions, the process can put its value in the updated `Bitmap` to “one” (line 8).

Algorithm 2: Fuction to update `NextCommit` and `MaxCommit` when `Bitmap` shows a majority for process $P_i, i \in 0..n-1$

Local State: bitmap; maxCommit; nextCommit;
Function `Update()`**:**

```
1   if count of "1"s in bitmap ≥ majority size then
2       maxCommit ← nextCommit;
3       bitmap ← {0, 0, ..., 0};
4       if nextCommit ≥ index of last entry or term of last entry ≠ current term then
5           nextCommit ← nextCommit + 1;
6       else
7           nextCommit ← index of last entry;
8           bitmap [i] ← 1;
9       end
10  end
```

This algorithm restricts the `NextCommit` from advancing to a point that can possibly be higher than the leader’s log by comparing the highest term in the log with the current term, as entries from previous terms can be removed from a follower’s log when a new term begins. However, this is not enough to ensure that after a new term begins, all followers’ `NextIndex` is smaller or equal to the last entry index + 1 in the new leader’s log. In order to satisfy this condition, we regress `NextCommit` and `Bitmap` to a value that is safe for any possible process elected. When a process discovers a higher term, it resets the `Bitmap` to “zeros” and sets the `NextIndex` to the value of `MaxCommit` + 1.

This comes from Raft’s safety restriction that says that the leader can’t commit entries from previous terms until it can commit an entry from its own term. In Driftwood, the same property is valid for the `MaxCommit`, where this value can only advance after a majority of processes have an entry in the current term.

Examples

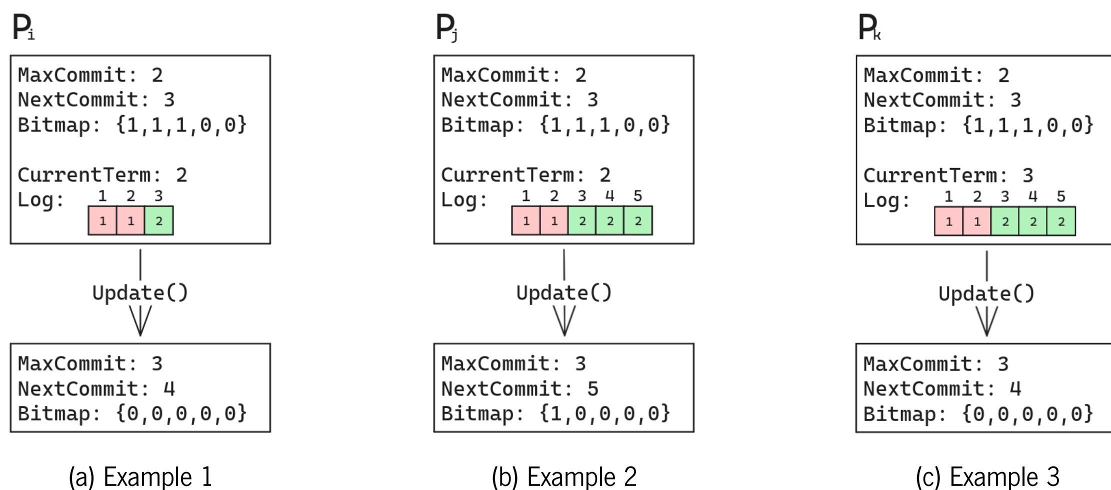
To further illustrate how the `Update` function operates, we will explore three examples with process abstractions in Figures 9 and an example of a system abstraction in Figure 10 are provided.

We start with Figure 9a where process P_i’s log has three entries, and the term of the last entry is equal to its current term. P_i notices a majority in its `Bitmap` and proceeds to update its `CommitStatus`. This means that a majority of processes successfully replicated the log up to the `NextCommit` which

corresponds to its last log index. So P_i sets the `MaxCommit` to the current `NextCommit`; `NextCommit` is then set to the index after that, which corresponds to the next entry that will possibly be received; and resets the `Bitmap` to "zeros". This means that P_i's log is completely committed.

The next example, in Figure 9b, shows a process P_j with a state similar to that of P_i, but its log is longer. It becomes evident why advancing the `NextIndex` to a higher value is more efficient. P_j has an entry with a higher index value within the current term, i.e., added by the current leader. Therefore, P_j sets the `NextIndex` to the last log index. Since it has successfully replicated the log up to this index, P_j resets the `Bitmap` and can set its value to "one."

The third example, in Figure 9c, shows a process P_k with a state similar to that of P_j, only the current term is higher. In this case, P_k necessarily has an outdated log as it discovered that other replicas replicated the leader's log up to an entry with term 3, as they could turn its value in `Bitmap` to "one" but P_k itself couldn't. So in this case, P_k discovers a majority, but P_k doesn't know if the entries from previous terms are present in the leader's log, so P_k increments its `NextIndex` by one and resets the `Bitmap`, but it can't set its value to "one" because it doesn't possess an entry from the current term.



(a) Example 1 (b) Example 2 (c) Example 3

Figure 9: Examples of processes' states

Another example in Figure 10, illustrates the logs of five processes in a cluster. In this scenario, followers 2 and 3 can't vote in the `Bitmap` for term 5 because they don't possess an entry with term 5 in their logs, resulting in an inconsistency with the leader's log, as seen in follower 2's log. However, they can still advance the replicated structures by observing a majority due to the `Merge` function, which uses the bitwise or operation to merge bitmaps. For safety, in this case, if followers 2 or 3 observe a majority in their `Bitmap` and are not able to replicate the received entries, they only increment the `NextCommit` by one (equivalent to the updated `MaxCommit` + 1).

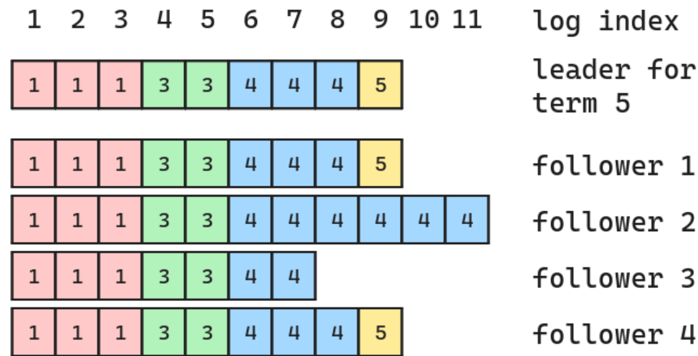


Figure 10: Possible logs for a five-process cluster

In this scenario, if the leader doesn't append more entries and maintains leadership, then eventually the NextCommit value will reach 9. After its log is replicated to a majority, the MaxCommit and NextCommit will be updated to 9 and 10 respectively, indicating that the log was totally committed and the next entry to be committed is the next entry to be appended by the leader.

3.2.2 Function Merge

Each process, upon receiving an AppendEntries request with a term equal to or higher than their current term, merge the CommitStatus of the sender included in the request with their own using the function Merge, according to Algorithm 3.

Firstly, the MaxCommit is updated to the highest value between local and received value (line 1). Then, if the local NextCommit is smaller or equal to the value received (nextCommit'), the information on the received Bitmap can be merged into the local Bitmap using a bitwise or operation (lines 2 to 4). This is possible because the processes with "one" bits in the Bitmap have successfully replicated the log up to the respective NextCommit, and thus have replicated successfully any log prefix. At this point, it is important to ensure that the NextCommit is not behind the updated MaxCommit. If the local NextCommit is smaller than the updated MaxCommit, it means that the process's variables are outdated, and waiting for the local Bitmap majority is redundant. So, in this situation, the received variables are necessarily more advanced, so we can safely set the local variables to match the received ones (lines 6 and 7).

The NextCommit value being smaller than MaxCommit is nonsensical and counterproductive, as it represents a redundancy in the system. Therefore, as a safety requirement, Driftwood ensures that NextCommit is always greater than MaxCommit both before and after the execution of the functions Merge and Update.

Algorithm 3: Function to merge `CommitStatus` in `AppendEntries` requests received for the process $P_i, i \in 0..n-1$

```
   Local State: bitmap; maxCommit; nextCommit
   Function Merge(bitmap', maxCommit', nextCommit'):
1      maxCommit ← Max(maxCommit, maxCommit');
2      if nextCommit ≤ nextCommit' then
3          bitmap ← BitwiseOR(bitmap, bitmap');
4      end
5      if nextCommit ≤ maxCommit then
6          bitmap ← bitmap';
7          nextCommit ← nextCommit';
8      end
```

The `Merge` function exhibits asymmetry, meaning that if we invert the local `CommitStatus` and the received one, the resulting local `CommitStatus` might not be the same.

Examples

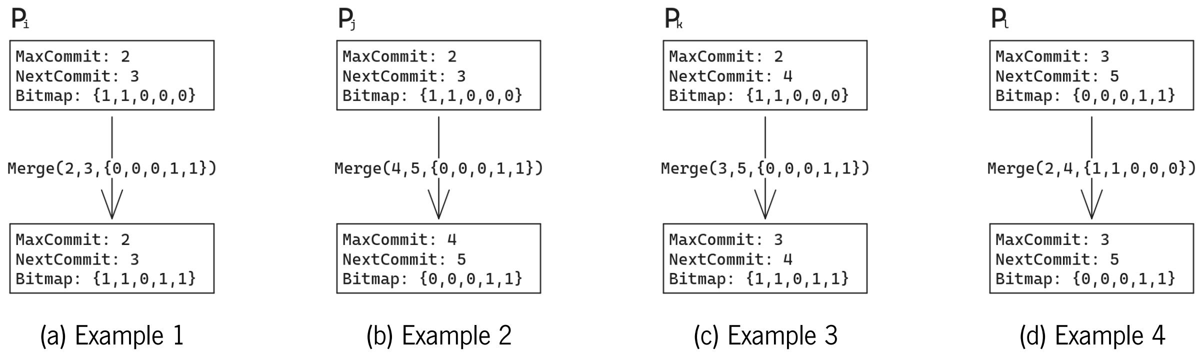
We showcase the function `Merge` in action through several examples in Figure 11. These examples demonstrate the outcome of merging the local `CommitStatus` with one received in an `AppendEntries` request.

In the first example, shown in Figure 11a, as expected, when the received `NextCommit` and `MaxCommit` values are the same as in the local state. In this case, the bitmaps are merged and the `NextCommit` and `MaxCommit` are unchanged.

In the second example, in Figure 11b, the received `NextCommit` and `MaxCommit` values are higher than the ones in the state. In particular, the received `MaxCommit` is higher than the `NextCommit` in local state, which indicates that the local values are outdated and keeping the current `NextIndex` is redundant. So these must be replaced with the received values which are more up-to-date.

The third example, in Figure 11c, is similar to the second example but, in this case, the received `MaxCommit` is smaller than the local `NextCommit`, so the local `NextCommit` is still valid. Merging the bitmaps makes sense since the received `NextCommit` is higher than the local `NextCommit`, and the received `Bitmap` is valid for any prefix up to the received `NextCommit`.

In the fourth example, in Figure 11d, we reverse the local and received `CommitStatus` presented in the third example. As the values in the state are more up-to-date, then there is nothing to be gained from the received information. In this scenario, the process cannot merge the bitmaps because it cannot ascertain whether the other processes' logs were successfully replicated up to the local `NextCommit`. As shown in the third and fourth examples, the function `Merge` displays asymmetry.



(a) Example 1 (b) Example 2 (c) Example 3 (d) Example 4

Figure 11: Examples of the `Merge` function

3.2.3 Overview of Version 2 of Driftwood

In Version 1 of Driftwood, followers propagate `AppendEntries` requests between these, excluding the leader. However, in Version 2 of Driftwood, followers also forward gossip messages to the leader, enabling him to also advance its own `CommitStatus`. One significant distinction in Version 2 is that the `CommitIndex` can be ahead in some followers when compared to the leader's, which is not possible in the original Raft or Version 1.

Each processe's `CommitIndex` increases monotonically according to their log state and `MaxCommit`. The `CommitIndex` is the maximum between the last entry's index in the log and `MaxCommit`, if the term of the last log entry is equal to the current term.

Functions `Merge` and `Update` are used upon receiving `AppendEntries` requests. The process will only merge the received `CommitStatus` from `AppendEntries` requests with terms equal to or higher than the current term. If this condition is met, the process merges the structures, which may result in reaching a majority. In this case, if the process is the leader, it can call the `Update` function and try advancing its `CommitStatus`. Followers can attempt to update their `CommitStatus` after trying to add the entries received in the request. Before calling the `Update` function, processes should also check if they are eligible to set their value in the bitmap.

On a new term, when a process starts an election or discovers a higher term, the variables `RoundLC`, `Bitmap` and `NextCommit` are reset.

Figure 12 summarizes the Version 2 of the Driftwood algorithm.

State

The state is the same as Raft's with the following additional variables

New volatile state on all servers:

roundLC: Initialized to 0 when a new term is discovered and incremented at each gossip round by the leader; follower's highest received roundLC in current term from AppendEntries gossip, reset to 0 on new term discovery

bitmap[]: Bitmap used to discover majority on the next maxCommit value

nextCommit: index of log entry being "voted" for next maxCommit value (initialized to 1, increases monotonically during a term)

maxCommit: maximum possible value of commitIndex (initialized to 0, increases monotonically during a term)

Rules for Servers

New rules for servers

All Servers:

- If RPC or AppendEntries gossip request contains term *T* > currentTerm: set currentTerm = *T* and convert to follower
- Set commitIndex = max(last index in log, maxCommit) if index of last log entry is equal to currentTerm
- If nextCommit ≤ last log entry index and log[nextCommit].Term = currentTerm: set its bitmap value to 1
- On new term set
  - roundLC = 0
  - bitmap = {0,0,...,0}
  - nextCommit = maxCommit+1

Follower:

- Gossips valid AppendEntries requests (with term ≥ currentTerm, leaderRound > roundLC and is not RPC)
- If election timeout elapses without receiving new valid AppendEntries (with higher leaderRound) or granting vote to candidate: convert to candidate

Leader:

- Upon election win or idle period: send gossip round of empty AppendEntries (heartbeats)
- While commitIndex < last log entry index periodically send AppendEntries gossip round

AppendEntries RPC/Gossip

Arguments:

term: leader's term

leaderId: leader's identifier

prevLogIndex: leader's commitIndex

prevLogTerm: term of prevLogIndex entry

entries[]: log entries to store (empty for heartbeat), starting from entry at prevLogIndex+1

leaderRound: leader's roundLC

isRPC: true if message is RPC

bitmap[]: server's updated bitmap

maxCommit: server's updated maxCommit

nextCommit: server's updated nextCommit

Result:

Sent to the leader if is RPC request or replication unsuccessful

term: currentTerm

success: true if follower contained matching entries

lastLogIndex: minimum of last entry index in log and prevLogIndex-1, in order to update nextIndex[]

Receiver Implementation:

1. Reply false and return if term < currentTerm
2. Merge nextCommit, maxCommit and bitmap §(3.2)

If server is leader:

3. Update nextCommit, maxCommit and bitmap §(3.2)

If server is follower:

3. Return if leaderRound ≤ roundLC and isRPC is false
4. If log contains an entry matching commitIndex and commitTerm then:

   a. replace conflicting entries and append entries not already in the log

   b. update nextCommit, maxCommit and bitmap §(3.2)

   Else delete conflicting entries (log entries starting from prevLogIndex)
5. If is RPC or log didn't match then reply to leader
6. If is not RPC then set roundLC = leaderRound and start gossip round with updated bitmap, maxCommit and nextCommit

Figure 12: A condensed summary of the new mechanisms that make the second version of Driftwood. We omit again the overlapping logic with Raft as well as the detailed explanations of the functions to update and merge the new replicated data structures, which are explained in 3.2.1 and 3.2.2, respectively.

3.3 Correction Argument

In this section, we discuss the safety of Driftwood, referring to Raft's established safety properties. These properties encompass Election Safety, Leader Append-only, Log Matching, Leader Completeness, and State Machine Safety. Given that Driftwood primarily modifies the Log Replication procedure, our focus is directed towards the Leader Append-only and Log Matching properties.

The **Leader Append-only** property, which stipulates that the leader never overwrites or deletes en-

tries, is trivially verified by the algorithm's specification. Just as in Raft, when a process is in the Leader state, it only appends new entries to its log upon receiving client requests. Additionally, Driftwood's leader just tries to replicate the new entries in subsequent gossip rounds until they are committed.

The **Log Matching** property specifies that when two logs contain entries with the same index and term, all entries with indexes smaller than or equal to the matching entry are guaranteed to be identical. In Driftwood, all processes initialize with an empty log, satisfying this property. When a leader is elected, upon receiving client requests, it initiates periodic gossip rounds until all the entries in its log are committed. When the `AppendEntries` gossip message is delivered to a follower, two scenarios can happen:

- The follower's log matches the leader's log up to the entry preceding `AppendEntries`' entries (i.e., validates the Log Matching property). Then the follower adds the received entries to its log, replacing any mismatched entries and ensuring its log is an equal replica of the leader's log.
- The follower's log doesn't match the leader's log at the entry preceding `AppendEntries`' entries, meaning there is some inconsistency between the logs. The follower informs the leader of unsuccessful replication and requests entries from a previous point. The leader obliges and starts an `AppendEntries` RPC, directly sending entries until the logs reach a matching point and the follower can replicate the leader's log.

The method used by the leader to identify committed entries differs between the two Driftwood versions:

- In Version 1, the followers notify the leader of their successful replication of the log. Upon receiving this confirmation from a majority of followers, the leader can commit the log up to the corresponding entry if it was created in the leader's term.
- In Version 2, followers that have successfully replicated the leader's log up to the `NextIndex` can set their bit in the `Bitmap` if they possess an entry from the current term. When the `Bitmap` reaches a majority of bits "1", the `MaxCommit` is set to the `NextCommit`. This change signifies that a majority of processes have successfully replicated the leader's log up to the updated `MaxCommit`. The leader identifies committed entries when it receives a `MaxCommit` value higher than the previous values it has received. Followers can commit their log entries once they possess an entry added by the current term's leader, indicating that their log is a prefix of the leader's log, and discover a `MaxCommit` higher than the index of their log's last committed entry.

In Driftwood, similar to Raft, the leader's strategy for dealing with log inconsistencies involves making followers totally replicate their own log, which can lead to overwriting entries from older terms. To guarantee that the leader doesn't overwrite committed entries from previous terms, Raft introduces a vital restriction that prevents leaders from assuming that entries from previous terms are committed by counting replicas that have successfully replicated these entries. The commitment of these entries is only recognized when a leader confirms that an entry from the current term has been successfully replicated. Version 1 of Driftwood maintains this restriction, as the commitment discovery process by the leader is identical to Raft's.

In Version 2 of Driftwood, since the commitment discovery process is different, this restriction is applied to the `Bitmap`. Specifically, a follower can only set its bit in the `Bitmap` if the last index in its log is greater than or equal to `NextCommit` and its term is equal to the current term. This behaviour is equivalent to Raft, where a follower informs the leader of its log's successful replication up to the entry with index `NextCommit`, and the leader only commits this entry if it receives the confirmation from a majority of processes that have successfully replicated an entry added by the leader. Consequently, in Version 2, `MaxCommit` can only progress when a majority of replicated logs contain at least one entry added by the current term's leader. Therefore, entries from previous terms can be considered committed when this condition is met. This approach aligns with Raft's safety properties, ensuring that committed entries are never overwritten by new leaders.

Chapter 4

Implementation

We implement Raft and the two versions of Driftwood in Paxi [10, 12], a prototyping framework for modelling strongly-consistent replication protocols, used mostly to evaluate and compare different replication protocols. In this chapter, we start by describing the Paxi framework in Section 4.1, in order to understand how it's used to implement replication algorithms and benchmark them. Then, in Section 4.2, we explain our Raft implementation, which will serve as the baseline comparison for Driftwood. We expanded the Raft implementation to Driftwood Version 1 and Version 2, explained in Section 4.3. Both the algorithms and implementations feature various parameters that must be carefully considered, described them in Section 4.4.

4.1 Paxi Framework

The Paxi framework [10, 12] was developed to empirically compare strongly-consistent replication protocols. Paxi includes components that implement common parts of many replication algorithms for networking, message handling, data storage, etc., with only two modules having to be developed to describe the distributed consensus/replication protocol. Each component resides in its own loosely coupled module which can be extended or replaced if necessary to better accommodate the distributed protocol, provided that the new module follows the existing interface. Paxi architectural overview is shown in Figure 13, where the two shaded blocks correspond to the two modules necessary to implement a distributed protocol. Often, it is only necessary to specify the protocol message structures and extend the node interface to reflect process behaviour according to the protocol.

Paxi also provides benchmarking support that can both generate tunable workloads in accordance with a fixed configuration file and collect respective performance data. Paxi is implemented in the Go [21] programming language.

We will now provide small explanations of the Paxi components relevant to developing and evaluating

the implemented protocols.

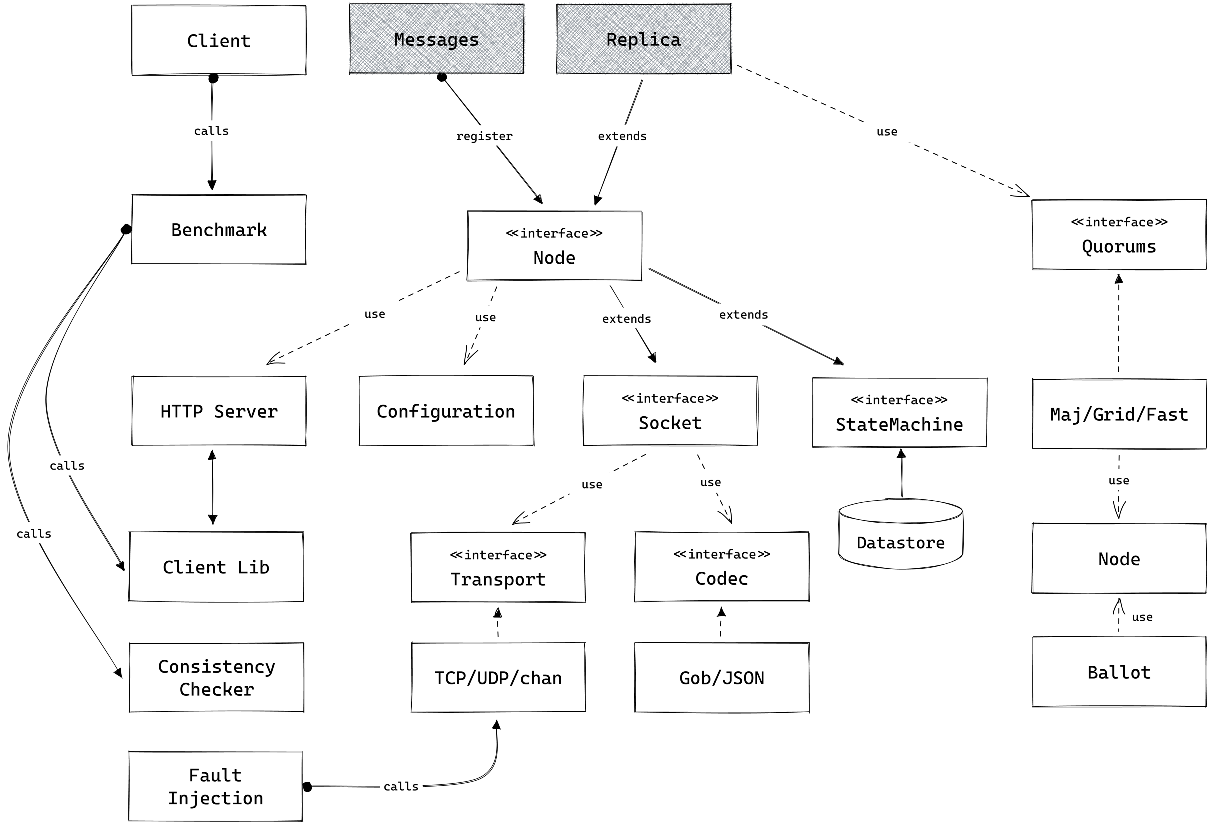


Figure 13: Paxi modules [12]

Configurations

Paxi has many configurable parameters for the processes and benchmarking. A configuration in Paxi is provided to all processes in a JSON file, containing vital information for the system such as: a list of peers with their reachable addresses, buffer sizes, message serialization and networking parameters, and other configurable settings. Once a configuration is loaded by a process, the configuration parameters remain fixed for it.

Networking

The message-passing model in Paxi is non-blocking and allows any algorithmic logic to be expressed as a set of event handlers. The networking module handles message encoding, decoding and transmission, simplifying message passing between nodes through the use of *Send()*, *Recv()*, *Multicast()* and *Broadcast()* interfaces. The transport layer used for communication between processes can be either TCP, UDP or Go channels.

Data store

Paxi comes with an in-memory multi-version key-value data store that is private to every process and is used as a deterministic state machine abstraction. The framework allows other data models to be implemented and used; however, this might influence client implementation.

RESTful Client and Benchmarking

Communication between the system and clients is done through a RESTful API that can request `Read` and `Write` operations. Client operation requests are sent, by default, to any process in the system. We used the Paxi client library for benchmarking, although any other benchmarking or testing tools could be easily used instead.

The benchmarker component in Paxi generates a workload for the system according to workload definition parameters defined in the configuration file. These parameters are summarized in Table 1.

Parameter	Default Value	Description
T	60	Run for T seconds
N	0	Run N operations if (N>0)
K	1000	Total number of keys
Throttle	0	Requests sent per second (0 if unlimited)
W	0.5	Write ratio
Concurrency	1	Number of concurrent clients
LinearizabilityCheck	True	Check linearizability at the end of benchmark
Distribution	"uniform"	Name of distribution used for key generation include "uniform", "normal" and "zipfian"
Min	0	Random: Minimum key number
Conflicts	100	Random: percentage of conflicting keys
Mu	0	Normal: Mean
Sigma	60	Normal: Standard Deviation
Move	False	Normal: Moving average (mu)
Speed	500	Normal: Moving speed in milliseconds
Zipfian_s	2	Zipfian: s parameter
Zipfian_v	1	Zipfian: v parameter

Table 1: Paxi benchmark parameters
[12]

The benchmarker allows us to test and evaluate the implemented protocols in terms of their performance, consistency, availability and scalability. Latency and throughput metrics are measured and registered throughout an experiment's runtime by the benchmarker. It registers, at the end of an experiment, the number of requests made, throughput, and latency metrics such as mean, minimum, maximum, median and 95, 99 and 999 percentiles of client response time.

In order to test and evaluate replication and consensus protocols under faults, Paxi has fault injection operations that can be used to simulate any node or network failure. Paxi provides the following four functions to inject faults into a node's network modules:

- **Crash(t)**: node stops communication with others during t seconds;
- **Drop(i, t)**: node stops sending messages to node i during t seconds;
- **Flaky(i, d, t)**: node stops sending messages with d probability during t seconds;
- **Slow(i, d, t)**: node delays sending every message d milliseconds to node i during t seconds;

Moreover, Paxi has a consistency checker component capable of verifying client linearizability and consensus between the processes state machines.

4.2 Raft Implementation

The implementation of Raft's logic is mostly synchronous, utilizing a handler function that processes events in a reactive loop. Within this loop, we prioritize timeout events, ensuring progress. In the absence of timeout events, we shift our attention to pending messages, processing the next message in the buffer with its respective handler function.

4.2.1 Messages Module

To implement Raft in Paxi we started by specifying the message data structures. Raft uses two kinds of RPCs: `AppendEntries` RPC and `RequestVote` RPC. So for each RPC, we specify a pair of data structures, one for the request and the other for the respective reply. We define the struct types `AppendREQ` and `AppendREP` for the `AppendEntries` RPC, and the struct types `VoteREQ` and `VoteREP` for the `RequestVote` RPC. The fields in each message data structure are defined as per the Raft algorithm and contain an additional field indicating the sender of the message.

In our approach, a Raft's RPC is equivalent to sending a request message and subsequently awaiting (in a non-blocking manner) for the reply to be processed by the event handler. This might negatively impact the algorithm's performance, as in Raft, RPCs can be issued in parallel.

We can see a code snippet of the `VoteREQ` struct type. All messages contain the `From` and `Term` fields and then their own unique fields as defined in the Raft paper [32].

```
type VoteREQ struct {
    From         paxi.ID // sender's ID
    Term         Term    // sender's term (candidate term)
    CandidateID  paxi.ID // candidate requesting vote
    LastLogIndex int     // index of candidate's last log entry
    LastLogTerm  Term    // term of candidate's last log entry
}
```

Listing 4.1: `VoteREQ` type struct code snippet

4.2.2 Replica Module

The Raft algorithm proposes that processes can be in one of three states: follower, candidate or leader. In our implementation, processes keep track of their states through a private variable.

We utilize the Golang built-in type `Ticker` to implement timeout events. The `Ticker` allows us to define a duration or interval, and it will send time events at regular intervals based on that duration. One of the advantages of using a `Ticker` is that we have the flexibility to stop or reset it as needed. This flexibility allows us to adapt the timeout behaviour according to our requirements. We define two tickers:

- `Election Ticker:` used by followers and candidates to detect when a certain random interval of time passes without a correct leader being detected, and this is achieved by resetting the ticker when a valid `AppendEntries` request is received;
- `Heartbeat Ticker:` used by leader to periodically send `AppendEntries` requests to all followers.

The Raft implementation possesses a main event handler that handles all events like timeouts and pending messages. Each message structure type has its own function handler that is called when the message arrives at the event handler. The code snippet 4.2 shows the handler function for `VoteREP` messages.

```
func (r *Replica) handleVoteREP(m VoteREP) {

    if m.Term > r.currentTerm {
        r.stepDown(m.Term) // transitions to or remains Follower

    }else if r.state == Candidate && m.Term == r.currentTerm && m.
   VoteGranted {
```

```
7
8        r.voteCount += 1
9
10       if r.voteCount > (paxi.GetConfig().N() / 2) { // has majority
11           r.stepUp() // transitions to Leader
12       }
13   }
14 }
```

Listing 4.2: Code snippet of the `VoteREP` handler function

When a leader process handles a client request, it appends the received operation as a new entry to the log and sends an `AppendEntries` request to each follower. Upon confirmation that a follower has successfully replicated the entries in the request, the leader updates the corresponding `MatchIndex` and `NextIndex`, effectively ceasing to send entries that have already been replicated in the follower's log. On the other hand, if a follower fails to replicate the entries, the leader will reduce the corresponding `NextIndex` and send a new `AppendEntries` request to that follower. This process repeats until the entries match the follower's log. However, it's possible that either the `AppendEntries` request or reply is not delivered, leaving the leader uncertain about whether the entries have been successfully replicated. To ensure full log replication across all followers, even in the presence of message loss, the leader periodically sends `AppendEntries` requests to each follower. These requests also serve as heartbeats and include the entries that have not yet been replicated by the respective receiver (according to the `NextIndex`).

This implies that new entries are not efficiently batched, leading to an increase in the number of exchanged messages during periods of heavy client loads, which adversely affects performance.

The algorithm assumes that clients should send the request directly to the leader. In our implementation, clients can send requests to any process, which will then forward the request to the leader if it exists; otherwise, it will respond with an error.

4.3 Driftwood

We have developed two versions of Driftwood. In Version 1, we extend the Raft implementation by introducing gossip rounds of `AppendEntries`. In Version 2, we go a step further by incorporating new replicated data structures, `CommitStatus`, into the gossip mechanism to discover committed log entries.

4.3.1 Version 1: Addition of Gossip

In Driftwood, the leader periodically sends `AppendEntries` requests using gossip to the followers while there are uncommitted entries in the log. To achieve this, we've introduced a new event in the event handler. This event is triggered by a dedicated `Ticker` that generates the event at fixed time intervals while it's active. The ticker starts when a new client request is received and the ticker is not already active. It's halted when the last entry in the log is committed. When the event occurs, the process initiates a permutation gossip round function. This function creates an `AppendEntries` request containing the uncommitted entries and sends it out using permutation gossip. You can find a description of the event handler behaviour in Algorithm 4.

The permutation of process identifiers used by each process is created on start-up and is fixed.

Algorithm 4: Event handler function in Driftwood implementation

```
Function EventHandler():
    Select
        Case Heartbeat Timeout do
            // Is leader;
            start permutation gossip round of new heartbeat
        End
        Case Permutation Gossip Timeout do
            // Is leader;
            start permutation gossip round
        End
        Case Election Timeout do
            // Is follower or candidate;
            transitions to the candidate state and initiates an election
        End
        Case Message Pending do
            handle next message in buffer
        End
    End
```

4.3.2 Version 2: New Replicated Data Structures

The implementation of Version 2 closely resembles the structure of Version 1, building upon its core components. Version 2's main difference includes the addition of `CommitStatus` to the state of each process, introducing three new variables:

- `NextCommit`: an integer initialized to 0;

- `MaxCommit`: an integer initialized to 1;
- `Bitmap`: an array of bytes, with its size equal to one-eighth of the system's size, initialized with zeros, where each bit corresponds to a specific process in the system.

The updating and exchange of these structures follow the related logic outlined in Figure 12. In Version 2, the leader can also receive `AppendEntries` requests to advance its own structures and identify new committed entries efficiently.

4.4 Parameters

The Raft and Driftwood implementations have certain parameters that must be thoughtfully configured based on the specific system properties, the number of replicas, network characteristics, and computational resources.

The election timeout value should be randomly selected from a predefined interval with specific minimum and maximum parameters. To prevent split votes, it's crucial to set a longer interval when there are a larger number of replicas, reducing the likelihood of multiple followers initiating elections simultaneously.

The heartbeat timeout is another parameter to consider. It should be set to a value smaller than the minimum election timeout to prevent unnecessary elections caused by message loss or delays. However, setting it to an excessively small value is also undesirable, as it can negatively impact performance.

Driftwood also includes another timeout, referred to as the round timeout, which dictates the interval between initiating gossip rounds. The chosen value must account for the time it takes for messages to be propagated to all processes in order to avoid a situation where messages from subsequent rounds arrive at certain replicas before those from previous rounds. Such a scenario could lead to unsuccessful replication causing the necessity of individual log replication, which should be avoided to decrease leader's workload.

Related to gossip, we must also take into account the fanout value. A commonly chosen value is the natural logarithm of the system's size, as it balances fault tolerance and rapid message delivery.

Theoretically, the Raft algorithm considers that the message has no maximum size, allowing the log entries in `AppendEntries` to grow infinitely. However, this is not practically viable, so we define a maximum limit for the number of entries sent in the message. The higher this value, the better the performance under high workloads.

Chapter 5

Experimental Evaluation

This chapter discusses the experimental evaluation of Raft and the two versions of Driftwood. We refer to the servers as the set of processes that constitute the replicated system. In other words, each server corresponds to one replica instance. Similarly, we define a client as a single thread of the Paxi client responsible for generating read and write requests to the servers.

We start by describing the experimental setup in Section 5.1. Then, in Section 5.2, we explain our reasoning for the parameters chosen for the consensus algorithm. Section 5.3 evaluates how replication scales, including Paxos for comparative analysis with our Raft implementation, by examining their performance with an increasing numbers of replicas. Section 5.4 assesses the algorithms' performance in a highly replicated system (125 replicas) by analysing them under increasing client request generation. Section 5.5 evaluates the impact of concurrent client requests on the system's performance. Section 5.6 examines resource usage, particularly CPU utilization, by the leader and followers to determine the point at which the leader becomes the system's bottleneck. Section 5.7 analyses the commitment latency between the leader and followers. Section 5.8 analyses how the algorithms behave in simulated networks with randomized partial partitions.

5.1 Experimental Setup

Each experiment involves Paxi servers (process replicas) and a Paxi client used for benchmarking the system. All experiments were conducted on a single machine equipped with 128 CPUs. Detailed CPU architecture information is provided in Table 2. Each Paxi server is allocated a dedicated CPU to minimize variations in results due to resource sharing.

Paxi servers and the client use the local loopback network for inter-process communication via the TCP/IP protocol, resulting in virtually zero message latency. Message (de)serialization uses the Golang Gob library [7]. To limit the size of `AppendEntries` messages, we capped the maximum number of

Architecture	aarch64
Byte Order	Little Endian
CPU(s)	128
Thread(s) per core	1
Core(s) per socket	64
Socket(s)	2
CPU max MHz	2600
CPU min MHz	200
BogoMIPS	200
L1d cache	64K
L1i cache	64K
L2 cache	512K
L3 cache	65536K

Table 2: CPU architecture information

entries sent at 100 per message.

The Paxi client serves a dual role in our experiments: generating workloads and collecting metrics for benchmarking. During execution, the client generates requests based on the predefined configuration. It can operate with multiple threads, enabling concurrent request submission, with or without a specified throttle parameter. These threads can function in one of two ways: after sending a request, the thread will either wait indefinitely for a response, or it will wait for a defined period of time for the response before resending the request to avoid scenarios where the thread can block due to the server failing to reply. In most experiments, we used blocking threads, as the requests are typically expected to reach a correct leader and the Paxi client employs ten threads to concurrently issue requests to the system.

To assess the new replication mechanisms, the servers are initially configured with a designated leader, allowing us to focus solely on the replication phase of the algorithms. As a result, elections do not occur, given that faults are not expected to occur. The exception to this is when we analyse the behaviour of the algorithms in non-transitive networks.

We conducted each experiment three times, using the same parameters, and then aggregated the resulting metrics by calculating the mean. The client issues requests for one minute, with the servers being initiated anew for each experiment. One minute sufficed for benchmarking most systems, as they remained stable with a correct leader operating consistently. However, for systems where the leader's stability was in question, which could lead to random halts due to frequent elections, we extended the experiment duration to two minutes. This extension was particularly essential when testing Raft in partially partitioned networks, where we observed notable performance variability in some scenarios.

5.2 Servers Parameters

The parameters for server settings, including election and heartbeat timeouts, were kept consistent across all algorithms since they use the same election procedure. These values were selected with consideration for a system size of up to 125 server replicas. The election timeout ranged from a minimum of 500 ms to a maximum of 5000 ms, while the heartbeat timeout was set at 50 ms.

For Driftwood, we needed to choose values for the leader's round timeout and gossip's fanout, both of which impact the overall system performance. The round timeout value directly affects the leader's CPU utilization. A higher value can cause the leader to remain idle for extended periods, increasing request latency, while smaller values prompt the leader to expend more CPU resources due to increased message exchange frequency. The selection of the round timeout value should take into account network message latency to ensure that gossiped messages have sufficient time to be delivered to all followers. This mitigates scenarios in which followers might receive messages from higher rounds before lower ones, which could potentially result in unsuccessful log replications. This isn't a concern in our experiments, since message latency is negligible. The selection of the fanout value has a relatively minor influence on the overall system's performance. This is because the necessary value for ensuring reliable gossip delivery is typically small, and thus, higher values have minimal impact on CPU usage.

The values for the round timeout and fanout used are based on extensive experimentation with various configurations, including different system sizes with 10 client threads generating requests concurrently. The configurations with the best performance results for Version 1 and Version 2 can be found in Table 3 and 4, respectively.

#Replicas	Round Timeout (μs)	Fanout	Throughput (req/s)	Mean Latency (ms)
5	300.00	2.00	9837.23	1.01
11	500.00	3.00	7217.69	1.38
19	700.00	4.00	4960.71	2.01
31	900.00	4.00	4093.73	2.44
53	1250.00	5.00	2975.93	3.36
77	2000.00	5.00	2127.21	4.70
101	3000.00	5.00	1565.41	6.38
125	3500.00	5.00	1244.87	8.03

Table 3: Best-performing parameters for Version 1

#Replicas	Round Timeout (μs)	Fanout	Throughput (req/s)	Mean Latency (ms)
5	200.00	2.00	6269.36	1.59
11	300.00	3.00	4068.22	2.45
19	400.00	3.00	3408.77	2.93
31	500.00	3.00	2801.67	3.56
53	600.00	3.00	2360.51	4.23
77	700.00	3.00	2109.35	4.74
101	700.00	3.00	1932.47	5.17
125	800.00	3.00	1721.50	5.80

Table 4: Best-performing parameters for Version 2

Version 1 performs better with a higher round timeout value compared to Version 2's, primarily because Version 1's leader must process replication replies from followers. The fanout value used for Version 1 had minimal impact on performance, as the CPU cost of gossiping a message is relatively insignificant. Version 2 obtained better performance with small fanout values due to enabling the `CommitStatus` to propagate through more processes and advance more rapidly.

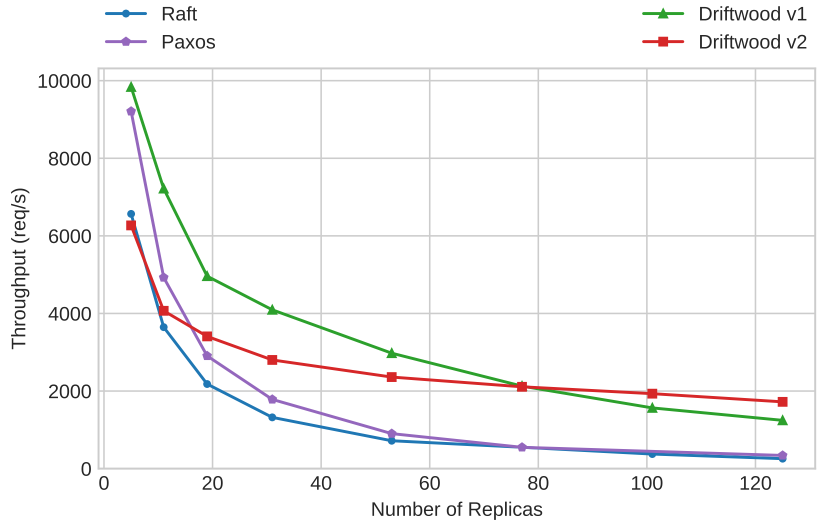
5.3 Replication Scalability

We assess the performance of the algorithms as we increase the number of replicas. Additionally, we also conducted these experiments with the classical Multi-Paxos [10] algorithm implementation to compare it with our Raft implementation.

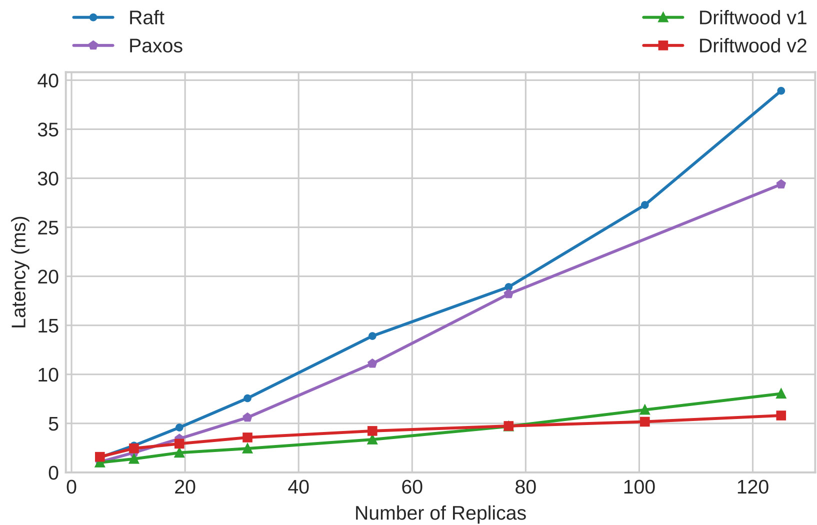
Figure 14 shows the results of our experiments. We can easily understand that Raft scales poorly as throughput decreases and client response time increases with the increasing number of replicas. This also justifies the reason why Raft is best used when a low number of replicas, 3 to 7, are necessary. According to the obtained results, our Raft implementation exhibits a similar behaviour to the Multi-Paxos implementation, albeit with lowers performance.

Both versions of Driftwood outperform Raft, mainly due to the leader efficiently sending log entries for replication in batches in the gossip rounds. For smaller systems with 5 replicas, Version 1 surpasses Raft's performance, reaching 9837.2 req/s, which is 1.5 times higher than Raft. Meanwhile, Version 2's performance is on par with Raft's with 6269.4 req/s, just 0.95 times lower.

As the number of replicas increases, the performance gap between the two versions of Driftwood narrows. With 77 replicas, they both achieve similar performance of about 2100 req/s (3.8 times higher than Raft). However, with a higher number of replicas, Version 2 starts exhibiting better performance than



(a)



(b)

Figure 14: Throughput (a) and Latency (b) with increasing number of replicas

Version 1. For the systems with 125 replicas, Version 1 reaches 1244.9 req/s (4.8 times higher than Raft) and Version 2 reaches 1721.5 req/s (6.7 times higher than Raft).

In terms of latency, Raft exhibits a steeper linear increase, reaching an average response time of 38 ms for systems with 125 replicas. In contrast, both versions of Driftwood experience a much more gradual increase, with an average response time of under 10 ms.

5.4 Performance

To evaluate the algorithms' performance with 125 replicas and 10 clients concurrently issuing requests according to a varying throttle parameter and analyse the mean response times.

We observe, in Figure 15, that Raft quickly reaches its limit at 256.9 req/s with a response time of 38.9 ms when confronted with 10 concurrent clients. However, Version 1 performs significantly better, maintaining a mean response time below 9 ms and achieving 1231.2 req/s. Version 2 further excels with a throughput of 1810.3 req/s and a mean response time below 6 ms.

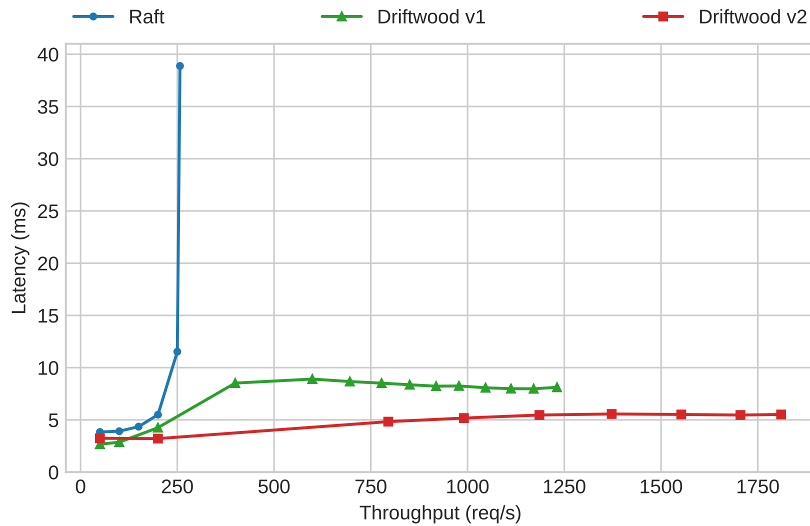


Figure 15: Mean response time according to reached throughput

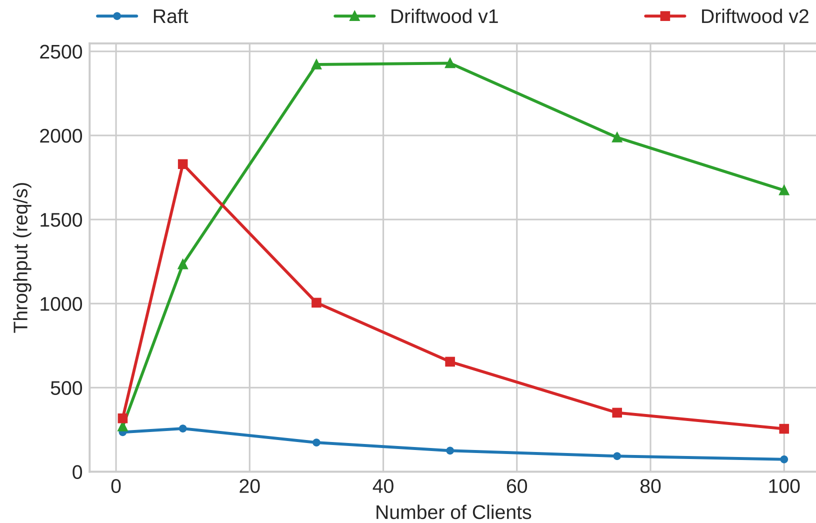
5.5 Impact of Concurrent Clients

In this section, we analyse the performance of the systems with 125 replicas and an increasing number of clients issuing requests to assess the algorithms' scalability under growing workloads.

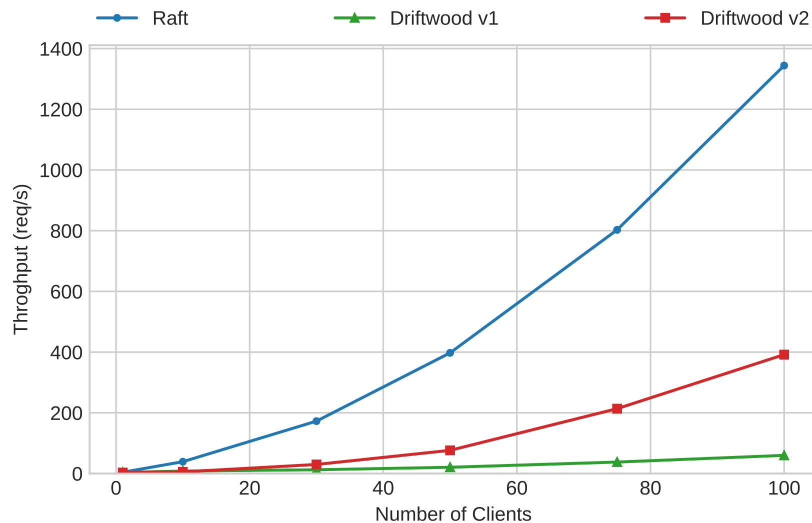
Figure 16a shows that Raft's already low performance diminishes as the number of clients increases. In contrast, the two versions of Driftwood exhibit an initial performance boost, followed by a decrease. Version 1 displays better scalability, reaching 2421.9 req/s and 2429.6 req/s at 30 and 50 clients, respectively, and then experiencing a gradual decline to 1673.6 req/s with 100 clients. On the other hand, Version 2 reaches its peak sooner at 10 clients, hitting 1829.5 req/s, but subsequently experiences a sharp drop, obtaining 255.3 req/s with 100 clients.

Driftwood exhibits better scalability due to its efficient replication of batched entries. However, its performance starts to decline at a certain point. In Version 1, this decline was attributable to the in-

creasing workload on the leader. In contrast, Version 2 experienced a decline earlier, primarily due to a higher number of concurrent requests that resulted in greater disparities between replicas' log positions. These inconsistencies delay the advancement of the `CommitStatus` and subsequently hinder overall performance.



(a)



(b)

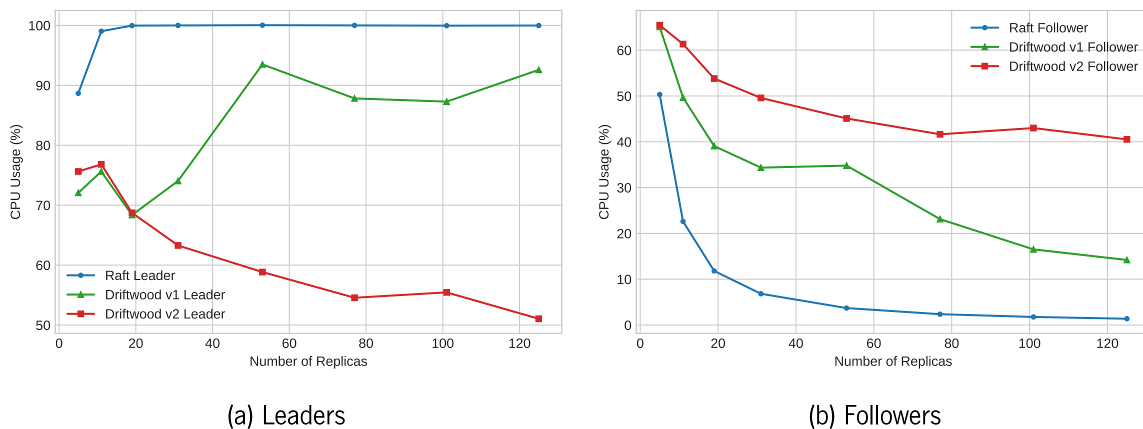
Figure 16: Throughput (a) and latency (b) with increasing number of concurrent clients

5.6 Use of Resources

To demonstrate Driftwood's decentralization of the leader's effort compared to Raft, we measured the CPU usage of each server process. The CPU utilization of each server is measured using the `psutil` Python

library [35] and is shown as a percentage of the dedicated core being utilized.

We start by evaluating the use of CPU by the leader and the mean follower CPU use with increasing numbers of replicas in the system. In Figure 17, we observe notable differences between Raft and Driftwood. Raft's leader rapidly consumes all available CPU resources, becoming the bottleneck of the system, while its followers show decreasing CPU usage as the number of replicas increases. In contrast, Driftwood's leader exhibits a more gradual increase in CPU utilization as the number of replicas grows, and its followers also experience more CPU usage due to participating in the gossip mechanism. In Version 2, the leader's CPU utilization decreases as the number of replicas increases, aligning with the reduction in followers' CPU usage. This can be attributed to the fact that in Version 2, all replicas play an equal role in replication, while the leader also handles the additional task of communicating with clients. Consequently, the leader does not become the bottleneck of the system. Furthermore, Version 2's followers, in comparison to Version 1, exhibit higher CPU usage because they are actively discovering the committed entries.



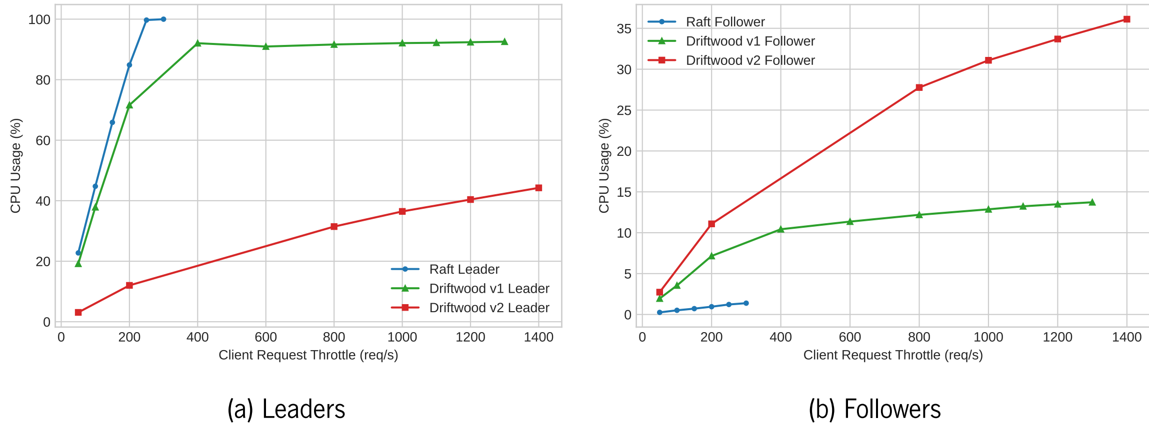
(a) Leaders

(b) Followers

Figure 17: Use of CPU with increasing number of replicas

We also analyse the CPU utilization according to client request throttle in a system with 125 replicas. In Figure 18 it is evident that Raft is a highly centralized system, with the leader using all of its CPU resources, compared to the followers that use almost none.

In Version 1, the leader's CPU usage grows exponentially, reaching its peak of approximately 90% much earlier at a throttle of 400 req/s. In contrast, in Version 2, the leader's CPU usage increases more gradually, reaching a maximum of around 40% at 1400 req/s. We also observe that the followers of Version 2 exhibit significantly higher CPU usage compared to the followers of the other algorithms.



(a) Leaders

(b) Followers

Figure 18: Use of CPU with increasing client request throttle

5.7 Server Commitment Discovery

We compare the commitment discovery times between a follower and the leader, specifically the time it takes for a server to commit an entry from the moment it is added to the leader's log. To achieve this, the leader records the timestamp when it adds entry i to its log and when it successfully commits entry i. Simultaneously, one follower also registers, in a separate file, when the entry i was committed to its log. Figure 19 presents the cumulative distribution function (CDF) of the results.

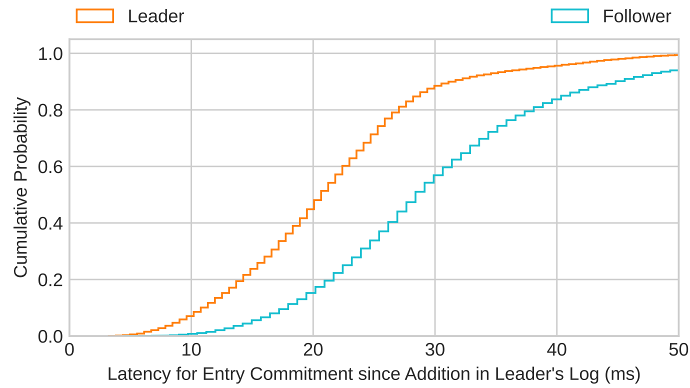
We can observe that in Raft and Version 1, the leader commits entries faster than the followers because it is responsible for advancing the commitment by obtaining acknowledgments. In contrast, Version 2 shows a different pattern, where the leader's commitment advancement is similar to that of the followers. This is because any server can independently discover new committed entries first.

5.8 Performance with Partial Network Partitions

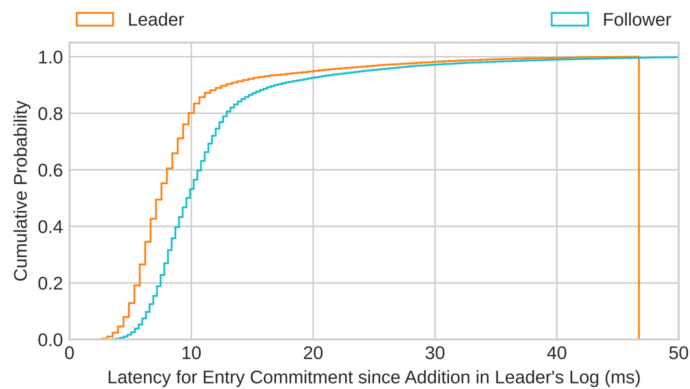
In this section, we analyse the performance and behaviour of the three algorithms in partially partitioned networks. To simulate this scenario, we introduce link failures between nodes, essentially making it impossible for some servers to exchange messages directly. However, for testing purposes, each server can still forward client requests to the server it thinks is the leader to simulate requests being directly sent to the leader, consistent with the algorithms' assumptions.

Numerous elections can potentially result in log entries being deleted, causing clients to become blocked while waiting for the corresponding reply. To avoid this, we alter the clients' behaviour to retry sending the request if a timeout occurs (set to 100 ms).

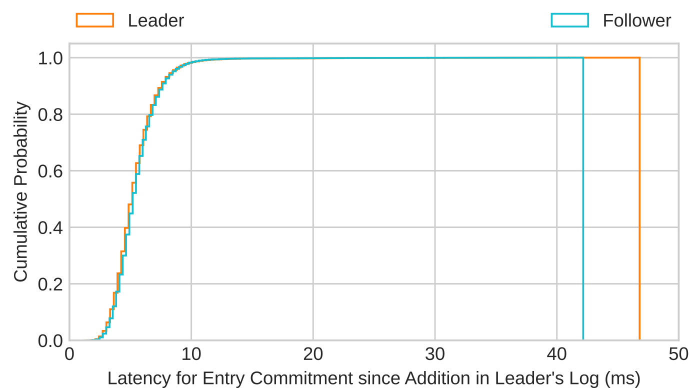
If a follower who is unaware of the current leader or a candidate receives a request, it responds with



(a) Raft



(b) Version 1



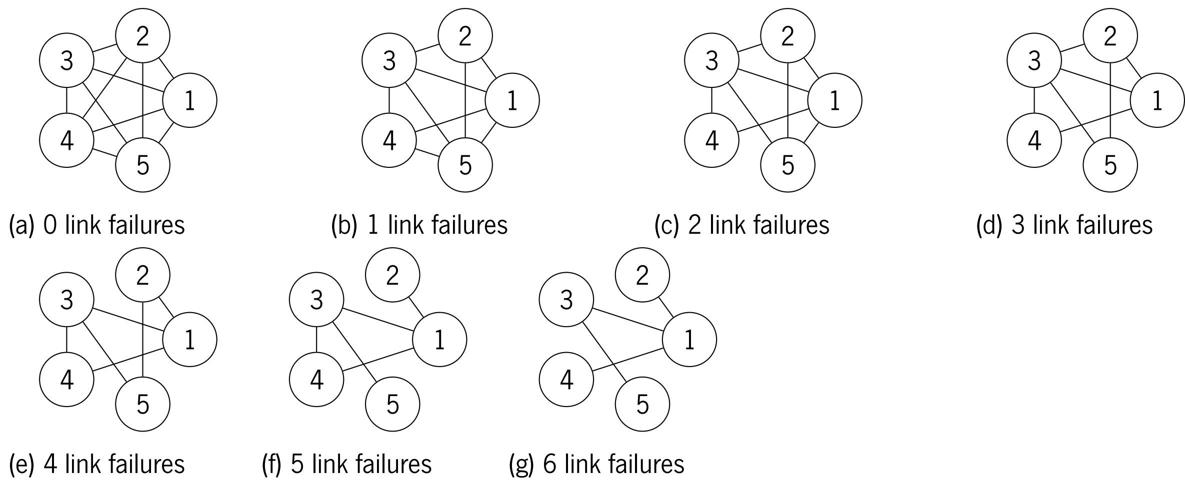
(c) Version 2

Figure 19: CDF of the time interval between a client request received by the leader and committed in leader/follower

an error. The client, upon receiving an error, will wait for 20 ms before attempting to send the request again, allowing time for a leader to emerge.

We ran the experiments with different numbers of link failures and system sizes. Figure 20 illustrates the network topologies simulated, each featuring a distinct number of link failures, for a system with 5 replicas.

Let's begin by examining the behaviour of the 5-replica systems, as illustrated in Figure 21a. In all



(a) 0 link failures (b) 1 link failures (c) 2 link failures (d) 3 link failures

(e) 4 link failures (f) 5 link failures (g) 6 link failures

Figure 20: Network topologies simulated

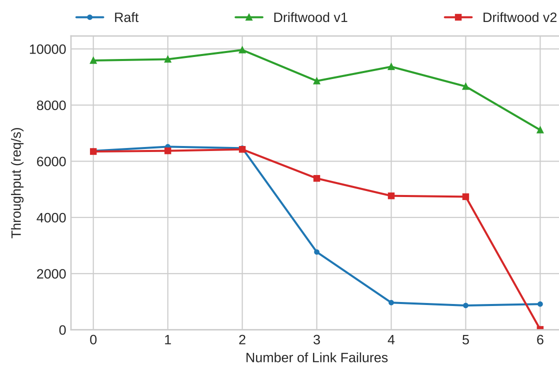
experiments, the process 1 is the first to initiate an election.

We start by discussing Raft's behaviour. In scenarios with 1 or 2 link failures, Process 1 can successfully assume leadership since it maintains communication with all other servers, ensuring the system operates normally; therefore, the performance remains similar to the scenario with 0 link failures. However, in the topology with three link failures, Process 1 becomes the leader by securing a majority of votes. Unfortunately, Process 5 remains unaware that a leader exits and will time out, initiating elections but failing to become the leader due to its outdated log. This cycle repeats until Process 3, the only process capable of communicating with all others, becomes the leader. As a result, we observed a temporary decrease in system performance. In network topologies with more than three link failures, there aren't any processes capable of communicating with all others, which means there is no stable leader possible. Consequently, elections occur constantly, significantly impairing system performance.

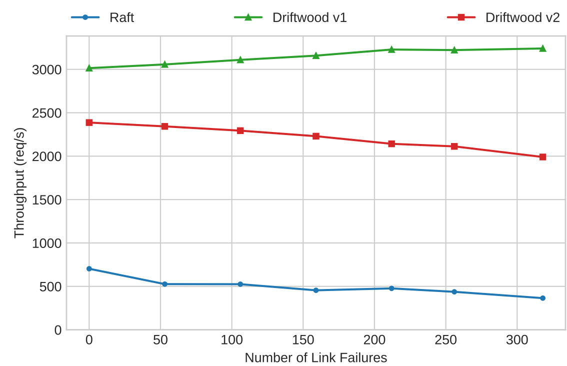
Version 1 shows a small decrease in performance with higher numbers of link failures. This decline can be attributed to the occasional failure of a gossip round to replicate the log for some followers. This, in turn, can lead to requiring various successive gossip rounds for the leader to commit entries.

Version 2 exhibits a more significant decrease in performance, primarily because of reduced exchanges of `CommitStatus`, which subsequently delays the discovery of committed entries by the leader. Moreover, we observed a notable stalling in the system's progress in the network, with six link failures. We discovered that this phenomenon is caused by a liveness restriction. For the `CommitStatus` to advance, it's necessary to have a subgraph within the network that includes a majority of nodes, including the leader, to be fully connected. If this condition isn't met, it's possible for processes to update the `CommitStatus` simultaneously with different `NextCommit` values, preventing any process from obtaining a majority on their `Bitmap`, thus halting progress. For example, in the network with six link failures depicted in Fig. 21,

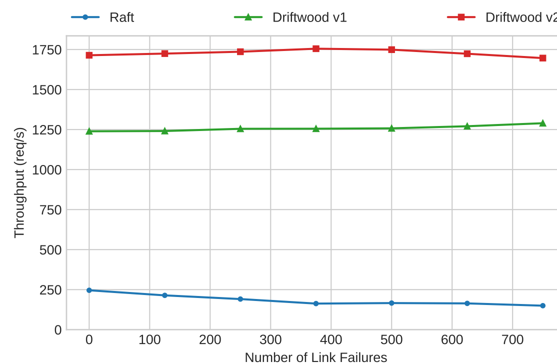
node 5 can fall behind the entries being gossiped and stop replicating the leader's log because it can't request the missing log entries since it's unable to communicate with the leader. As a result, it will be unable to set its bit on the `Bitmap`. Meanwhile, node 1 is the only node capable of exchanging `CommitStatus` messages between nodes 2, 3, and 4. It's possible that the leader updates its `NextCommit` to a higher value while the other nodes update their `NextCommit` to a smaller value. In this scenario, the leader's `Bitmap` won't merge with the `Bitmaps` of the other nodes, and the nodes themselves will only merge with node 1's `Bitmap`. As nodes 2, 3, and 4 can't communicate with each other, the `Bitmaps` will never show a majority, and the `CommitStatus` won't advance.



(a) 5-Replica Systems



(b) 53-Replica Systems



(c) 125-Replica Systems

Figure 21: Performance of systems with 5, 53 and 125 replicas under varying link failures

For systems with 53 and 125 replicas, we intentionally use a varying number of link failures that is divisible by the number of replicas. Raft's performance consistently decreases. However, Version 1's performance remains stable and even slightly increases with a higher number of link failures due to a decrease in message redundancy. In Version 2, the performance once again decreases with increasing link failures in systems with 53 replicas. In contrast, in systems with 125 replicas, the performance remains consistent due to the higher number of nodes participating in the exchange of the `CommitStatus`.

Chapter 6

Conclusion

Driftwood presents a new way to efficiently replicate data in distributed systems. The addition of gossip enhanced fault tolerance and allows partial networks to occur without substantial performance degradation. Simultaneously, the leader's workload in log replication was effectively decentralized through the followers, which resulted in an overall performance increase as we expected. Since Driftwood is built upon Raft, only modifying the process by which the leader replicates its log entries and introducing a novel approach to discovering committed log entries, it inherits the core understandability associated with Raft. Thus, if someone can grasp Raft, comprehending Driftwood becomes a relatively straightforward process.

In this thesis, we present two versions of Driftwood, differentiated by the way they make commitment progress. Both versions use gossip mechanisms to decentralize the leader's workload on log replication. The leader periodically gossips about uncommitted entries. In cases where a follower, due to missing messages, falls behind in replicating committed entries, it directly requests the missing entries from the leader. Version 1's leader discovers which entries are committed the same way as in Raft, by receiving the replies from all followers with information about the replication's success. On the other hand, Version 2's processes exchange, through the gossip, a replicated data structure that we called `CommitStatus`, with compact information about every process's log replication status on the current term. Every process can advance the `CommitStatus` and determine which entries are committed. To accurately advance `CommitStatus` we present two functions: one for merging with a `CommitStatus` received from another process, and another for updating it when new entries can be committed.

6.1 Driftwood vs. Raft

The evaluation results for Version 1 reveal that the leader can more effectively replicate its log entries through the followers by utilizing periodic gossip rounds. The performance is optimized by periodically replicating uncommitted entries in batches, in contrast to our Raft implementation, where the leader sends

an individually created messages containing the missing entries of the recipients to all followers whenever a new entry is appended to its log. In a system with 5 replicas, Version 1 achieved a throughput of 9837 req/s, outperforming Raft by 1.4 times. It's worth noting that this performance boost can be attributed to the presence of multiple clients requesting concurrently. In a scenario with a single client, the mean latency can be determined by the gossip round interval. However, as the number of concurrent requests increases, Version 1 demonstrates an efficient method of batching. As demonstrated in the evaluation, in a system with 125 replicas, the throughput started at 269 req/s with a single client. As the number of clients increased to 50, the throughput reached its peak at 2430 req/s before gradually receding to 1674 req/s at 100 clients.

Version 1's performance exhibits a more gradual decline as the number of replicas increases, compared to Raft, due to the decentralization of the replication process through gossip. Specifically, with 5 replicas, Version 1 achieved a throughput of 9837 req/s, which then decreased to 1245 req/s with 125 replicas. This represents a throughput reduction of 7.9 times, significantly less compared to Raft, which experienced a 26-fold reduction. Additionally, Version 2 presented better replication scalability by advancing commitment in a decentralized manner. This meant that the leader no longer had to gather the followers' replication success responses, reducing his workload and allowing him to reach its limit at a later stage. With 77 replicas, both versions of Driftwood demonstrated similar throughput. However, as the number of replicas increased, Version 2 began to surpass Version 1 in terms of throughput. Specifically, with 5 replicas, Version 2 achieved 6269 req/s, decreasing to 1721 req/s at 125 replicas, resulting in a 3.6-fold decrease.

The addition of periodic gossip rounds allowed the leader to decentralize its replication workload across all followers, as was intended. Unlike Raft, Driftwood's followers actively participate in log replication through gossip of messages with uncommitted entries, rather than the leader solely replicating log entries to all followers. In Driftwood, the leader's CPU usage is based mainly on the round interval parameter. In a system with 125 replicas, we observed that Raft's leader rapidly reached its CPU usage limit, becoming the bottleneck of the system, while Version 1's leader was able to handle the 10 clients without exhausting its CPU. We also observe that Version 1's followers also use more CPU due to participating in the gossip. In the case of Version 2, the leader consumes less CPU because it doesn't need to collect replication success replies from followers to discover committed entries, while the followers do more substantial work due to more frequent gossip rounds and advancing the `CommitStatus`.

Gossip serves a dual purpose in our system. Beyond its role in log replication, it also plays a crucial role in disseminating heartbeats across the network. This heartbeat dissemination is vital for preventing

unnecessary elections caused by link failures and improving the system reliability in partially connected networks. We observed that Version 1 can tolerate a substantial number of link failures due to gossip delivery reliability. The evaluation further demonstrates that Version 1 not only sustains its performance in partially connected networks, but can even enhance it due to the leader receiving fewer followers' replication replies. However, in Version 1, the leader still needs to be capable of communicating with a majority of followers to receive a corresponding majority of replication responses and ensure the progress of the commitment process, which in turn guarantees system liveness. Moreover, in both versions of Driftwood, for the leader to replicate its log to a majority of followers, it needs to be correctly connected to a majority of followers, as the gossip can fail to deliver replication messages and followers might need to request missing entries directly to the leader. This requirement applies to both versions of Driftwood, as gossip replication doesn't guarantee message delivery, and consequently, there's the possibility that followers may need to directly request the leader for missing entries in its log.

Version 2 of Driftwood demonstrates that log entries for both the leader and followers are committed at a consistent rate, a consequence of the decentralized commitment advancement process. This stands in contrast to Raft and Version 1, where the leader first discovers and commits log entries and then informs the followers to do the same. A low follower commitment latency opens up the possibility for clients to opt for relaxed consistency guarantees, allowing them to directly make read requests to any replica, thereby reducing client response latency.

6.2 Driftwood: Aspects to Improve

The Driftwood algorithm we've developed offers a promising alternative to Raft. It upholds Raft's safety rules and election procedure while modifying the leader replication process. Nevertheless, there are aspects that can be further improved by introducing new mechanisms and making necessary alterations that could potentially enhance both performance and fault tolerance.

We will begin by addressing the round interval parameter, which significantly impacts the system's performance. In our experiments, we set the round interval to an optimal value for performance. However, this approach can be less suitable for real-world systems due to their inherent unpredictability.

In Appendix A.1, we can observe how the round interval and fanout parameters impact the performance within a system comprising 125 replicas for Version 1 with 10 clients. Throughout our experiments, we observed that systems with a greater number of replicas tend to benefit from a higher round interval value, particularly in the case of Version 1. In this version, the leader's CPU resources are more strained

due to the processing of followers' replies. In contrast, Version 2 distributes the replication workload more evenly across processes. This means the leader has greater flexibility when starting gossip rounds. Additionally, the increased use of gossip rounds in Version 2 contributes to faster advancements of the `CommitStatus`.

In real-world scenarios, the number of incoming requests can fluctuate over time, in contrast to our simulated benchmark, where it has a constant maximum. Opting for a higher round interval value can result in larger batch sizes and longer client latencies, even if the leader has ample resources to initiate more frequent gossip rounds. Conversely, selecting a value that's too small may lead to the leader spending more time processing followers' replies than handling client requests, leading to higher latency. To address this challenge, a dynamic round interval can be implemented, adjusted based on the leader's CPU utilization. This ensures that the CPU is neither fully utilized with too small a value nor underutilized with a value that's too high, thereby optimizing resource allocation for replication.

The gossip round in Driftwood employs an eager push, where the entire message payload is transmitted. However, this method has a significant impact on bandwidth usage. An alternative approach is known as lazy push, where processes send a message advertisement, not the complete payload, and if the recipient hasn't received it yet, they make an explicit pull request.

In the context of Driftwood, most of the gossip message's payload consists of batched log entries. It's possible that some or all of these entries are redundant because the recipient may have already replicated them. Instead of transmitting the full batch, the payload could be optimized by sending only the indices and terms of the first and last entries. This way, the recipient can check whether their log already contains these entries. If they do, the leader will gossip the message when it's received for the first time. If not, it makes a pull request for the missing entries.

The pull mechanism further decentralizes the algorithm from the leader, decreasing the frequency with which the leader has to replicate the log directly with followers. Instead, followers with updated logs can replicate the current leader's log entries for those with logs lagging behind.

In Version 1, similar to Raft, a process can only become the leader if it can communicate with the majority of other processes. However, if there are link failures and the leader loses communication with a majority of followers, it cannot receive a majority of replication replies and commit entries. Nevertheless, the leader continues to gossip heartbeats to maintain its leadership status. In this situation, it's possible that a follower is capable of communicating with the majority of processes and could potentially become a stable leader. To address this issue, the leader requires a failure detector to recognize whether it can communicate with the majority of followers. If it detects this, it will transit to the follower state and await

the election of a new leader.

The Raft election procedure could also be enhanced by incorporating gossip. Instead of candidates individually soliciting votes from each process, they could disseminate a gossip message containing a bitmap that collects votes during exchanges. When the number of votes in the bitmap reaches a majority, a direct message is sent to the candidate to notify it of its election victory. This approach revises the election liveness constraint [15], which in Raft requires the existence of a process with an up-to-date log and the ability to communicate with a majority to ensure a leader appears. With this new mechanism, a leader can be elected even if a process is unable to communicate with the majority of processes. It's worth noting that the leader of Version 1 might encounter issues in replicating the log to a majority of followers and in committing new entries if it is unable to communicate directly with a majority of the processes, as discussed earlier.

We initially anticipated that Version 2 would not encounter the above problem due to its fully decentralized commitment discovery process. However, we discovered that the developed `Merge` function does not consistently converge and advance in certain scenarios involving partially connected networks. To address this issue, we need to modify the `Merge` function to make it asymmetric, ensuring convergence while preserving the pre- and post-conditions. An alternative approach involves replacing the `CommitStatus` in the current state with the `CommitStatus` received if it has a `NextCommit` value either lower (but higher than `MaxCommit`) or higher than the current `NextCommit`.

Version 2 exhibited lower performance than expected, primarily because the `CommitStatus` advanced too slowly within the gossip mechanism. To address this, we enhanced performance by increasing the frequency of gossip rounds. Another potential alternative solution could involve increasing the number of times the gossip message is forwarded by the followers or implementing a time-to-live mechanism in the gossip protocol, thereby encouraging more frequent message exchanges. This would differ from the current behaviour, where followers stop forwarding the gossiped message after receiving it for the first time. Furthermore, the efficiency of `CommitStatus` in advancing commitment may not meet the initial expectations, primarily due to the `Merge` function. The alterations mentioned earlier to this function could potentially enhance its performance.

6.3 Final Remarks

In this thesis, we present two versions of Driftwood that significantly improve Raft. These versions effectively decentralize the leader's replication workload among the followers and enhance fault tolerance,

particularly by improving liveness in partially connected networks. Both versions have shown promising results in various aspects when compared to Raft, all without significantly increasing the overall algorithm's complexity.

Driftwood retains most of Raft's logic, which provides a strong foundation for its reliability. Nevertheless, the novel replication mechanisms of Driftwood introduce an element of unpredictability. While we tried to maintain consistency with Raft's principles, the intricacies of these new mechanisms require a closer examination. Formal verification is essential to confirm that these changes do not unknowingly undermine Raft's safety properties.

The algorithms were evaluated under normalized conditions, where the latency between processes was basically negligible. Such environments are seldom encountered in real-world scenarios, emphasizing the need to conduct further analysis to ascertain how Driftwood performs under more realistic conditions.

Bibliography

[1] Replication layer | cockroachdb docs. https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html. Retrieved December 2, 2022.

[2] Raft and high availability. https://www.pingcap.com/blog/raft-and-high-availability/. Retrieved December 2, 2022.

[3] Data-centric application development toolkit for kubernetes. https://atomix.io/.

[4] Cokcroachdb. https://www.cockroachlabs.com/.

[5] Consul. https://www.consul.io/.

[6] Etcd: Raft library. https://github.com/etcd-io/etcd/blob/main/raft/README.md. Retrieved December 2, 2022.

[7] Gob: The go binary data encoder/decoder. https://golang.org/pkg/encoding/gob/. Accessed on 2023-09-15.

[8] Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[9] M3: Uber's open source, large-scale metrics platform for prometheus. https://www.uber.com/en-PT/blog/m3/.

[10] *Paxi* framework. https://github.com/ailidani/paxi. Accessed: 2023-01-5.

[11] Trillian: General transparency. https://github.com/google/trillian/.

[12] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Dissecting the performance of strongly-consistent replication protocols. *Proceedings of the 2019 International Conference on Management of Data*, 2019. URL https://api.semanticscholar.org/CorpusID:195259350.

[13] Peter Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering*, 1976. URL https://api.semanticscholar.org/CorpusID:263865061.

[14] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

[15] Qihao Bao, Bixin Li, Tianyuan Hu, and Dongyu Cao. Model checking the safety of raft leader election algorithm. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 400–409, 2022. doi: 10.1109/QRS57517.2022.00048.

[16] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-paxos: Offloading the leader for high throughput state machine replication. *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120, 2012. URL https://api.semanticscholar.org/CorpusID:2957025.

[17] Daniel Cason, Nenad Milosevic, Zarko Milosevic, and Fernando Pedone. Gossip consensus. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 198–209, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385343. doi: 10.1145/3464298.3493395. URL https://doi.org/10.1145/3464298.3493395.

[18] Konstantinos Choumas and Thanasis Korakis. When raft meets sdn: How to elect a leader over a network. *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 140–144, 2020. URL https://api.semanticscholar.org/CorpusID:221120541.

[19] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8:85–110, 1990. URL https://api.semanticscholar.org/CorpusID:15404410.

[20] Brett D. Fleisch. The chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006. URL https://api.semanticscholar.org/CorpusID:1617028.

[21] Google. The go programming language. https://golang.org/, 2018.

[22] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. 1994. URL https://api.semanticscholar.org/CorpusID:13974342.

[23] HashiCorp. Consensus protocol | raft. https://developer.hashicorp.com/consul/docs/architecture/consensus, 2022.

[24] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL https://doi.org/10.1145/78969.78972.

[25] Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distributed Syst.*, 14:248–258, 2003. URL https://api.semanticscholar.org/CorpusID:260460027.

[26] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Comput. Networks*, 2:95–114, 1978. URL https://api.semanticscholar.org/CorpusID:3960146.

[27] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

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

[29] Leslie Lamport. Generalized consensus and paxos. 2005. URL https://api.semanticscholar.org/CorpusID:18985767.

[30] Tom Lianza and Chris Snook. A byzantine failure in the real world. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/. Accessed: 2022-12-20.

[31] Alberto Montresor. Gossip and epidemic protocols. 2017. URL https://api.semanticscholar.org/CorpusID:198939240.

[32] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[33] J. Pereira and R. Oliveira. The mutable consensus protocol. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 218–227, 2004. doi: 10.1109/RELDIS.2004.1353023.

[34] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure data lake store: A hyper-scale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 51–63, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3056100. URL https://doi.org/10.1145/3035918.3056100.

[35] Giampaolo Rodola. Cross-platform lib for process and system monitoring in python. https://pypi.org/project/psutil/, 2022.

[36] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990. ISSN 0360-0300. doi: 10.1145/98163.98167. URL https://doi.org/10.1145/98163.98167.

[37] Fred B. Schneider. Chapter 7 : Replication management using the state machine approach. 1993. URL https://api.semanticscholar.org/CorpusID:9549911.

[38] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319942. doi: 10.1145/2465351.2465386. URL https://doi.org/10.1145/2465351.2465386.

[39] Pasindu Tennage, Antoine Desjardins, and Eleftherios Kokoris-Kogias. Mandator and sporades: Robust wide-area consensus with efficient request dissemination. *ArXiv*, abs/2209.06152, 2022. URL https://api.semanticscholar.org/CorpusID:252211927.

[40] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741964. URL https://doi.org/10.1145/2741948.2741964.

[41] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. Paxosstore: High-availability storage made practical in wechat. *Proc. VLDB Endow.*, 10(12):1730–

1741, aug 2017. ISSN 2150-8097. doi: 10.14778/3137765.3137778. URL https://doi.org/10.14778/3137765.3137778.

Appendix A
Details of Results

In this Appendix, we provide detailed results that were used to create the graphs presented in Section 5. These metrics are abbreviated and explained as follows:

- Replicas: Number of replicas in the system, which is equivalent to the number of servers (server configuration);
- Fanout: Fanout parameter used in gossip (server configuration);
- Interval: parameter that determines the time interval between starting gossip rounds by the leader, in microseconds (μs);
- Clients: Number of concurrent clients issuing requests (client configuration);
- Throttle: Client request throttle setting (client configuration);
- Throughput: Measured throughput;
- Mean: Average latency measured;
- Min: Minimum response time obtained for a request;
- Max: Maximum response time obtained for a request;
- P95: 95th percentile of latency;
- P99: 99th percentile of latency;
- Leader CPU: CPU usage of the leader process;
- Follower CPU: CPU usage of a follower process.

A.1 Parameters

Interval	Fanout	Throughput	Mean	Min	Max	P95	P99
3500	3	1278.39	7.82	2.04	141.98	18.97	27.32
	4	1258.73	7.94	1.90	144.33	19.22	27.32
	5	1230.64	8.12	1.93	134.45	19.67	28.54
4000	3	1209.93	8.26	2.08	151.60	18.48	25.98
	2	1208.55	8.27	2.43	158.31	18.47	27.47
	4	1199.35	8.33	2.04	176.59	18.73	26.72
	5	1198.86	8.33	2.00	144.99	19.05	27.64
4500	5	1137.69	8.78	2.02	138.82	18.39	27.90
3000	4	1129.81	8.85	2.29	159.43	21.58	30.28
3500	2	1126.86	8.87	2.80	143.68	19.66	29.77
5000	5	1101.84	9.07	1.96	160.15	18.52	29.32
3000	5	1098.76	9.09	2.41	149.46	22.06	31.93
	2	1091.92	9.15	2.80	144.30	20.66	30.71
	3	1086.30	9.20	2.39	123.05	21.15	30.21
5500	5	1051.44	9.50	1.99	167.37	18.31	29.01
2500	2	1040.77	9.60	2.59	138.70	21.82	31.46
6000	5	1020.89	9.79	1.97	168.52	18.08	29.69
2500	3	993.71	10.06	2.44	155.96	22.73	31.50
	4	961.22	10.40	2.44	154.17	23.90	33.78
2000	2	927.09	10.78	2.60	153.28	23.59	34.23
2500	5	925.83	10.79	2.52	149.92	24.80	37.86
2000	3	905.25	11.04	2.36	141.38	24.72	35.44
	4	862.55	11.59	2.52	152.25	26.21	39.03
1500	2	853.15	11.72	2.79	143.19	25.59	37.46
2000	5	817.10	12.24	2.38	145.00	28.21	42.96
1500	3	813.78	12.29	2.60	147.28	28.10	41.05
	4	765.35	13.06	2.51	155.70	30.90	46.16
	5	707.55	14.13	3.25	156.70	34.29	52.50
1000	5	558.53	17.90	3.68	205.89	43.32	63.77
500	5	203.91	48.84	4.88	191.86	82.02	99.45

Analysis of Optimal Parameters for Version 1 in a System with 125 Replicas and 10 Clients

A.2 Replication Scalability

The following experiments were conducted with 10 clients simultaneously issuing requests, with each client immediately sending another request upon receiving a response (i.e. throttle was set to 0).

Replicas	Requests	Throughput	Mean	Min	Max	P95	P99
5	552599.33	9209.98	1.06	0.19	57.00	1.40	2.70
11	295601.33	4926.66	2.02	0.33	25.68	2.68	3.57
19	174362.00	2906.02	3.44	0.51	21.78	4.62	5.57
31	107075.00	1784.57	5.60	1.19	24.75	7.64	8.88
53	54161.67	902.65	11.10	2.18	30.05	14.82	16.78
77	32986.00	549.74	18.19	2.96	45.63	23.95	26.36
125	20426.50	340.42	29.38	4.81	145.39	40.56	45.42

Replication Scalability: Paxos Results

Replicas	Requests	Throughput	Mean	Min	Max	P95	P99
5	394164.00	6566.40	1.52	0.20	169.96	2.27	14.35
11	218820.33	3646.57	2.74	0.34	91.39	4.61	13.94
19	130858.67	2180.95	4.58	0.66	87.34	8.00	14.40
31	79305.33	1321.74	7.56	1.06	33.56	14.40	20.35
53	43132.67	718.85	13.91	1.85	48.91	25.55	34.02
77	33068.67	551.14	18.91	2.59	65.17	33.64	44.00
101	22552.67	375.84	27.28	3.14	75.62	45.91	56.56
125	15412.33	256.93	38.92	5.50	91.46	60.74	70.33

Replication Scalability: Raft Results

Replicas	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
5	2.00	300.00	9837.23	1.01	0.23	341.16	1.48	8.86
11	3.00	500.00	7217.69	1.38	0.32	85.08	2.25	11.01
19	4.00	700.00	4960.71	2.01	0.38	180.80	2.96	10.62
31	4.00	900.00	4093.73	2.44	0.55	120.37	3.41	10.83
53	5.00	1250.00	2975.93	3.36	0.83	244.88	6.62	15.46
77	5.00	2000.00	2127.21	4.70	1.17	79.72	10.33	18.07
101	5.00	3000.00	1565.41	6.38	1.57	108.74	14.34	21.04
125	5.00	3500.00	1244.87	8.03	1.93	204.44	19.30	28.14

Replication Scalability: Version 1 Results

Replicas	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
5	2.00	200.00	6269.36	1.59	0.28	385.60	2.96	13.17
11	3.00	300.00	4068.22	2.45	0.40	63.02	6.40	16.76
19	3.00	400.00	3408.77	2.93	0.59	105.00	6.58	15.40
31	3.00	500.00	2801.67	3.56	0.93	103.27	7.31	14.76
53	3.00	600.00	2360.51	4.23	1.24	95.11	7.91	14.11
77	3.00	700.00	2109.35	4.74	1.65	105.09	8.47	15.02
101	3.00	700.00	1932.47	5.17	1.78	78.43	9.29	15.14
125	3.00	800.00	1721.50	5.80	1.95	212.01	10.47	16.57

Replication Scalability: Version 2 Results

A.3 Performance

The following experiments were performed with a system size consisting of 125 replicas, with a fanout of 5 and for Version 1 and 2 for Version 2. Additionally, 10 clients simultaneously issued requests with a varying throttle.

Throttle	Requests	Throughput	Mean	Min	Max	P95	P99
50	3001.67	50.02	3.84	3.34	13.02	4.69	9.37
100	6001.33	100.01	3.92	3.22	22.28	5.89	10.66
150	9001.67	150.01	4.35	3.11	28.87	10.12	18.63
200	12001.33	200.02	5.50	3.08	38.03	16.92	26.00
250	15001.00	250.02	11.53	3.14	61.57	30.29	41.42
300	15417.33	256.93	38.88	4.22	87.79	61.39	70.76

Performance: Raft Results

Throttle	Requests	Throughput	Mean	Min	Max	P95	P99
0	73873.50	1231.15	8.12	1.91	200.91	19.88	28.92
50	3000.33	50.01	2.67	1.98	109.95	2.65	12.46
100	6000.00	99.99	2.86	1.90	128.92	5.50	13.79
200	12000.50	199.99	4.25	1.81	122.46	14.90	20.63
400	23974.50	399.56	8.52	1.87	172.89	23.74	30.81
600	35931.00	598.84	8.91	1.80	146.18	23.95	32.11
700	41749.00	695.75	8.67	1.84	139.58	23.34	31.94
800	46671.50	777.82	8.52	1.80	137.25	22.55	32.19
900	51029.00	850.47	8.36	1.84	139.25	22.17	31.79
1000	55126.50	918.73	8.23	1.83	164.40	21.90	31.67
1100	58671.00	977.83	8.25	1.84	154.16	21.71	32.09
1200	62799.50	1046.58	8.07	1.84	169.04	21.17	31.99
1300	66731.50	1112.17	7.99	1.86	126.62	20.76	30.53
1400	70255.50	1170.82	7.98	1.88	149.00	20.30	29.80

Performance: Version 1 Results

Throttle	Requests	Throughput	Mean	Min	Max	P95	P99
0	108620.33	1810.30	5.52	1.96	134.34	9.55	16.97
50	3001.67	50.01	3.24	1.62	99.81	4.36	5.22
200	12000.00	200.00	3.21	1.63	127.37	4.36	5.48
800	47707.67	795.11	4.83	1.51	144.50	7.51	14.39
1000	59441.33	990.68	5.17	1.55	161.05	8.49	14.48
1200	71142.33	1185.69	5.47	1.60	143.98	9.10	14.99
1400	82350.00	1372.48	5.57	1.68	145.80	9.36	15.29
1600	93134.33	1552.21	5.52	1.83	154.87	9.27	14.92
1800	102308.00	1705.12	5.46	1.87	173.88	9.28	15.14

Performance: Version 2 Results

A.4 Impact of Concurrent Clients

The following experiments were performed with a system size consisting of 125 replicas, with a fanout of 5 for Version 1 and 3 for Version 2.

Clients	Throughput	Mean	Min	Max	P95	P99
1	235.23	4.24	3.32	25.08	5.08	14.63
10	256.84	38.93	4.52	98.22	60.77	70.50
30	173.59	172.57	27.35	307.96	212.78	243.56
50	125.30	397.47	88.20	708.04	538.51	593.17
75	92.80	802.72	181.51	1380.75	1085.60	1177.03
100	73.53	1344.02	202.51	2699.56	1893.27	2139.08

Impact of Concurrent Clients: Raft Results

Clients	Throughput	Mean	Min	Max	P95	P99
1	269.46	3.71	1.93	93.68	8.13	18.32
10	1233.30	8.11	1.94	145.81	19.80	28.28
30	2421.91	12.39	2.28	179.85	29.18	39.95
50	2429.58	20.58	2.42	202.37	40.84	58.46
75	1988.03	37.74	2.50	347.74	74.11	109.53
100	1673.58	59.80	2.66	609.26	126.97	200.84

Impact of Concurrent Clients: Version 1 Results

Clients	Throughput	Mean	Min	Max	P95	P99
1	317.52	3.14	1.53	153.96	4.35	5.26
10	1829.52	5.46	2.00	142.63	9.47	16.51
30	1005.25	29.84	3.63	238.57	53.85	83.99
50	654.29	76.37	4.00	465.05	166.51	211.08
75	351.00	213.70	7.40	664.69	409.77	508.65
100	255.29	391.75	7.39	1194.60	660.69	912.40

Impact of Concurrent Clients: Version 2 Results

A.5 Use of Resources

The following experiments were conducted with 10 clients simultaneously issuing requests, with each client immediately sending another request upon receiving a response (i.e. throttle was set to 0).

Replicas	Leader CPU	Follower CPU
5	88.66	50.32
11	99.03	22.61
19	99.96	11.81
31	99.99	6.83
53	100.03	3.70
77	100.00	2.37
101	99.96	1.76
125	99.98	1.37

Use of Resources: Raft Results

Replicas	Fanout	Interval	Throughput	Mean	Leader CPU	Follower CPU
5	2.00	300.00	9321.23	1.06	72.06	65.08
11	3.00	500.00	7093.91	1.40	75.61	49.64
19	4.00	700.00	5005.68	1.99	68.36	39.07
31	4.00	900.00	4155.20	2.40	74.04	34.35
53	5.00	1250.00	2998.66	3.33	93.49	34.80
77	5.00	2000.00	2120.76	4.71	87.81	23.12
101	5.00	3000.00	1550.97	6.44	87.28	16.51
125	5.00	3500.00	1227.01	8.14	92.58	14.20

Use of Resources: Version 1 Results

Replicas	Fanout	Interval	Throughput	Mean	Leader CPU	Follower CPU
5	2.00	200.00	6128.64	1.63	75.62	65.45
11	3.00	300.00	4024.28	2.48	76.82	61.33
19	3.00	400.00	3452.85	2.89	68.72	53.80
31	3.00	500.00	2813.36	3.55	63.28	49.58
53	3.00	600.00	2390.66	4.18	58.84	45.10
77	3.00	700.00	2167.41	4.61	54.55	41.63
101	3.00	700.00	1965.50	5.08	55.45	43.00
125	3.00	800.00	1776.58	5.62	51.05	40.52

Use of Resources: Version 2 Results

A.6 Performance in Network Partitions

5-Replica Systems

Faults	Throughput	Mean	Min	Max	P95	P99
0	6368.78	1.55	0.24	99.92	2.15	15.37
1	6518.44	1.52	0.23	99.95	2.04	14.88
2	6465.93	1.53	0.23	99.47	2.08	15.24
3	2768.77	4.73	0.21	96.84	16.07	29.63
4	966.89	0.48	0.22	64.74	0.78	2.40
5	864.75	0.50	0.22	57.35	0.90	2.66
6	915.09	0.47	0.21	88.58	0.74	2.01

Performance in Network Partitions with 5 Nodes: Raft Results

Faults	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
0	2.00	300.00	9588.51	1.04	0.26	335.01	1.59	8.77
1	2.00	300.00	9632.12	1.03	0.25	342.78	1.56	8.81
2	2.00	300.00	9962.55	1.00	0.25	409.54	1.52	6.95
3	2.00	300.00	8858.21	1.13	0.25	306.74	2.00	5.71
4	2.00	300.00	9367.47	1.06	0.25	317.25	1.87	4.22
5	2.00	300.00	8663.78	1.15	0.24	311.24	1.97	4.17
6	2.00	300.00	7113.38	1.41	0.24	215.85	2.04	3.88

Performance in Network Partitions with 5 Nodes: Version 1 Results

Faults	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
0	2.00	200.00	6347.66	1.57	0.27	207.33	2.93	13.14
1	2.00	200.00	6371.96	1.56	0.26	271.93	2.91	12.78
2	2.00	200.00	6423.74	1.55	0.27	240.24	2.86	11.87
3	2.00	200.00	5390.53	1.85	0.28	197.23	3.73	8.85
4	2.00	200.00	4768.93	2.11	0.28	143.03	4.38	8.39
5	2.00	200.00	4738.35	2.11	0.28	150.38	4.57	8.14
6	2.00	200.00	12.41	3.19	0.40	10.29	7.03	9.55

Performance in Network Partitions with 5 Nodes: Version 2 Results

53-Replica Systems

Faults	Throughput	Mean	Min	Max	P95	P99
0	703.16	14.22	1.57	50.19	26.36	34.57
53	526.21	13.37	1.26	96.96	25.55	34.44
106	525.15	12.75	1.14	94.04	25.26	35.88
159	454.76	11.56	1.15	98.87	24.61	37.34
212	476.62	10.55	1.02	93.67	23.84	34.80
256	437.32	10.03	1.10	98.76	22.80	36.68
318	364.59	8.00	1.05	99.88	20.19	31.72

Performance in Network Partitions with 53 Nodes: Raft Results

Faults	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
0	5.00	1250.00	3014.07	3.31	0.81	133.80	6.61	15.65
53	5.00	1250.00	3056.68	3.27	0.82	149.17	6.27	15.18
106	5.00	1250.00	3109.17	3.21	0.82	149.45	6.00	14.46
159	5.00	1250.00	3157.98	3.16	0.81	118.04	5.70	13.78
212	5.00	1250.00	3228.23	3.09	0.82	117.93	5.25	13.12
256	5.00	1250.00	3221.88	3.10	0.82	111.13	5.20	12.90
318	5.00	1250.00	3240.30	3.08	0.83	134.08	4.98	12.48

Performance in Network Partitions with 53 Nodes: Version 1 Results

Faults	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
0	3.00	600.00	2386.95	4.18	1.34	68.15	7.87	13.90
53	3.00	600.00	2343.01	4.26	1.31	102.83	7.51	13.23
106	3.00	600.00	2293.79	4.35	1.38	89.09	7.35	13.01
159	3.00	600.00	2229.95	4.48	1.46	106.54	7.23	12.53
212	3.00	600.00	2142.24	4.66	1.39	93.61	7.40	12.40
256	3.00	600.00	2112.62	4.73	1.49	92.94	7.40	12.12
318	3.00	600.00	1990.39	5.02	1.45	107.08	7.77	12.03

Performance in Network Partitions with 53 Nodes: Version 2 Results

125-Replica Systems

Faults	Throughput	Mean	Min	Max	P95	P99
0	245.74	40.70	4.16	84.09	61.90	70.45
125	214.04	40.53	4.60	97.80	60.38	69.56
250	190.93	40.39	2.93	99.11	61.10	71.78
375	162.71	40.08	2.40	98.89	59.56	73.32
500	165.91	40.17	4.24	98.71	61.57	74.36
625	163.88	39.31	3.05	98.79	60.07	73.99
750	149.58	38.78	3.33	99.61	59.75	78.48

Performance in Network Partitions with 125 Nodes: Raft Results

Faults	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
0	5.00	3500.00	1239.07	8.07	1.93	68.10	19.54	27.89
125	5.00	3500.00	1240.72	8.06	1.94	70.73	19.47	27.33
250	5.00	3500.00	1254.67	7.97	1.94	77.04	18.94	27.48
375	5.00	3500.00	1255.17	7.96	1.94	84.17	18.72	27.49
500	5.00	3500.00	1257.53	7.95	1.90	84.78	18.88	26.77
625	5.00	3500.00	1270.61	7.87	1.93	75.65	18.66	26.59
750	5.00	3500.00	1289.47	7.75	1.92	73.54	18.01	26.02

Performance in Network Partitions with 125 Nodes: Version 1 Results

Faults	Fanout	Interval	Throughput	Mean	Min	Max	P95	P99
0	3.00	800.00	1713.99	5.83	1.70	98.10	10.77	17.80
125	3.00	800.00	1724.43	5.79	1.97	89.87	10.41	17.91
250	3.00	800.00	1736.05	5.75	1.96	97.15	9.92	16.80
375	3.00	800.00	1755.03	5.69	1.85	99.16	9.61	15.17
500	3.00	800.00	1749.01	5.71	1.95	85.35	9.48	15.01
625	3.00	800.00	1723.53	5.79	2.13	98.91	9.44	14.92
750	3.00	800.00	1696.36	5.89	2.03	95.85	9.47	14.71

Performance in Network Partitions with 125 Nodes: Version 2 Results

Este trabalho é cofinanciado pela Componente 5 - Capitalização e Inovação Empresarial, integrada na Dimensão Resiliência do Plano de Recuperação e Resiliência no âmbito do Mecanismo de Recuperação e Resiliência (MRR) da União Europeia (EU), enquadrado no Next Generation UE, para o período de 2021 - 2026, no âmbito do projeto ATE, com a referência 56.