PRIME : PRobabilistIc MEmbership – Large Scale Membership and Consistency

Francisco Miguel Polaco Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisors: Prof. Miguel Ângelo Marques de Matos
Prof. Rodrigo Seromenho Miragaia Rodrigues

Examination Committee

Chairperson: Prof. João Emílio Segurado Pavão Martins
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. João Carlos Antunes Leitão

November 2018

This thesis is dedicated to everyone who has
guided me to become what I am today.

Acknowledgments

Finishing a five-year study cycle, culminating in the development of PRIME, was one of the most exceptional experiences of my life. However, success does not arise because one wants. Instead, it requires hours of hard work and failure, which must be overcome with persistence.

While chasing success, persistence is something put to the test. Fortunately, I have plenty of caring and supportive people around me.

I thank my father António for transmitting ethical and professional values that were essential to my success. My mother Paula and grandmother Lurdes for the emotional support and for being worried every time I hit a roadblock. I am grateful to my brother for always having something fun to say or show, like *memes*, helping me unwind a bit. I have really enjoyed our journey in life, since the beginning.

A big thanks to Inês for being a great guide in life and a fantastic woman, always motivating me to be better in every aspect and understanding the need for constant evolution. You are the best!

Many thanks to all my close friends, either back home or in Lisbon, for every moment we had, making it possible for me to be at my best. A big thanks to my older friends and other relatives as well, for always providing excellent advice. I am sorry for not naming all of you, but I could not do it in such a small space!

My sincere thanks to Professor Miguel Matos and Professor Rodrigo Rodrigues, my official advisors, and to Nuno Machado and Francisco Maia, my unofficial advisors, for contributing to PRIME's development, my own improvement and for keeping motivated with their words.

I also thank Professor João Leitão for the critical feedback during the thesis proposal presentation and INESC-ID for providing funds allowing the usage of cloud computing.

Finally, studying at Instituto Superior Técnico provided me some restless nights, but also gave me some of the best moments in life one could want. I have met great people with a positive attitude and brilliant minds. It is, without a doubt, an essential variable in my development and in the critical yet positive way I face challenges.

Life has been a great adventure, I am hoping it remains that way!

Resumo

Atualmente, com as oportunidades que a computação na nuvem nos trouxe, as organizações trabalham para ter o máximo de disponibilidade dos serviços. No entanto, para conseguir atingir tal meta, a coerência foi sacrificada, o que tornou o desenvolvimento de aplicações distribuídas muito mais difícil.

Semânticas fortes tornam um sistema lento e limitam a sua escalabilidade. Assim, estes serviços oferecem apenas garantias fracas, o que pode causar o funcionamento errado do sistema, por exemplo, balanceamento de dados errado devido a nós terem vistas diferentes no sistema.

Muitas aplicações distribuídas são construídas sobre uma abstração que oferece uma lista atualizada dos nós corretos – um serviço de filiação. Relacionando estes serviços com as duas semânticas, a semântica forte impede o desenvolvimento de uma solução para escalas grandes. Por outro lado, a semântica fraca, em situações de instabilidade, não permite a coerência da lista de nós corretos.

Neste trabalho, propomos o PRIME, um serviço de filiação total alternativo que usa como meio de disseminação principal um algoritmo probabilístico configurável, que oferece uma abstração de ordem total. Tirando partido dele, conseguimos construir um serviço escalável e probabilisticamente consistente com alto desempenho.

O PRIME é modular e tem uma interface simples, sendo fácil tanto para integrar num sistema já existente ou para ser usado num sistema novo.

Comparámos o PRIME contra outros sistemas reais. Os resultados mostram que o PRIME tem uma performance similar a soluções que oferecem garantias fracas, ao mesmo tempo que oferece coerência nas vistas, que progridem na mesma ordem em todos os nós corretos.

Palavras-chave: sistemas distribuídos, serviço de filiação, probabilístico, coerente, grande escala

Abstract

Nowadays, with the opportunities cloud computing brought us, companies aim to provide the maximum service availability they can. However, this came at the expense of consistency, which made it much more challenging to develop a distributed application.

Stronger semantics will slow down systems and limit scalability. Therefore, these services provide weak guarantees which may lead to incorrect system operation, for instance, wrong data balancing due to nodes having different views of the system.

Many distributed applications are built on top of an abstraction able to provide an updated list of correct nodes – the membership service. Considering the two semantic types, stronger semantics do not allow for solutions to perform well in large scale. In contrast, weaker semantics are unable to keep the lists consistent in a unstable environment.

In this thesis we propose PRIME, an alternative total membership service using as its main dissemination method a tunable probabilistic algorithm, which provides a scalable total order abstraction. Taking advantage of it, we can build a scalable and consistent service with high performance.

PRIME is modular, and it has a simple Application Programming Interface (API), meaning it is easy to either integrate into already built applications or to build a new application using it.

We compared PRIME with other state of the art systems. The results show that PRIME has similar performance to solutions with weak guarantees and it preserves view consistency allowing the view of all correct node to progress in the same order.

Keywords: distributed systems, membership service, probabilistic, consistency, large scale

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Tables xiii
List of Figures xv
Nomenclature 1
Glossary 1

1 Introduction **1**
1.1 Motivation 2
1.2 Goals 2
1.3 Thesis Outline 3

2 Background and Related Work **5**
2.1 Membership Protocols Overview 5
2.2 Membership Protocols with Strong Guarantees 6
2.2.1 ISIS & Horus 6
2.2.2 Rapid 7
2.2.3 JGroups 7
2.2.4 Norbert 7
2.3 Membership Protocols with Weak Guarantees 8
2.3.1 SWIM 8
2.3.2 Gossip-style Failure Detection Service 9
2.3.3 CYCLON 10
2.4 Background 10
2.4.1 EpTO – PRIME's Dissemination Mechanism 10
2.4.2 ϕ Accrual Failure Detector 11
2.5 PRIME's Target Applications 12
2.5.1 Scatter 12
2.5.2 Etna 13
2.5.3 Industry Applications 13

2.6 Summary . . . 14

3 Architecture **15**

3.1 Architectural Overview . . . 15

3.2 Membership Manager . . . 16

3.2.1 Maintain Membership . . . 17

3.3 Hole Manager . . . 19

3.4 Failure Manager . . . 20

3.4.1 Logical Node Placement . . . 21

3.4.2 Suspicion Mechanism . . . 21

3.5 Summary . . . 23

4 Implementation **25**

4.1 Implementation Overview . . . 25

4.2 Application Programming Interface . . . 26

4.3 Optimizations . . . 26

4.3.1 Failure Detector State Transfer . . . 26

4.3.2 Ring Internal Representation . . . 27

4.3.3 View's Hash Caching . . . 27

4.3.4 Monitor . . . 27

4.4 Summary . . . 27

5 Evaluation **29**

5.1 Metrics . . . 29

5.2 Evaluation Scenarios . . . 30

5.3 Results . . . 30

5.3.1 Update Installation During Bootstrap . . . 30

5.3.2 Reaction Results . . . 32

5.3.3 Churn Results . . . 32

5.3.4 Catastrophic Results . . . 32

5.3.5 PRIME's Node Removal Time Decomposition . . . 34

5.3.6 Views' Total Order Progression . . . 36

5.3.7 Resource Usage . . . 37

5.3.8 PRIME's Bootstrap Time Without CPU Cap . . . 39

5.4 Summary . . . 39

6 Conclusions **41**

6.1 Achievements . . . 41

6.2 Future Work . . . 42

Bibliography **43**

List of Tables

2.1 Characterization of the different works 14

3.1 Out of order deliver cases 19
3.2 How are goals achieve with PRIME 23

5.1 Reaction times in seconds 32
5.2 Memory Usage of Tested Systems (Min/Max) 37
5.3 Resources difference between PRIME and Rapid 39
5.4 Summary of Results 40

List of Figures

2.1 Dissemination Mechanism Stack 12

3.1 Architecture Overview 16
3.2 Finding the missing timestamp 19
3.3 Logical Ring 21

5.1 Bootstrap Time Comparison 31
5.2 Bootstrap CDF 31
5.3 Churn 1% Removal CDF 33
5.4 Churn 2% Removal CDF 33
5.5 Churn 5% Removal CDF 33
5.6 Catastrophic 10% Removal CDF 34
5.7 Catastrophic 20% Removal CDF 34
5.8 Catastrophic 30% Removal CDF 34
5.9 Failure Detection Decomposition 35
5.10 Distance Between Views 36
5.11 Network Usage Comparison 38
5.12 Network Usage CDF 38
5.13 Bootstrap Time Comparison Without PRIME's CPU cap 39

Chapter 1

Introduction

Distributed systems used to be built at a smaller scale than today, generally with a few dozens of nodes operating over a relational database, to ensure strong consistency, with a monolithic architecture. Back then, for companies to scale, they had to buy more hardware which reflected in high maintenance costs, causing a decrease in their profit margin.

Moreover, systems may have multiple database replicas to improve availability and may use Group Communication with reliability and ordering properties to synchronize them. This approach requires a total and consistent membership. This is expensive and requires a significant amount of coordination among the nodes making systems impossible to scale to thousands of replicas.

Group Communication was applied in a Local Area Network (LAN) because links are less prone to failures. Outside these controlled static environments, mainly if latency cannot be ignored and failures may easily happen, this technique is not so trivially implemented, and performance can suffer.

Today, in order to meet users expectations companies adopted a business model, made possible by cloud computing, where companies pay only for the resources used and aim to provide the maximum of availability they can.

Furthermore, the flexibility of cloud computing allows to have larger system deployments to serve more users and continued software updates to introduce features or bug fixes, resulting in larger profit margins by reducing unnecessary tasks, like system maintenance[1][2].

This shift in business model took less than a decade and drove small clusters using view synchrony[3] to become large ones using weaker models, to achieve the desired performance. New architectures like microservices[4] also aim to scale more. The popularity of these models led to the creation of technologies like Amazon DynamoDB[5] and Akka[1].

For instance, Twitter[6] prefers that some people read the newest post of Katy Perry, the user with most followers, in the first seconds of posting, while others only see it after ten minutes, instead of having their system unusable by leveraging a strong consistency approach.

Nevertheless, weak consistency makes developers' work much harder than before. Applications have to be aware that after issuing an operation, like writing to the storage, if it issues another one, there

[1] https://akka.io/

is nothing that ensures it will read what it just wrote.

This example shows how frustrating is to work with systems that guarantee almost nothing, where additional training and effort is required, and how it is entirely different from developing to an environment without distribution or with a single centralized server.

1.1 Motivation

Storage systems that offer weaker consistency models need to use techniques like gossip[7] to disseminate data and partial views of the system.

Apache Cassandra[2][8] is an example of a storage system, used in real-world companies, like Apple[9] or Netflix[10]. Cassandra is a decentralized fault-tolerant NoSQL database built on top of an eventual consistent Distributed Hash Table (DHT), where data is replicated among nodes according to a specified replication factor.

Cassandra has open tickets, in particular, one[11] since early 2015, regarding membership problems, in production environments, showing that it is difficult to get it right, despite being a critical component of the system.

Furthermore, in 2010, Facebook decided to migrate its messaging service from Apache Cassandra to Apache HBase[3], due to the latter offering a stronger consistency model which made the developers' work far easier[12].

These examples highlight the importance of having a membership system that is scalable yet can offers good consistency guarantees.

Ensuring a stable and consistent membership in systems like these is fundamental because there is a direct impact of data balancing and distribution among the nodes, when nodes leave or join, which will impact performance. Additionally, they might trigger failure recovery mechanisms sometimes causing catastrophic outages[13], due to bugs caused by their difficult development.

As an example of a failure recover mechanism, a system may continue to progress in degraded mode, disabling features or reducing performance, which can cause service unavailability.

In short, PRIME is motivated by the need of a stronger abstraction, to ease applications' development, and by the importance of decreased system instability, due to re-balancing and failure recovery mechanisms that affect system performance.

1.2 Goals

We want to offer application developers a better membership service that is simultaneously able to a scale of thousands of nodes, offers probabilistic strong consistency, and has good performance.

Membership is a fundamental building block of some distributed systems. I this thesis, we propose **PRIME**. A membership service with the following features:

[2]http://cassandra.apache.org/
[3]https://hbase.apache.org/

- **Highly scalable.** It is able to effectively support systems composed by thousands of nodes;
- **Decentralized.** There is not a set of master nodes. All nodes may have the same behavior;
- **Consistent.** It preserves view consistency allowing the view of all correct node to progress in the same order;
- **Maintains a total view of the system.** Every node knows of the other nodes in the system;
- **Churn[14] tolerance.** It resists to scenarios where a small percentage of nodes join and fail concurrently, during a period of time;
- **Catastrophic failures resilience.** It tolerates a large percentage of concurrent node failures.

In order to achieve consistency and scalability, we follow a decentralized approach and use a dissemination total order probabilistic algorithm with a tunable and quantifiable probability of agreement among the nodes. These features make our approach a middle ground between strong and weak consistency.

1.3 Thesis Outline

The remainder of the document is structured as follows. Chapter 2 discusses different state-of-the-art approaches to build membership services, failure detection approaches and dissemination mechanisms and the connections between them. It also presents distributed applications which could benefit from PRIME.

Chapter 3 describes the proposed solution and its high-level architecture. Chapter 4 details the implementation and optimizations. Chapter 5 defines the evaluation parameters, metrics, environments and presents experimental results. Finally, Chapter 6 concludes this document.

Chapter 2

Background and Related Work

To create something like what we propose to, we cannot proceed without considering existing work. In this Chapter, we cover membership protocols relevant to PRIME's development, either because they are a direct competitor or part of PRIME's dissemination mechanism described in Section 2.4.1. A membership protocol or service is an abstraction able to provide an updated list of correct nodes – system view – in a given system.

Additionally, a membership protocol needs to be aware of failed nodes so that it may remove them from the correct nodes' view. So we present failure detection solutions, some already built into membership protocols.

Finally, since PRIME is an auxiliary service to build distributed applications that could leverage a membership protocol, for example, data stores, we detail some of those applications in Section 2.5 and how they could leverage PRIME.

Our system model has the following properties:

- Crash failure model;
- Asynchronous nodes and network;
- Fair-lossy channels.

2.1 Membership Protocols Overview

There are two opposite approaches to keep track of a system's membership. On the one hand, there are membership systems which are consistent but take a significant amount of time to process changes, such as Horus[15]. These systems ensure a correct total view of the system at the expense of performance. Examples are discussed in Section 2.2.

On the other hand, some membership systems sacrifice consistency to improve performance. This means that each node may have a different view of the same membership, which can be undesirable for some applications. Examples are discussed in Section 2.3.

New alternative approaches try to merge the two philosophies described above, aiming to develop a membership which is consistent and efficient. We fall into that category. However, there are already existent systems that also share the same goal. An example of it is discussed in Section 2.2.2.

Furthermore, there are approaches that address the presence of malicious nodes[16]. In the context of our work, we focus exclusively on crash faults.

2.2 Membership Protocols with Strong Guarantees

In this section, we present a classic membership solution and two more recent ones, in particular, one which is used in nowadays applications. As stated in the previous overview, these solutions suffer from scalability issues, due to the strong guarantees offered by them.

The approach usually followed, to ensure strong guarantees, is using a service like Apache ZooKeeper[1] or Chubby[17], where a set of replicated nodes store and maintain the truth, in this case of the system's membership.

This solution is quite popular due to being simple and straightforward.

2.2.1 ISIS & Horus

ISIS[18] is a toolkit to develop fault-tolerant distributed applications. To develop membership protocol, developers had to use process groups, an ISIS' communication mechanism, which were inefficient. Some of the reasons for this were:

- A detected failed node would cause each node to broadcast its failure. This would cause the same view change times the number of nodes destabilizing the system;
- A conservative protocol to ensure strong consistency requiring more acknowledgments than needed, as a result, applications would not have any benefit of using asynchronous communication.

Horus[15] appeared as a solution to these problems, allowing for the use of a membership protocol abstraction.

At this point in time, lighter approaches to existing mechanisms and better software engineering practices were emerging shaping Horus to develop lighter groups with a well-defined interface and to be used as a membership mechanism.

Horus was revolutionary, because it used these lighter mechanisms, like threads instead of multiple processes, making it much faster, than other alternatives commonly used at the time.

Nonetheless, Horus was developed taking into account small and mostly stable environments, while we target large scale dynamic settings. It is also curious to verify that nowadays common practices, like the use of threads and standard libraries for memory management, were not common when ISIS was originally proposed.

[1] https://zookeeper.apache.org/

2.2.2 Rapid

Rapid[19] is a new membership management system that tries to operate at a large scale while providing a consistent membership. Rapid aims to solve the same problem we do, but it follows a different approach, based on fast consensus.

Every node in Rapid monitors N *subjects* and is monitored by N *observers*. These roles are calculated using a deterministic topology that each node can compute locally. The monitoring can generate **JOIN** and **REMOVE** irrevocable messages both disseminated by gossip. The former is generated when a new node wants to join the system, while the latter is broadcast when there is a reachability problem with a subject.

Since multiple nodes will broadcast these messages, the remaining nodes can collect all of them creating the same view change proposal in almost every node. Finally, to commit this proposal, a Fast Paxos[20] variation is executed if there is a quorum larger than three-quarters of the nodes.

Usually, Rapid can achieve a view change with Fast Paxos[20]. However, if there is not a large enough quorum, a classical Paxos[21] is executed.

In Chapter 5, Rapid is compared against PRIME, where we have identified that its strong guarantees sometimes are violated when the system suffers from failures.

2.2.3 JGroups

JGroups[22] is a clustering library written in Java offering reliable one-to-one or one-to-many communication primitives. It enables applications to join or leave clusters, send information to nodes within a cluster and it keeps the system's view, tracking joined, departed, and failed nodes.

JGroups has the ability to adapt to the developer needs, meaning he can choose what guarantees will be ensured, to avoid paying the price of unnecessary features, for instance, the ordering semantics, reliable message transmission, and encryption.

JGroups is used in nowadays applications like JBoss[23]. However, JGroups' performance rapidly degrades with the increase of cluster size[24], making it impossible to consider for large-scale applications.

2.2.4 Norbert

Norbert[25] is a membership protocol, made by LinkedIn, that relies on Apache ZooKeeper.

Apache ZooKeeper is a clustering software, which ensures strong consistency by electing a leader, using a consensus algorithm. Write operations are limited in scalability since every write must go through the leader, which will atomically broadcast the operation. As a result, consistency is ensured sacrificing some performance.

In theory, Norbert provides strongly consistent membership awareness, but in practice it cannot keep the views consistent, due to the way watchers work in ZooKeeper. This behavior is discussed in Section 5.3.6. Additionally, its scalability is compromised due to ZooKeeper's design.

In Chapter 5, this system will be compared with PRIME, as a representative of strong consistency membership protocols.

2.3 Membership Protocols with Weak Guarantees

In this section, we focus our discussion on gossip-based membership protocols. Due to their high scalability and resilience, these protocols are widely used nowadays in systems like Cassandra[8], Serf (presented in Section 2.3.1), and Ringpop[26].

However, since they only guarantee weak semantics, keeping the view correct, i.e. the same at all nodes, can be an issue.

2.3.1 SWIM

SWIM[27] is a protocol that allows nodes to have weak guarantees of the membership by disseminating updates to the view using a gossip approach, making it more scalable.

Typically, the classic approaches to membership services do not scale beyond a few nodes, due to the overhead caused by the number of messages exchanged to maintain a total and consistent membership.

The failure detection mechanism in SWIM tries to be aware of problems between links, like traffic congestion, to not falsely declare nodes as failed. This avoids view incoherency since nodes are only removed if they indeed failed. The protocol is succinctly explained next.

Consider two nodes N_1 and N_2. When a period of failure detection starts in N_1, it chooses N_2 randomly from its view and sends it a ping. If N_2 does not reply after a given timeout, N_1 will ask the other nodes in its view to ping N_2. In short, N_1 ask another node, which probably is going to use another route, to ping N_2 and check if the node has failed.

To further avoid view incoherency, SWIM tackles the false positives rate of the Failure Detector (FD) by introducing an intermediary state between Alive and Failed.

Consequently, the states that characterize nodes in the membership are Alive, Suspected and Confirmed – the idea of this keyword is that the node is confirmed to be failed.

The Alive state represents when a node is working correctly and replying to the FD. The Suspected state means that the node did not respond to the failure detector and it is under a time window to reply to any ping. Should it stay in the same state until the end of the time window, it will be considered faulty. However, if the node reply to any ping message, it will transit to the Alive state.

Finally, the Confirmed state means that the node is faulty and will be eventually discarded from the membership of every node. As a result, nodes are not immediately rejected, allowing them to overcome short periods of high load or network congestion.

Whenever there is a change in the state of the membership, for instance, a node in the view changes to a Suspected state, there is the need to disseminate an update. To do so, when a node replies to the messages of other nodes' FD, it piggybacks membership updates.

This technique allows an efficient propagation of information, one of the key aspects of SWIM, because it takes advantage of the packets that are sent as part of the normal operation of the system. Therefore, each node's view will eventually become a total view.

SWIM is very scalable. Its techniques to be aware of link failures, of decreasing the false positives and of piggybacking membership updates makes it more resilient to link failures and more efficient. The latter is the result of less instability in the system and because nodes need to send fewer messages to disseminate membership updates.

Nevertheless, it only provides a weak consistent view of the membership and we aim to ensure stronger guarantees.

Serf

Serf[28] is a clustering service built on top of Memberlist[2], which is an optimized implementation of SWIM, made by HashiCorp. These optimizations are described in Lifeguard[29][3] and in their website[30] as well.

In Chapter 5, this implementation will be measured against PRIME, due to its extreme scalability, as a representative of protocols with weak consistency.

2.3.2 Gossip-style Failure Detection Service

The work in [31] provides failure detection based on random gossiping that takes into consideration network topology and is able to scale both in detection time and network load.

Every node has a list, which can be seen as a local view, where for each member it has its address and a counter — that will be used to detect failures. Besides, it is kept the last time each counter associated with the nodes in the list was updated. This protocol can be seen as a partial view membership service, where each node only knows some nodes.

Consider two nodes N_1 and N_2. Periodically, the counter corresponding to N_1 is incremented. Then, N_1 selects a random node N_2 from its list and sends its list to N_2. Upon receiving a message containing a list, N_2 merges its list with the one from the message.

Additionally, N_2 sets its counter to the maximum received from the different gossip messages. Furthermore, using the last time a node's counter has been incremented, one can check if enough time went by to consider it failed (T_{fail}).

Since this system has a weak consistency model, it is possible for a node to consider other faulty and still receive membership updates including the latter. To avoid this, faulty nodes are only forgotten after a period of time (T_{cleanup}) larger or equal than the one needed to consider a node faulty — $T_{\text{cleanup}} \geq T_{\text{fail}}$.

The most significant contribution of this work is taking into account the network topology when nodes are gossiping to others. The core idea is to gossip freely in a subnet, but be more conservative when gossiping between subnets and even more when doing it across Internet domains. Connections between subnets and between domains can be seen as bridges, whose failure will cause partitions.

[2]https://github.com/hashicorp/memberlist
[3]https://www.hashicorp.com/blog/making-gossip-more-robust-with-lifeguard

To do so, the system uses a probability such that, periodically, on average one node that belongs to a subnet will send a message to another node in another subnet. The same happens for domains. This technique improves the resilience to partition since there is lower traffic going through the bridges.

Again, this approach suffers from only providing a weak consistent view of the membership, which can cause problems like the ones already discussed.

2.3.3 CYCLON

CYCLON[32] is an inexpensive membership protocol with weak guarantees based on gossip, with a rather simple design but amazing capabilities, focused on unstructured Peer-to-Peer (P2P) networks.

These networks have a random nature regarding links between nodes, high dynamism and self-healing properties when catastrophic failures happen.

CYCLON exploits the randomness, to maintain a Directed Acyclic Graph (DAG) containing all nodes, in a way that a node knows of the existence of some other random nodes, that is a system's partial view. The partial views do not grow linearly with the system's size making it a scalable mechanism.

Note that a partial view differs from a total view because the former has only part of all nodes in a given system, whereas the latter has every node.

Periodically, a node contacts others from its partial view and exchange information, allowing its view to be refreshed. This is particularly interesting since CYCLON does not have a typical failure detector to remove failed nodes.

Instead, if a node does not exchange information, it will be eventually forgotten and removed from the view of every node, effectively removing it from the network. Using this mechanism, a node easily can introduce a new node N by having another node replace an element of its view by N.

CYCLON belongs to the Peer Sampling Service (PSS) protocol family, which is an abstraction that provides a uniform random sample of all nodes in the system. Using this, nodes can exchange information with each others. In practice, they can be seen as partial view membership protocols with weak guarantees.

2.4 Background

In this section, we present two works used by PRIME's design and implementation. The first is a dissemination mechanism used to achieve the guarantees we aim, while the second is used to monitor the nodes' health.

2.4.1 EpTO – PRIME's Dissemination Mechanism

EpTO[33] is a dissemination mechanism that ensures that nodes eventually agree with each other in a set of received events, with high probability, and that this set is delivered to the application.

This probability can be tuned to the point that it is more likely for a hardware failure to happen than a total order violation.

Total order provides a powerful abstraction making it of particular interest to design Distributed Systems. In fact, for our work having total order would mean that we could easily discard old membership updates that contain nodes already failed.

Furthermore, we can push updates such that every node progresses by the same order. However, deterministic total order algorithms cannot scale to the magnitude that we target. Therefore, a probabilistic algorithm can help to achieve that goal.

EpTO has key features such as the use of logical clocks and it being decentralized since it does not require coordination among nodes. EpTO does not assume anything about network and node synchrony.

EpTO has two components. The first is responsible for dissemination of events and the other for ordering. When they arrive from the network, they will go through the dissemination component, which achieves probabilistic agreement. Then, events will be re-transmitted to other nodes and will be sent to the ordering component, which achieves total order, and will deliver the ordered events to the application.

The dissemination component has a PSS protocol, based on CYCLON described at Section 2.3.3, and a maximum constant Time-to-live (TTL), which represents the number of rounds that each event needs to be relayed during its dissemination. In addition, this component has a buffer of next events used to store and re-transmit them.

The ordering component can use either global or logical clocks, but using the second will double the global TTL. Nevertheless, in practice, this only happens if the network has low activity since the clock will be updated every time it receives an event.

As always, there is a trade-off in the rounds duration. Having small rounds is going to make some of them useless because some events have yet to arrive. Having big rounds will increase the delivery delay. Typically, classic approaches use a failure detector to check if replicas have significant latency, discarding them if that is the case.

Our solution is built upon EpTO since it offers a mechanism that delivers updates in a total order fashion with a given adjustable probability while keeping the system scalable. At the same time, since the algorithm is probabilistic, there is always a chance that a node fails to deliver the events in a total order. Therefore, some tolerance mechanism has to be built to deal with those cases.

In short, this means that PRIME is built on top of EpTO, whose uses CYCLON, as illustrated in Figure 2.1. Thus, PRIME maintains a system's total view using a partial view.

2.4.2 ϕ Accrual Failure Detector

ϕ Accrual[34] is a failure detector that collects heartbeats to decide the state of the node. However, this state is not Boolean as others failure detectors. This work introduces a continuous value – ϕ – representing the node' state. Nonetheless, despite the ϕ value, applications may decide over the two classic states only – Alive and Failed.

ϕ tends to infinity as time goes by, but a heartbeat avoids this tendency. A perfect working node will have its ϕ value equal to zero and the failed node will have its value above a certain threshold.

Therefore, applications using this abstraction have a stronger expressiveness than traditional ones,

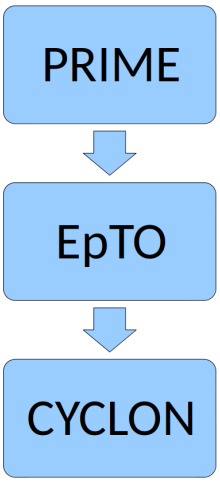

Figure 2.1: Dissemination Mechanism Stack

allowing them to decide better. For instance, an AI component identifies a pattern leading to a problem. This component can, as soon as the pattern appears, execute a protocol to avoid the problem. For example, if nodes are starting to have larger ϕ values, caused by overloaded nodes which will end up with an unresponsive system, it may launch new nodes to load balance it, instead of letting the nodes crash due to being out of memory.

ϕ is influenced by the behavior history of a node, meaning that even if a perfect node fails some heartbeats in a row, its long past will not let the ϕ value increase into values that consider the node failed. This feature makes it a more reliable failure detector.

Implementations of ϕ Accrual are widely used in real-world applications, particularly through the usage of Akka Cluster[35], proving the quality of it. Furthermore, an implementation of this failure detector is used in the current version of PRIME.

2.5 PRIME's Target Applications

Membership is an interesting problem. Still, having the perfect solution for this has no interest, if it cannot be applied. Storage systems, in particular, key-value stores heavily rely on membership services to accomplish their purpose.

Usually, key-value stores are based on structured P2P networks, where it is assumed a somewhat stable environment. However, this does not apply to systems in the scale of thousands of machines, where it is fairly easy for nodes to crash causing the environment to become unstable.

2.5.1 Scatter

Scatter[36] is a distributed storage system, using a peer-to-peer approach, that tries to find a trade-off between strong consistency and scalability, resulting in linearizable guarantees.

Classic DHTs suffer from inconsistencies when failures occur. Namely, assignment violations that cause multiple nodes to own a particular key-value pair, which causes performance problems, and rout-

ing violations that create network partitions making requests impossible to complete.

Scatter is based on a DHT, where the key-space is split among groups. Each group is composed of a set of nodes that are synchronized among them using Paxos[21] and is responsible for a subset of key-value pairs. In addition, each group knows its current, previous, and following neighbors, which avoids gaps in the ring.

To tackle scalability issues, Scatter supports multi-group operations, making the system dynamic and adaptable. For example, to solve the previous case, Scatter can split the group into two, making consensus run faster in each group, instead of having only one large group.

Since these operations modify several groups, Scatter uses a distributed transaction model by executing a two-phase-commit. In particular, before a group sends any message, it runs a consensus to make sure that every node belonging to it are in the same state.

This allows for another group node to pick up the leader's role if the current leader fails.

We believe that Scatter dynamism is its biggest contribution. By making a system adaptable to different scenarios, the chances of a system staying available with good performance are increased. However, there is a particular invariant stating that adjacent groups can always reach each other, which can be unrealistic in a dynamic environment.

Still, Scatter uses Paxos in a Wide Area Network (WAN), which makes the consensus progress slowly, leading to high latency. Instead of using Paxos, Scatter may use PRIME to avoid the problems discussed, increasing its performance.

2.5.2 Etna

Etna[37] is a system that allows atomic reads and writes on a Chord DHT[38]. Inconsistencies about the location of objects are frequent in this type of DHT because they may be frequently moved when nodes are constantly joining and leaving. So, to identify them, Etna relies on the Paxos protocol to reach a consensus of which set of replicas contains the object.

In short, it tries to create a system with strong guarantees by using a weaker consistent mechanism with a highly scalable structure, at a lower level, and applying a protocol that offers strong consistency, at a higher level.

However, as stated in [36], Etna's approach is subject to object unavailability and performance problems.

Etna can be highlighted by its modularity and work re-usage, that can be seen as an opposite design philosophy to Scatter, described in Section 2.5.1, where its DHT suffers from several modifications.

Furthermore, modularity is a property that is extremely interesting to our work, because, hypothetically, we could swap the Paxos module by PRIME and improve Etna's performance and availability.

2.5.3 Industry Applications

Apache Cassandra[8] is a system widely used in production environments all around the world. However, its weak consistency membership mechanism causes, in the presence of network congestion,

inconsistencies in the ring, leading to service failures.

Additionally, Cassandra forces to wait for two minutes[11] between node deployment or shutdown and when a node fails, it must be decommissioned manually, using *nodetool*[39], which may threat system stability and it is a tedious task for the system administrator. This two-minute wait and other unresolved tickets regarding consistent membership issues[40] could potentially be solved using PRIME.

Service Fabric (SF)[41] is a Microsoft platform for handling microservices deployed in the Azure cloud, mainly to maintain such services. SF has a strong consistency membership protocol built using a ring with special nodes to avoid inconsistencies.

Although Microsoft solution works for their use cases, having special roles complicates the design. SF may benefit from PRIME, since every node has the same roles and it is a more general solution, simplifying the design.

2.6 Summary

From all solutions presented, we summarize in Table 2.1 a comparative analysis of the different systems, taking into account the consistency model and the membership scope. Some of the approaches discussed above are not present since they do not address the membership issue.

For sake of recalling the various membership scopes, we present:

- **Total** – Each node knows all nodes in the system;
- **Partial** – Each node only knows some nodes

System	Consistency	Membership Scope
Horus (2.2.1)	Strong	Total
Rapid[4] (2.2.2)	Strong	Total
JGroups (2.2.3)	Strong	Total
Norbert[4] (2.2.4)	Strong	Total
SWIM (2.3.1)	Weak	Total
Serf[4] (2.3.1)	Weak	Total
Gossip Failure Detector (2.3.2)	Weak	Partial
CYCLON (2.3.3)	Weak	Partial
PRIME	**Consistent with Quantifiable Probability**	**Total**

Table 2.1: Characterization of the different works

[4]Compared with PRIME in Chapter 5

Chapter 3

Architecture

In this Chapter, we present PRIME's architecture. PRIME is a membership service with the following features:

- **Highly scalable.** It is able to reach thousands of nodes;
- **Crash failure model.** Nodes may crash at any time, but they do not trick other nodes;
- **Asynchronous nodes and network.** Nodes do not block each time they send a message and there is no upper bound on propagation delay;
- **Fair-lossy links.** Links may lose messages;
- **Decentralized.** There is not a set of master nodes. All nodes may have the same behavior;
- **Consistent.** It preserves view consistency allowing the view of all correct node to progress in the same order;
- **Efficient.** It has good performance and it uses the minimum resources required to function;
- **Maintains a total view of the system.** Every node knows of all the remaining nodes;
- **Churn[14] tolerance.** It resists through a small percentage of nodes joining and failing, during a period of time;
- **Catastrophic failures resilience.** It tolerates a large percentage of nodes failing.

3.1 Architectural Overview

PRIME has three main components:

- **Membership Manager** – Component responsible for maintaining the system's total view as a ring;
- **Hole Manager** – This component has a role in handling holes caused by EpTO's algorithm;

- **Failure Manager** – This component handles all failure detection and the decision of a dead node's removal of the system.

All components exchange messages through the network among nodes to, for example, detect failures or solve holes.

PRIME's main dissemination algorithm is EpTO, which keeps a partial view of the system to be able to disseminate information. This partial view is maintained by a PSS, which in practice is an implementation of CYCLON.

Despite being different components, the Membership Manager connects them. It exchanges information directly with the Hole Manager to fix possible issues and updates the Failure Manager in the case of a view change.

A simple graphical representation of the interaction between components can be seen in Figure 3.1.

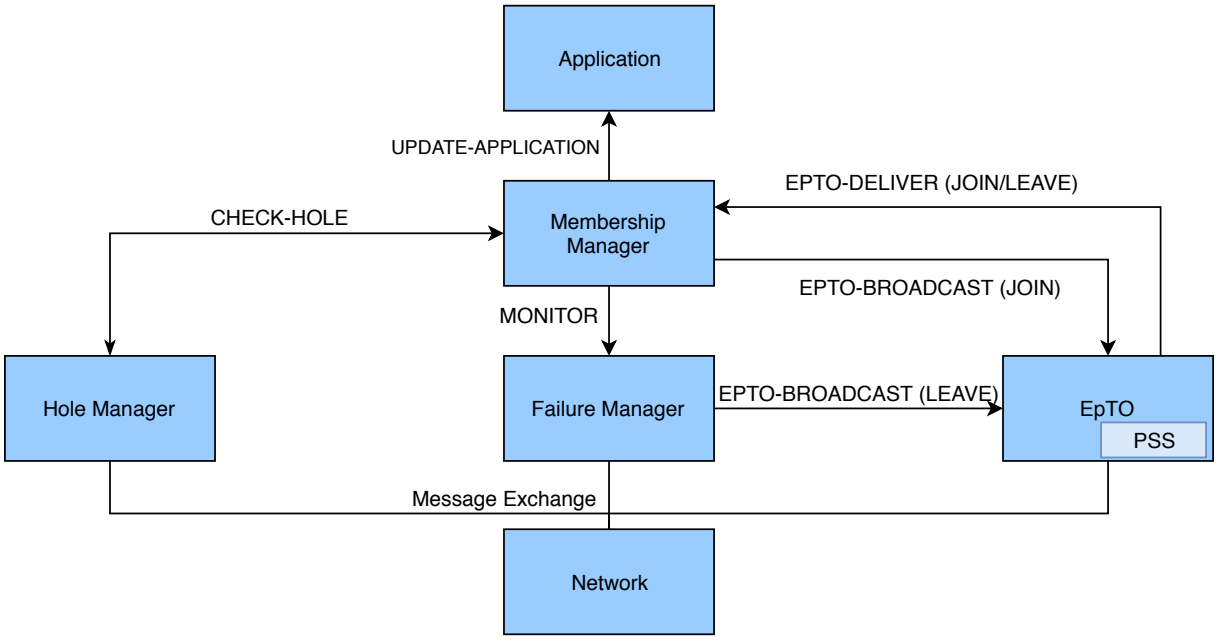

Figure 3.1: Architecture Overview

3.2 Membership Manager

EpTO provides a clear interface for broadcasting messages about nodes leaving or joining the system and it delivers membership updates in total order and with probabilistic agreement.

PRIME also ensures that, since every updates is received through EpTO. This means that membership progression may have holes among the nodes, due to probabilistic agreement, which forces us to handle them with care. We detail this in Section 3.3.

3.2.1 Maintain Membership

While EpTO provides a strong abstraction, we still face several challenges, to maintain the membership in PRIME. For instance, a node needs to bootstrap itself after contacting another node in the system and it needs to tolerate EpTO's holes and out of order deliveries. During this and the following subsection we will cover these challenges.

To add or remove a node from the membership, there are two types of messages: JOIN and LEAVE respectively. Despite EpTO being efficient, PRIME only uses it to disseminate JOIN and LEAVE messages.

For the sake of simplicity, let's assume that the system is stable – every node has the most recent view – and a new node N intends to join the network. Following Algorithm 1, N will contact any node by sending a DISCOVERY message. Note that in the following algorithms, in some lines, there is a notion of Guardians, which will be explained in Section 3.4.1.

Then, the contacted node will EPTO-BROADCAST an update stating that N wants to join. Simultaneously, N starts to receive EPTO-DELIVER events which are stored in a queue, while it waits for a view's copy.

EpTO will disseminate and deliver the update to every node, which will process it. To do so, first, each node needs to check if there is a hole and repair it if needed. Then, the total view structure is updated accordingly.

The application is updated. Since the view has changed, the nodes monitored may change as well, so the Failure Manager also needs to be updated.

Finally, the new node N needs a view's copy so it may also process the updates. Each node will check if it's their role – first Guardian – to send it and fulfill that promise.

After N receives the view's copy, it will discard old updates stored in the queue and apply the remaining ones. In the end, it will initialize its Failure Manager.

Out of order delivers

Since current EpTO design may deliver old events out of order, instead of dropping them as the original design, PRIME needs to take into consideration JOIN events of a node which may have been removed already and discard them.

All possible combinations of out of order events are displayed in Table 3.1 with some considerations.

Since a LEAVE followed by an out of order JOIN is the only problematic case, we can easily solve this by tracking JOIN messages delivered by the out of order callback (Algorithm 1, line 33) and ignore them when they match an already processed LEAVE.

This avoids the use of a CRDT[42], that while it would always solve any inconsistencies without the need of message exchanging, it would also unnecessarily affect performance because the remaining combinations do not represent a problem.

Algorithm 1: Membership Manager (node n)

```
initially  total-view ← {}
initialized ← false
update-queue ← []
ooo-control ← {}
SEND DISCOVERY(n) TO random-node // A random node already in the system


upon receiving DISCOVERY (node)
    if total-view.initialized = true then
        update.command ← JOIN
        update.node ← node
        update.hash-sent ← HASH(total-view)
        EpTO-BROADCAST (update)
procedure PROCESS-UPDATE (update, is-out-of-order)
    CHECK-HOLE (update)
    if update.command = LEAVE then
        total-view ← total-view \ {update}
        ooo-control ← ooo-control ∪ {update}
    else
        if not is-out-of-order ∧ update ∈ ooo-control then
            total-view ← total-view ∪ {update}
    UPDATE-APPLICATION (update)
    MONITOR (total-view)
procedure SEND-VIEW (node)
    if p = FIRST-GUARDIAN (node) then
        SEND VIEW(total-view) TO node
upon receiving VIEW (view-delivered)
    total-view ← view-delivered
    forall update in update-queue do
        // Process all updates that were stored in the queue, while a view's copy was not sent
        PROCESS-UPDATE (update, false)
    MONITOR (view)
    total-view.initialized ← true
upon EpTO-Deliver (update)
    if total-view.initialized = true then
        PROCESS-UPDATE (update, false)
        SEND-VIEW (update.node)
    else
        update-queue ← update
upon EpTO-Deliver-Out-Of-Order (update)
    if total-view.initialized = true then
        PROCESS-UPDATE (update, true)
    else
        update-queue ← update
procedure UPDATE-APPLICATION ()
    // Updates application
procedure FIRST-GUARDIAN (node)
    // Returns first guardian of node
```

First Update	Out of Order Update	Is Problematic?	Considerations
JOIN N	JOIN N	No	Equal updates are only considered once.
JOIN N	LEAVE N	No	It is impossible to have a LEAVE broadcasted before a JOIN since the node needs to exist in the view to be removed.
LEAVE N	JOIN N	Yes	A delayed JOIN may arrive after a LEAVE, for instance, due to network congestion.
LEAVE N	LEAVE N	No	Equal updates are only considered once.

Table 3.1: Out of order deliver cases

3.3 Hole Manager

As stated previously, EpTO may cause holes when delivering a stream of updates. While this can be extremely rare, PRIME needs to tolerate it, so all nodes can converge.

All updates have a hash of the view that was installed when it was broadcasted. Using that remote hash and following Algorithm 2, we can compare if it is the same as either the hash of the locally installed view or an older view hash. If that is not the case, then a hole has occurred and it needs to be fixed.

Nevertheless, it is not trivial to identify where, in the entire stream of updates, the hole happened, because, in moments of high activity, such as bootstrap or churn, the majority of updates will have an old hash.

To identify the missing update, all updates are versioned. EpTO already supports this by tagging events with a pair of timestamp and unique node identifier to avoid giving the same version to concurrent broadcasts. This helps PRIME to fix a hole because a node needs to ask for the help of another one by sending the timestamp to it.

The main idea is to decreasingly go from the update timestamp to zero asking another node – a random Guardian – to verify the updates already received with that timestamp. Once identified the culprit, the missing update is returned and applied for to fix the hole.

In Figure 3.2, the hole fixing process starts in timestamp thirty and goes until twenty two, where it finds the missing one (mark as a red circle).

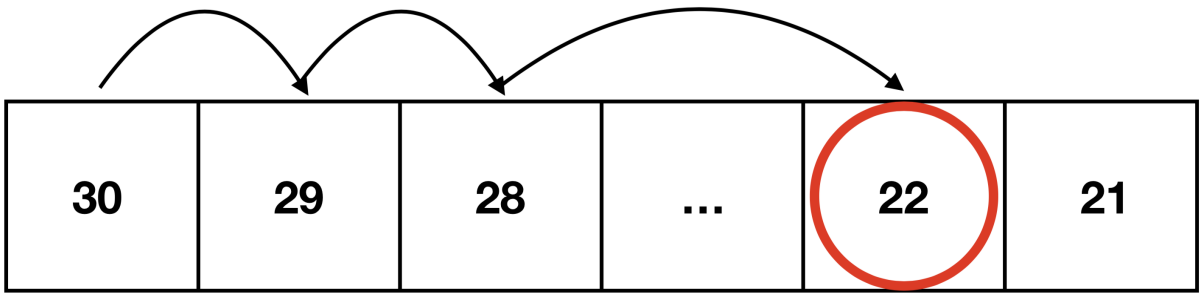

Figure 3.2: Finding the missing timestamp

However, if a hole still exists, which is extremely rare, the CHECK-HOLE procedure can be repeated to solve it. Note that an explicit request is made if a particular timestamp does not have any update

associated. (Algorithm 2, line 10)

To keep the system robust and reliable against an unexpected event, in the worst case scenario, if a hole cannot be fixed, a view's copy is requested and installed, so the node can continue to progress.

Algorithm 2: Hole Manager

```
1  procedure CHECK-HOLE (update)
       // The node needs to fix a hole while our local hash is not the same as the remote one
       // and if the node has never seen such hash since it means it never received at least one
       // update
2      while HASH (total-view) ≠ update.hash-sent ∨ NEVER-SEEN (hash-sent) do
3          ts ← update.timestamp
4          a-guardian-node ← GET-RANDOM-GUARDIAN ()
5          while ts ≥ 0 do
               // Trying to get the missing updates by their timestamp
6              updates ← GET-UPDATES-WITH-TS (ts)
7              if updates ≠ ∅ then
8                  updates-to-fix ← SEND VERIFY(updates, ts) TO a-guardian-node
9              else
10                 updates-to-fix ← SEND REQUEST(ts) TO a-guardian-node
11             if updates-to-fix ≠ ∅ then
12                 forall to-fix in updates-to-fix do
13                     PROCESS-UPDATE (to-fix, true)
14                 break
15             ts ← ts − 1
16 upon receiving VERIFY-UPDATES (updates, timestamp)
17     my-updates ← GET-UPDATES-WITH-TS (timestamp)
18     if my-updates ≠ updates then
19         return my-updates \ updates
20     else
21         return ∅
22 upon receiving REQUEST-UPDATES (timestamp)
23     return GET-UPDATES-WITH-TS (timestamp)
24 procedure GET-RANDOM-GUARDIAN ()
       // Returns a random guardian of p
25 procedure GET-UPDATES-WITH-TS (timestamp)
       // Returns a list of updates applied with the same timestamp
26 procedure NEVER-SEEN (hash-sent)
       // Returns true if hash-sent never existed
```

3.4 Failure Manager

When designing a distributed system, it is mandatory to include failure tolerance techniques, since failures are the norm rather than the exception.

Nodes may fail simultaneously. However, PRIME can handle this situation, since after the view is updated, for instance, by removal of a failed node, the Failure Manager state is reset. While some failure detection state is lost, increasing detection time of a second failed node, it ensures correctness.

The second node will always be detected because the lost state will be generated again since the node has failed.

3.4.1 Logical Node Placement

To tolerate a failure, one needs to detect it first, so it is an important part of the entire failure tolerance mechanism. Therefore, to detect a failure, nodes need to monitor each other, due to our decentralized design. Consequently, each node has a logical ring structure representing the total view.

For each node, there are two classes of neighbors: **Protegees** and **Guardians**. The former are nodes which a node N has to monitor, typically nodes ahead of N. The latter are nodes monitoring N, commonly nodes behind N.

Both classes have a constant number of elements, usually the same. Also, to ensure that each node monitors the right Protegees without exchanging messages to agree on them, the ring is deterministically sorted using each node identifier.

Considering Figure 3.3 and node p as an example, blue nodes are Guardians monitoring p and red nodes are Protegees monitored by p.

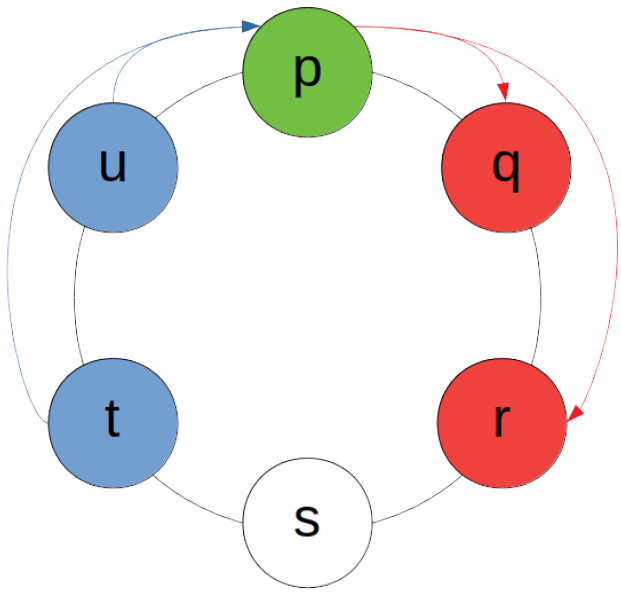

Figure 3.3: Logical ring from the point of view of node p: $\#guardians = \#protegees = 2$

Guardians are also used to recover from holes and are responsible for sending a view's copy when a new node joins.

3.4.2 Suspicion Mechanism

The necessity of multiple Guardians arises from the possibility of mistakes a single node can make while evaluating if another has failed.

Following Algorithm 3, every time there is a change in the membership, the ring may change forcing the Failure Manager to update the Protegees and Guardians, including their state (Algorithm 3, line 6).

Whenever a node is detected as failed by a Guardian's FD, it sends a SUSPICION message to the other guardians of the suspected node (Algorithm 3, line 24).

All Guardians keep collecting SUSPICION messages until either they receive an ALIVE message (Algorithm 4, line 5), meaning a Guardian detected that the node is not failed, or a majority of guardians have sent SUSPICION messages (Algorithm 4, line 3).

The former clears the suspected node state regarding suspicions, while the latter forces the first Guardian to EPTO-BROADCAST a LEAVE message to remove the failed node from the membership.

Algorithm 3: Failure Manager (node n)

```
1  initially current-guardians ← {}
2  current-protegees ← {}
3  suspected-nodes ← {}
4  total-view ← {}
5
6  procedure MONITOR (view)
7      total-view ← view
8      new-guardians ← GET-GUARDIANS (total-view, p)
9      new-protegees ← GET-PROTEGEES (total-view, p)
10     if current-guardians ≠ new-guardians then
11         forall current in current-guardians do
12             DISABLE-HB(disable-heartbeat)(current)
13         forall new in new-guardians do
14             ENABLE-HB (new)
15         current-guardians ← new-guardians
16     if current-protegees ≠ new-protegees then
17         forall current in current-protegees do
18             DISABLE-FD (current)
19         forall new in new-protegees do
20             ENABLE-FD (new)
21         current-protegees ← new-protegees
22         suspected-nodes ← {}
23
24 upon SUSPICION-FROM-FAILURE-DETECTOR (suspected-node)
25     if suspected-node in suspected-nodes then
26         return
27     suspected-node.dead-counter ← suspected-node.dead-counter + 1
28     suspected-nodes ← suspected-nodes ∪ {suspected-node}
29     forall guardian in get-guardians(total-view, suspected-node) do
30         SEND SUSPICION(suspected-node) TO guardian
31 upon ALIVE-FROM-FAILURE-DETECTOR (suspected-node)
32     suspected-node.dead-counter ← 0
33     suspected-nodes ← suspected-nodes \ {suspected-node}
34     forall guardian in get-guardians(total-view, suspected-node) do
35         SEND ALIVE(suspected-node) TO guardian
```

Algorithm 4: Failure Manager (node n) 2

```
1  upon receiving SUSPICION-FROM-OTHER-GUARDIAN (suspected-node)
       // Retrieves the local instance of suspected-node
2      suspected-node ← current-protegees[suspected-node]
3      suspected-node.dead-counter ← suspected-node.dead-counter + 1
4      if suspected-node.dead-counter > (GUARDIANS-SIZE (suspected-node) / 2 ) ∧ n =
         FIRST-GUARDIAN (suspected-node) then
5          EPTO-BROADCAST (<LEAVE, suspected-node, HASH (total-view)>)
6  upon receiving ALIVE-FROM-OTHER-GUARDIAN (suspected-node)
       // Retrieves the local instance of suspected-node
7      suspected-node ← current-protegees[suspected-node]
       // Resets suspected-node state regarding suspicions
8      suspected-node.dead-counter ← 0
9      suspected-nodes ← suspected-nodes \ {suspected-node}
10 procedure GUARDIANS-SIZE (suspected-node)
       // Returns the current number of guardians
11 procedure ENABLE-HB (guardian-node)
       // Enables periodic heartbeats to guardian-node
12 procedure DISABLE-HB (guardian-node)
       // Disables periodic heartbeats to guardian-node
13 procedure ENABLE-FD (protegee-node)
       // Enables failure detection for protegee-node
14 procedure DISABLE-FD (protegee-node)
       // Disables failure detection for protegee-node
```

3.5 Summary

In Table 3.2, we summarize how PRIME achieves the goals presented in Section 1.2, considering the architecture described.

Goal	Technique
Highly scalable	Design with scalability in mind and built on top of scalable dissemination mechanism.
Decentralized	All nodes have the same behavior defined and there is not a central server. All the processing relies on local state only.
Consistent	EpTO ensures total order deliver of the updates.
Efficient	Usage of fast algorithms and data structures.
Maintains a total view	All updates are delivered through EpTO.
Churn and Catastrophic failures tolerance	Failure Manager design.

Table 3.2: How are goals achieve with PRIME

The usage of EpTO has a prime role in achieving scalability, consistency and keeping a total view. Efficiency was achieved using proper data structures and algorithms. A decentralized design is enabled by all the processing relying in local state and failure tolerance capability is due to our Failure Manager design.

Chapter 4

Implementation

We have implemented a prototype, with significant optimizations, that allowed the results presented in Chapter 5. PRIME is implemented as a Java 10 library. It is open-sourced under an Apache 2 license[1]. Our choice of using Java 10 lies on it having native support for Docker[43], which was our main deployment technology, and because EpTO's and CYCLON's implementation was written in Kotlin[2].

4.1 Implementation Overview

Our implementation uses User Datagram Protocol (UDP) for almost all messaging, for example, heartbeats, hole solving or discovery. For sending a view's copy, we decided to use Transmission Control Protocol (TCP) due to its size.

However, to serializing the ring structure, we take advantage of an efficient binary serialization library – FST[3] – to make the process faster. All communication uses JDK Socket API exchanging byte arrays in the case of UDP and the result of FST serialization in the case of TCP.

Every node has a unique identifier so that the ring is ordered as we desire. In the case of our implementation, each node has a random id, for which we used JDK UUID class.

EpTO's current implementation uses a tracker, in practice a REST[44] Python script, where it retrieves its partial view. To avoid having to define a first node, PRIME also uses the tracker to decide whether it should start a membership or join an already existent one.

Additionally, it is used as a way of discovering nodes to further contact and join the membership. To interact with the Tracker, our implementation uses the Apache HTTP library[4] and Google Gson[5]. Note that the Tracker is used only for bootstrap purposes.

Guardian nodes have to monitor their Protegees. As a result, PRIME uses an implementation of ϕ Accrual Failure Detector, described at Section 2.4.2, ported by a developer from Akka to Java[6].

[1] It is private at the moment. https://github.com/francisco-polaco/prime-probabilistic-membership
[2] https://kotlinlang.org/
[3] https://github.com/RuedigerMoeller/fast-serialization
[4] https://hc.apache.org/
[5] https://github.com/google/gson
[6] https://github.com/komamitsu/phi-accural-failure-detector/

Our approach of fixing holes in PRIME was one of the significant challenges we have faced. It suffered several iterations until the final solution. As described in Section 3.3, PRIME uses view's hashes to detect the presence of holes and exchanges updates with another node to fix it.

The hashes are computed using Google Guava[7] 128-bit murmur3 algorithm (x64 variant).

To achieve the update exchanging, every node keeps a history of old updates. However, since it can quickly fill the memory up, we used a LinkedHashMap with its method removeEldestEntry overridden, so it removes the oldest entry when the map is above a configurable size.

4.2 Application Programming Interface

The two main possible ways of an application to interact with any service are synchronous and asynchronous interactions. A synchronous one is usually blocking and will return the result to the calling thread, whereas an asynchronous interaction generally needs a callback registration to be called once the operation is completed and does not block the calling thread.

The former is easier to use but makes the application less responsive to users. The latter is much more difficult to program, since it is concurrent by nature, still makes applications much more responsive.

We took inspiration from C# and .NET Framework and decided to provide both programming paradigms. We present PRIME's API at Listing 4.1.

```
// Using synchronous programming
Ring<Node> getView();
// Using assynchronous programming
// Our service offers a method to register a callback method
void registerOnUpdateViewCallback(OnUpdateViewCallback callback);
// The callback method should have this signature:
void onUpdateViewCallback(Ring<Node> view);
```

Listing 4.1: API

4.3 Optimizations

In this Section, we describe the most relevant optimizations done in PRIME, which help achieve the results shown at Section 5.3.

4.3.1 Failure Detector State Transfer

As discussed in Section 3.4, even if multiple nodes fail simultaneously, PRIME will converge and remove all failed nodes. However, the approach described can be optimized to reduce removal times.

Each Protegee has a corresponding FD keeping state about it. This state is cleared each time the view is changed, but it does need to be that way. It can be kept for Protegees that remain after the new view is installed.

[7] https://github.com/google/guava

Since the state for possible failed Protegees is maintained, the suspicion mechanism is triggered almost immediately making the removal of failed nodes faster.

4.3.2 Ring Internal Representation

Initially, due to the nature of a circular list, we had implemented the Ring as an ArrayList, with a circular iterator, which was sorted every time a node was added and scrolled through to avoid duplicates.

Naturally, there was a significant overhead in adding nodes, so we changed the original implementation into a TreeSet since it orders elements by default among other features like avoiding duplicates.

4.3.3 View's Hash Caching

Hashing the current view is something often calculated, since it is used in the hole detection mechanism, as explained in Section 3.3. So, to improve performance, we store the view's hash after a write operation. Every subsequent request to retrieve the hash is achieved by returning the stored value.

4.3.4 Monitor

The MONITOR method described at Algorithm 3 can be extremely inefficient. Each time the view is changed, it will disable periodic heartbeats for the old Protegees, enable them for the new ones, disable the FD for the old Guardians and enable it for the new ones.

To avoid incurring into costs of FD management and scheduling tasks of nodes that will remain under the Failure Manager control, we calculate the nodes that will be no longer considered and the new ones.

Then, we only apply such costly operations to these particular nodes, significantly improving performance.

4.4 Summary

PRIME was implemented with 4637 lines of code, with a greater effort into maximizing UDP communication for greater performance and scalability.

It has a pretty simple API, so all applications may integrate PRIME easily. All four optimizations contributed successfully for reducing overheads, providing the results shown in Chapter 5.

An instructions file is available at the repository, so everyone can build and use PRIME.

Chapter 5

Evaluation

In this Chapter, we evaluate PRIME by comparing it with other state of the art solutions.

We choose metrics and testing environments that have two goals. On the one hand, they help to show that PRIME approach is correct, that it maintains a consistent total system's view, where its updates are totally ordered. On the other hand, they reveal that our design is efficient and scalable.

PRIME's results are compared with three other membership protocols Serf (2.3.1), Rapid (2.2.2) and Norbert (2.2.4) representing strong and weak consistency protocols.

5.1 Metrics

We consider the following metrics:

- **Views progression.** Number of distinct views delivered to the application, with the same local timestamp, in different nodes. It can be seen as the distance between views and it measures the consistency of a system;
- **Update installation during bootstrap.** The time taken to broadcast, deliver, process and install an update during bootstrap. This shows the latency for each update;
- **Failed nodes removal.** The time taken to remove a failed node from the view of all the nodes since multiple nodes can suspect a specific node at the same time. This checks if explicit dissemination of an event is efficient;
- **Resource usage.** To verify the impact on resources that PRIME has. We measure Memory, Network, and CPU usages.

The first metric dictates if our work is a success or not because it corroborates if the system ensures a progression of all view by the same order. The remaining ones are used as a way of comparing our mechanism with the others.

We used logs collected along the code execution to retrieve all the metrics needed. Our deployment system provides resource metrics logging out-of-the-box, application metrics required additional logging

code written in a lightweight way. All logging was stored in each node persistent storage instead of uploading them to a remote location, avoiding interference with the network and impact the results.

5.2 Evaluation Scenarios

To evaluate the system properly, we have run experiments with a varying number of nodes, churn conditions and catastrophic failures. These allow to show how the system behaves when facing real world situations.

The four systems faced the same tests, where every one of them was run with 100 and 250 nodes. We wanted to evaluate larger system sizes but it was not possible due to budget restrictions.

The deployment of these experiments took place in Google Cloud Platform[1] using n1-standard-1[45] instances (1 vCPU and 3,75 GB of RAM) where each one had Docker containers limited to one-tenth of the CPU, connected using an overlay network.

The number of containers per node was taken into consideration to avoid having a CPU load above 70%. However, note that, due to Rapid's high CPU usage, we had to uncap the CPU per container, to run it. In Section 5.3.7, we make some considerations about resource usage.

5.3 Results

In this section, we present data comparing Serf, Rapid, Norbert, and PRIME regarding view progression, node removal, and bootstrap considering the scenarios, discussed in Section 5.2.

We denote each experiment as System Size, for instance, PRIME 100 is a PRIME experience with 100 nodes.

5.3.1 Update Installation During Bootstrap

In Figure 5.1, we present the bootstrap time for each system and system size. The bars represent the average time and the error bars show the confidence interval with a degree of confidence of 95%[2].

When comparing the systems, Serf is the fastest and Norbert is the slowest. Even so, Serf is way more variable when bootstrapping than the others, represented by the large error bars.

Norbert degrades with the system size abruptly, due to the reasons pointed in Section 5.3.6, namely the time taken to set all the watches grows with the system size, making a cluster with 250 nodes, take 396 seconds (about six and a half minutes) to bootstrap, on average.

Note that even though Rapid is faster than PRIME for 100 nodes, it becomes slower with 250 nodes. We could not validate if this trends continues as the system size grows.

As shown by both the Figure 5.1 and the Figure 5.2, PRIME has a small variation when bootstrapping, with values close to the Serf ones.

[1] https://cloud.google.com/
[2] Every degree of confidence in this thesis has a value of 95%.

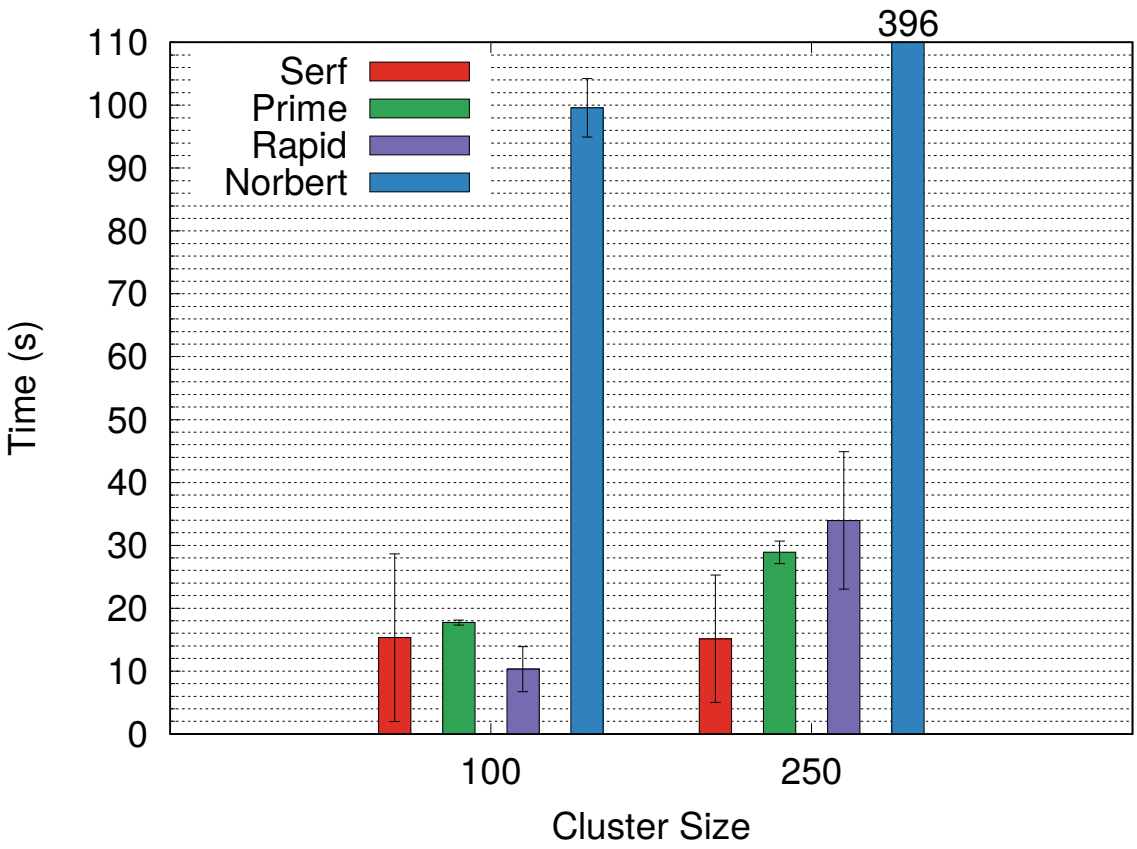

Figure 5.1: Bootstrap Time Comparison

Figure 5.2 shows a Cumulative Distribution Function (CDF) with respect to bootstrap time. Note that the values are different, since Figure 5.1 shows the average and confidence interval, whereas in Figure 5.2 we are presenting its best results. For instance, Rapid 250 nodes bootstrap values are significantly worse in Figure 5.1.

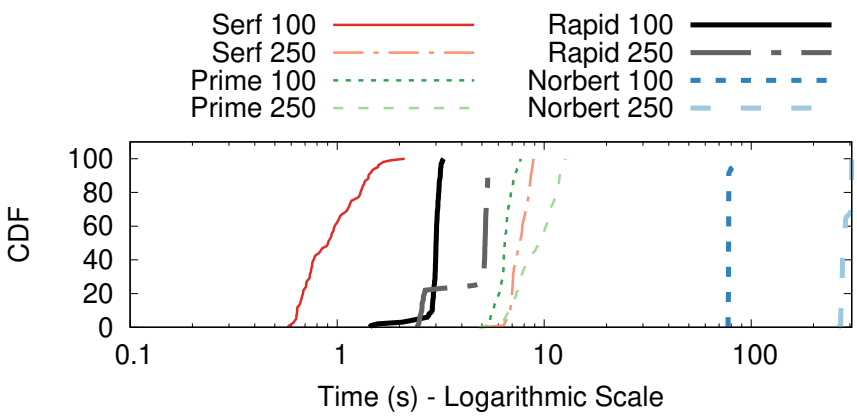

Figure 5.2: Bootstrap CDF

5.3.2 Reaction Results

Reaction scenarios assess the systems' speed of removing a single failed node. In Table 5.1 we present the reaction results for runs with 100 and 250 nodes.

As expected, PRIME has a values in between Serf and Norbert. PRIME's value will be decomposed and explained in Section 5.3.5.

During our experiments, we realized that sometimes Norbert does not detect any node failure. This is probably due to an implementation bug but we were unable to investigate it further. One example of this case can be seen in Table 5.1, where the Norbert 250 entry is not available.

System Size	Serf	Rapid	Norbert	PRIME
100	9,8	13,1	35,6	17,3
250	11,3	12,5	N.A.	17,9

Table 5.1: Reaction times in seconds

5.3.3 Churn Results

Churn scenarios test the systems' resilience in tolerating several nodes dying in a certain period. In practice, these scenarios wait for the systems to stabilize. Then, a certain percentage of nodes are killed every minute, for four minutes.

There are three types of churn tests, where the percentage of nodes killed every minute varies. These are 1%, 2% and 5%.

When comparing the three systems using a CDF, we conclude that Serf is the fastest, Norbert the slowest and PRIME and Rapid are in between those two. It is an expected result because PRIME and Rapid aim to an alternative approach that could be scalable, making it faster than Norbert, yet consistent, making it slower than Serf.

It is observable how Norbert degrades with system size, for instance, in Figure 5.3 the difference between 100 and 250 nodes is abysmal, while PRIME, Serf and Rapid change only slightly. If we increased the system size, Norbert would suffer immensely.

While we can observe a degraded behavior of Rapid with 250 nodes, we believe its due to CPU starvation. The degradation can be so severe that the failed nodes are kept in the views, as shown by its line overlapping the YY axis, in Figure 5.4 and Figure 5.5.

In short, a periodic 2% and 5% churn break Rapid's consistency.

5.3.4 Catastrophic Results

Catastrophic scenarios evaluate the protocols' tolerance when facing a significant failure. Each experiment consists in stabilizing a system and killing a large percentage of nodes.

There are three types of catastrophic tests, where the percentage of nodes killed differ. These are 10%, 20% and 30%.

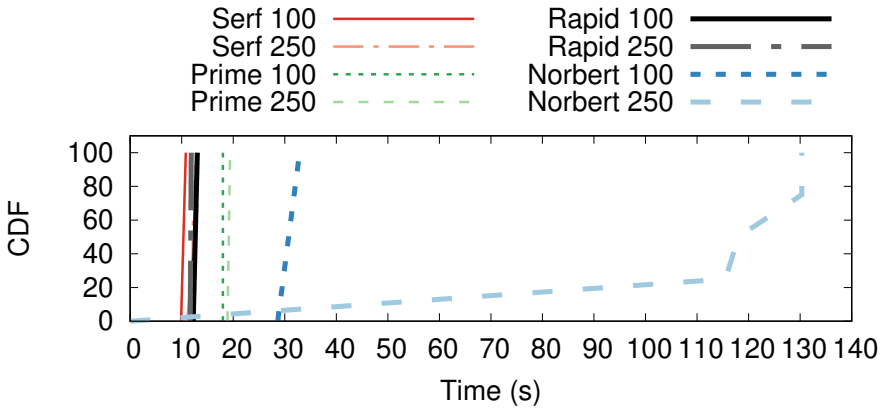

Figure 5.3: Churn 1% Removal CDF

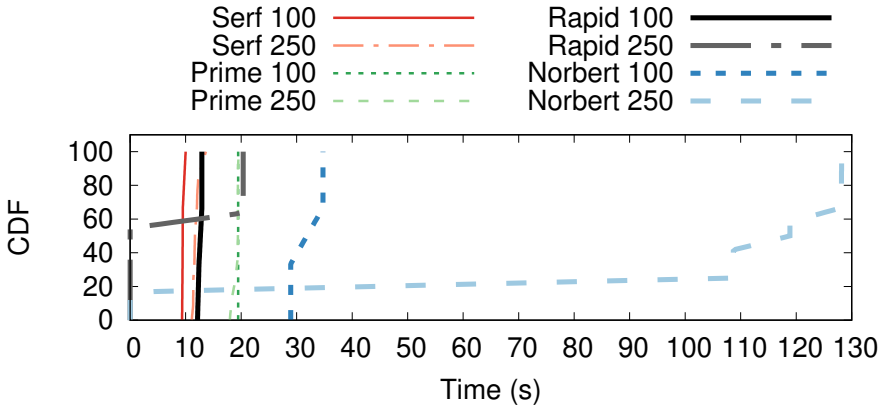

Figure 5.4: Churn 2% Removal CDF

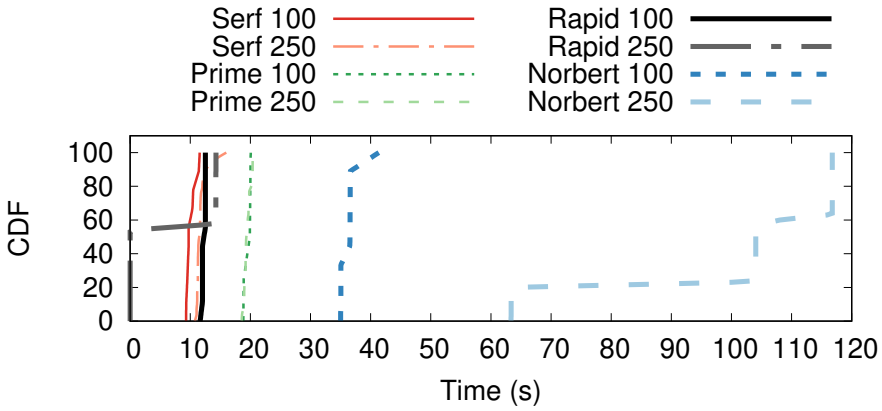

Figure 5.5: Churn 5% Removal CDF

Serf's excellent results are caused by the suspicion timeout mechanism built into SWIM, described in Section 2.3.1, which forces nodes to be considered dead if a node remains a suspect after a specified timeout.

Once more, Norbert's results show that it degrades quite rapidly with the size of the system.

Again, we can observe a degraded behavior of Rapid with 250 nodes. With a catastrophic failure of 10%, depicted in Figure 5.6, it did not converge[3].

[3]Part of the CDF curve is on top of YY axis

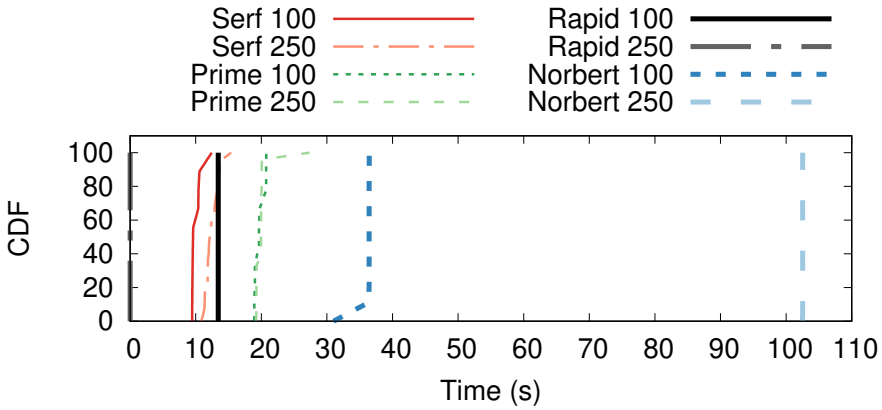

Figure 5.6: Catastrophic 10% Removal CDF

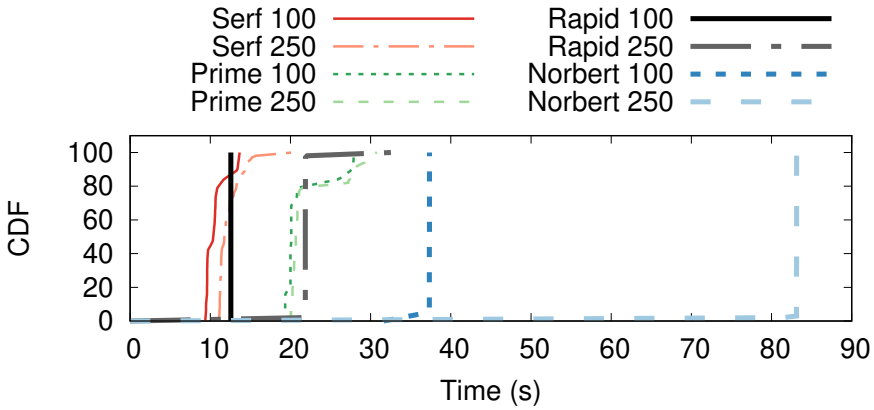

Figure 5.7: Catastrophic 20% Removal CDF

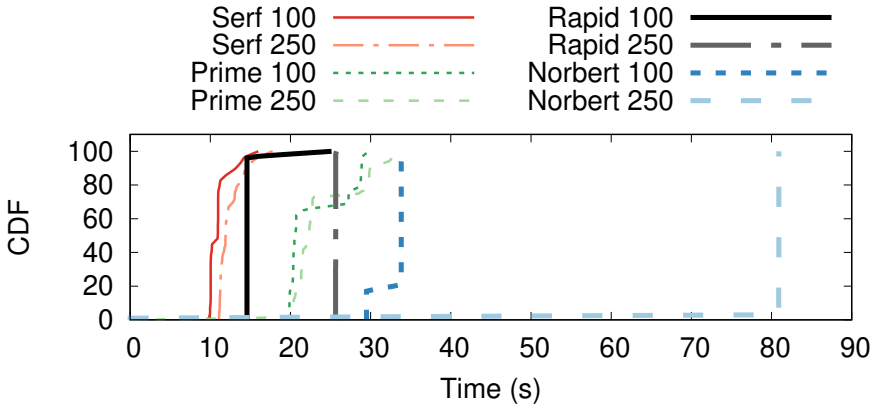

Figure 5.8: Catastrophic 30% Removal CDF

PRIME is the only system that has similar behavior between system sizes and tests, while ensuring a consistent view.

5.3.5 PRIME's Node Removal Time Decomposition

To detect and respond to a failure, PRIME has to reach several steps. The time taken to remove a dead node from the view since it died can be decomposed into two components:

(A) One takes into consideration the amount of time that the guardian nodes FDs take to trigger a suspicion – detection time.

(B) The other considers the time needed to make a decision and to install it in every view.

(B) is entirely related with PRIME, where the majority of time is spent disseminating the update[4].

(A) takes a big chunk of the total time. However since we can change the FD's implementation or its configuration, this chunk does not impact the algorithm performance directly when facing a dead node.

This decomposition is shown in Figure 5.9, where the purple and green block represents the time taken to detect a failed node. The purple segment represents the first node FD to detect a failure and the green section represents the last node needed to form a quorum.

The remaining colors represent PRIME's algorithm, where we emphasize the minimal time required to decide about the removal.

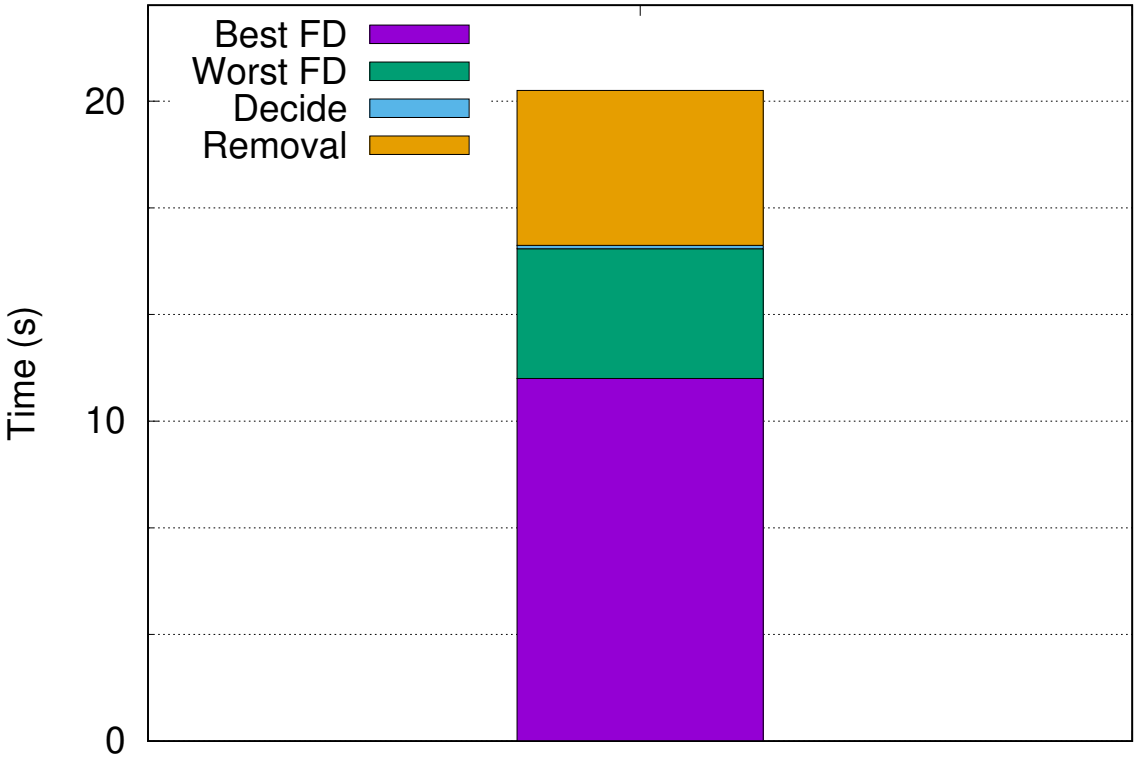

Figure 5.9: Failure Detection Decomposition

A system administrator can change the FD's configuration according to the target network. However, this is not an easy task, since a wrong configuration can lead to false positives, which causes system instability and incorrectness when keeping the membership.

If the network is stable, then the FD may tolerate lesser missing heartbeats, leading to a faster detection. Otherwise, the FD may tolerate more, leading to a slower detection.

[4]In fact, EpTO is responsible for this period. Nonetheless, EpTO is part of PRIME.

5.3.6 Views' Total Order Progression

The results presented in this section represent the number of distinct views delivered to the application, with the same local timestamp, in different nodes. It can be seen as the distance between views and it measures the consistency of a system.

A value of one shows that a system was consistent, with every node delivering the same view. A value above one is undesirable and represents a consistency violation. In practice, the bigger the value the worst.

In Figure 5.10, the bars represent the average distance and the error bars show the confidence interval with a degree of confidence of 95%. The average was calculated using all the runs in the churn and catastrophic failure scenarios. We used the views progression metric, which is computed by counting the number of distinct views with the same local timestamp, between nodes.

The results presented show that PRIME can achieve our major goal, while other systems cannot. Thus, PRIME is the only system that can progress in a total order fashion.

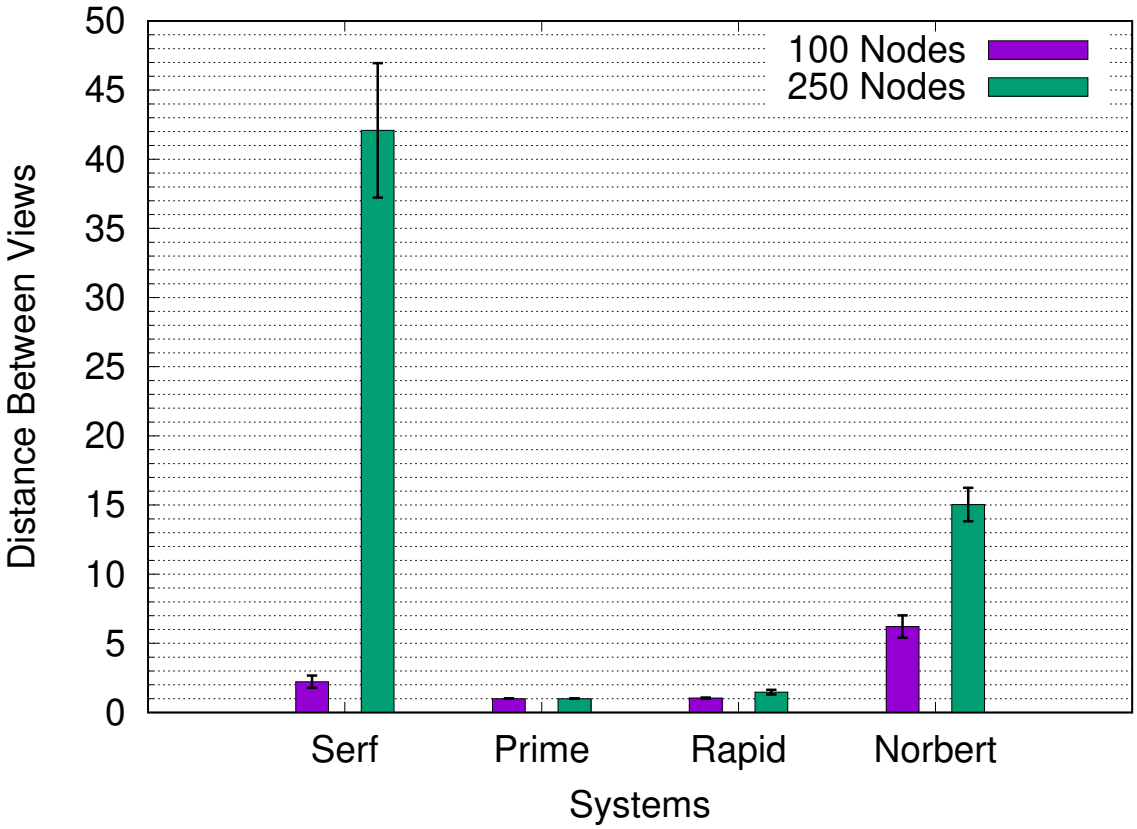

Figure 5.10: Distance Between Views

Serf's view progression value is expected, due to its weak guarantees. In contrast, we would expect Norbert – a ZooKeeper based solution – to have a view progression value of one as PRIME has.

However, due to the way ZooKeeper is designed, Norbert has to register a watch, each time it intends to receive a membership update. A watch is a one time trigger, meaning an application will need to set one again if it wants to receive another update. In this case, each time Norbert node gets a membership

update, it needs to set another watch.

In practice, this means that if a node N^{th} joins the system, ZooKeeper will need to handle $N-1$ watches being set. With a large enough system, it is expected an increase in latency in setting these watches. As a consequence, some nodes in Norbert may miss some update, destroying the strong consistency abstraction[46].

Regarding Rapid, with a size of 250 node, a few times consistency was not achieved. We interacted with the authors of Rapid about this issue but could not pinpoint the exact cause at the time of this writing[47].

5.3.7 Resource Usage

The resource usage results are important since an auxiliary system needs to leave a small usage footprint. In this section, we present an analysis of Memory, Network, and CPU usage, of the four systems.

Memory

Using the Java Virtual Machine (JVM)'s flag -Xlog:gc, we were able to get the minimum and maximum amount of memory used by the three Java solutions.

To get the memory usage of Serf, we periodically sent a USR1 signal to the process, making it dumping telemetry information[48].

The memory usage interval for each system, with 250 nodes, is summarized in Table 5.2.

System	Memory Usage (MB)	
Serf[5]	2,6	
Rapid	10	28
Norbert	9	71
PRIME	10	75

Table 5.2: Memory Usage of Tested Systems (Min/Max)

Serf is written in Go, making the memory usage very efficient. Norbert needs ZooKeeper and the results shown are concerning only the client. Rapid memory usage is better than PRIME, but as described next its CPU usage is greater.

Network

Regarding network usage, PRIME is the worst of all systems, due to its dissemination mechanism – EpTO. Serf and Norbert are the most inexpensive concerning network usage, as depicted in Figure 5.11.

Serf, Norbert and PRIME almost did not degrade from the system size increase. However, Rapid suffered quite a lot, as presented in Figure 5.12.

[5] Average usage

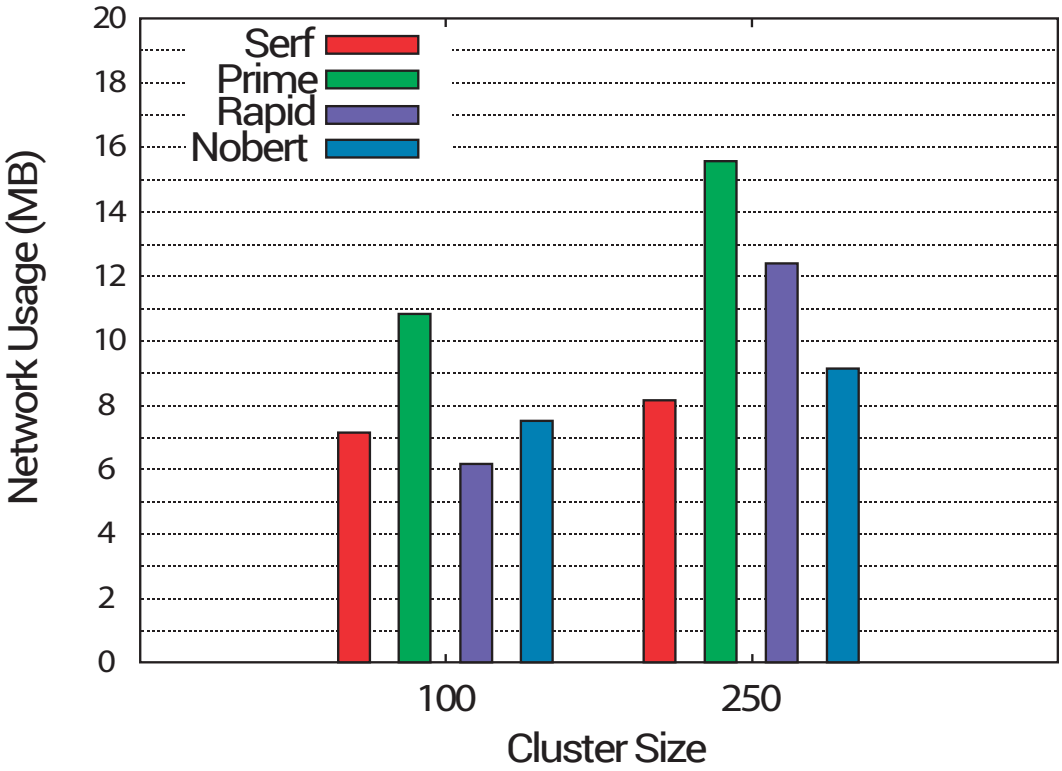

Figure 5.11: Network Usage Comparison

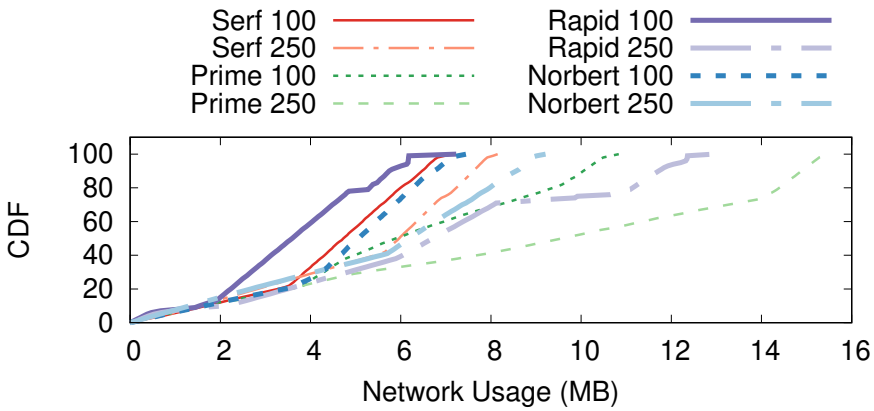

Figure 5.12: Network Usage CDF

Although PRIME's network usage is larger than the others, its grow is sublinear with the system's size, making it scalable.

CPU

Serf, Norbert and PRIME were able to run and make progress with each process limited to one-tenth of the CPU, whereas Rapid was not.

Rapid's unusual load is caused by a library used in its implementation[47] – gRPC. Table 5.3 shows the difference of resources that were needed to allocate between Serf, Nobert, PRIME, and Rapid. Note

that under the one-tenth CPU limitation, Rapid was unable to achieve any progress.

System	100 Nodes	250 Nodes
Serf	15 VMs	40 VMs
Rapid	50 VMs	100 VMs
Norbert	15 VMs	40 VMs
PRIME	15 VMs	40 VMs

Table 5.3: Resources difference between PRIME and Rapid

Rapid needs to have one CPU per process to properly function, which is unrealistic when considering an auxiliary service.

If Rapid were to be used in a real-world application, one CPU core would need to be dedicated to maintaining the system's membership and another core to function properly. In practice, a company deploying an application like this, it would double the expenses in Google Cloud[49].

PRIME may be slower, but it achieves consistency without the need for a dedicated core, keeping the costs down. Considering Table 5.3 again, we conclude that PRIME is between 60% and 70% less expensive than Rapid.

5.3.8 PRIME's Bootstrap Time Without CPU Cap

Because Rapid is very CPU intensive, it requires much more machines to run than PRIME. We compared Rapid against PRIME with the CPU uncapped, shown in Figure 5.13, to evaluate the behavior of PRIME.

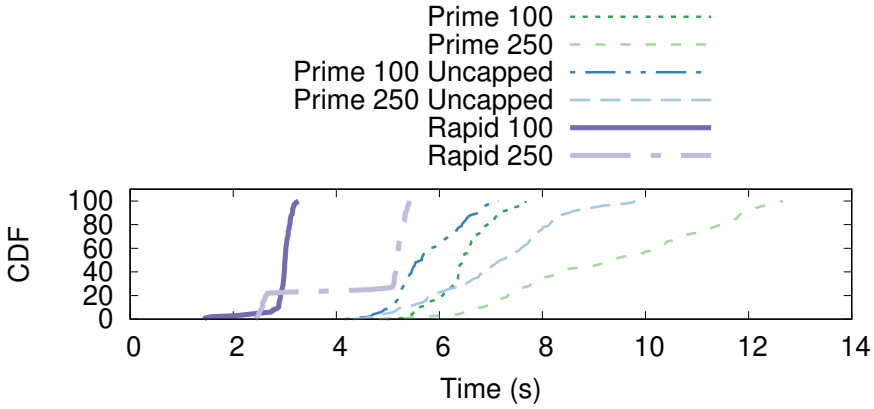

Figure 5.13: Bootstrap Time Comparison Without PRIME's CPU cap

The results of PRIME does not change significantly, because it does not suffer from CPU bottleneck as Rapid. Instead, as detailed in Section 5.3.7, PRIME suffers from network bottleneck.

5.4 Summary

In short, PRIME achieves its primary goal with good performance.

Despite being theoretically less consistent than a ZooKeeper approach, in practice, it can ensure stronger guarantees while having a much better performance. This is because, at the time of this writing, Norbert can miss some updates resulting in inconsistency.

We believe that PRIME has its place as a less expensive alternative to Rapid. PRIME might be slightly slower, but it is incredibly lightweight when compared to Rapid.

Clearly, PRIME can be improved in many ways, including its performance. Some considerations about the work ahead are discussed in Section 6.2.

We present a summary with respect to consistency, speed and resource usage, regarding the four systems, in Table 5.4. The represented scale is informal and can be sorted from worst to best in the following order: $--, -, +, ++$.

System	Consistency	Speed	Resource Usage[6]
Serf	$--$	$++$	$++$
Norbert	$-$	$--$	$+$
Rapid	$+$	$++$	$--$
PRIME	$++$	$+$	$+$

Table 5.4: Summary of Results

[6] $++$ is better.

Chapter 6

Conclusions

The problem of maintaining membership in large-scale systems is more pertinent now than ever before, with companies changing from data stores, such as Facebook, to ones that ensure just a little more consistency to ease the development of their products.

As a consequence, teams that dropped strong guarantees to ensure availability are currently trying to adopt different technologies which offer better guarantees. PRIME empowers organizations and people, by providing them a more consistent, yet scalable, mechanism to ground their applications.

With PRIME, we bring a fresh approach to membership services, by using a probabilistic total order algorithm which allows the total order of the updates and a probabilistic agreement among the nodes, with an arbitrarily high probability that can be tuned freely.

PRIME is the only solution that achieves consistency in all tested scenarios. It is slightly slower than weak consistency solutions and it has a decent resource usage. Given the obtained results, scaling from 100 to 250 nodes, we conclude that PRIME is scalable.

6.1 Achievements

We consider our most significant achievement to be the creation of a total view membership protocol, which offers a total order progression of views while making it scalable.

Whereas designing the solution was something natural, coding PRIME was difficult, due to its high concurrency and network component. Even so, we believe having an implementation, with good results, ready to be used by other application, is another significant achievement.

On a personal level, being able to apply the knowledge acquired in the last years and tampering around real distributed systems components like a failure detector or a dissemination mechanism, is something remarkable and eye-opening. Additionally, I had the possibility to exchange ideas with people with the same interests and far more experienced, improving my way of communicating and broadening horizons.

6.2 Future Work

As future work, there are five main directions where PRIME can expand.

Improving PRIME Run experiments on a larger scale since we could not increase the scale, due to budget constraints. This would help to discover implementation and performance issues and better assess how scalable and resilient PRIME is.

Apply PRIME Integrate PRIME in a real-world application like Apache Cassandra, to evaluate in practice the benefits of our membership service.

Use PRIME natively An entire distributed application, like a data store, could be developed using PRIME natively. By building an application on top of it, PRIME can be adapted to use its dissemination mechanism – EpTO – to disseminate application updates, increasing the application scalability.

Partition tolerance Developing a partition healing algorithm would improve PRIME to a whole new level. Right now, if a partition occurs, it will form two independent clusters.

Improving API Develop a queryable interface, increasing the abstractions provided to an application. For instance, one could ask PRIME what nodes are closer or what nodes are in the same region/data center.

During the development of PRIME, EpTO's authors identified some engineering issues that are causing the network bottleneck of PRIME. Currently, they are being solved, which will greatly benefit PRIME's network performance, in the future.

Bibliography

[1] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, 2010. ISSN 00010782. doi: 10.1145/1721654.1721672. URL http://portal.acm.org/citation.cfm?doid=1721654.1721672.

[2] Kevin Jones. Cloud computing: a business model game changer — Media Network — The Guardian, 2012. URL http://www.theguardian.com/media-network/media-network-blog/2012/mar/30/cloud-computing-business-model.

[3] O. Babaoglu, A. Bartoli, and G. Dini. Enriched view synchrony: a programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997. ISSN 0018-9340. doi: 10.1109/12.600823.

[4] J. Lewis. Microservices - A definition of this new architectural term, 2014. URL http://martinfowler.com/articles/microservices.html.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo. *ACM SIGOPS Operating Systems Review*, 41(6):205, 2007. ISSN 01635980. doi: 10.1145/1323293.1294281. URL http://portal.acm.org/citation.cfm?doid=1323293.1294281.

[6] Mazdak Hashemi. The Infrastructure Behind Twitter: Scale — Twitter Blogs. pages 1–16, 2017. URL https://blog.twitter.com/2017/the-infrastructure-behind-twitter-scale.

[7] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing - PODC '87*, pages 1–12, 1987. ISSN 01635980. doi: 10.1145/41840.41841. URL http://portal.acm.org/citation.cfm?doid=41840.41841.

[8] A. Lakshman and P. Malik. Cassandra. *ACM SIGOPS Operating Systems Review*, 44(2):35, 2010. ISSN 01635980. doi: 10.1145/1773912.1773922. URL http://portal.acm.org/citation.cfm?doid=1773912.1773922.

[9] M. Asay. Apple's secret NoSQL sauce includes a hefty dose of Cassandra, 2015. URL https://www.techrepublic.com/article/apples-secret-nosql-sauce-includes-a-hefty-dose-of-cassandra/.

[10] Netflix. NoSQL at Netflix. *Netflix*, 2011. URL https://medium.com/netflix-techblog/nosql-at-netflix-e937b660b4c.

[11] Apache. [CASSANDRA-9667] strongly consistent membership and ownership - ASF JIRA, . URL https://issues.apache.org/jira/browse/CASSANDRA-9667.

[12] K. Muthukkaruppan. The Underlying Technology of Messages, 2010. URL https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919{%}5Cnhttps://www.facebook.com/note.php?note{_}id=454991608919.

[13] Amazon. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. Technical report, 2011. URL http://aws.amazon.com/message/65648/.

[14] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. *Proceedings of the 6th ACM SIGCOMM on Internet measurement - IMC '06*, (August):189, 2006. doi: 10.1145/1177080.1177105. URL http://portal.acm.org/citation.cfm?doid=1177080.1177105.

[15] B. B. Glade, K. P. Birman, R. C. B. Cooper, and R. Van Renesse. Light-weight process groups in the Isis system. *Distributed Systems Engineering*, 1(1):29–36, 1993. ISSN 09671846. doi: 10.1088/0967-1846/1/1/004.

[16] M. K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996. ISSN 00985589. doi: 10.1109/32.481515. URL http://ieeexplore.ieee.org/document/481515/.

[17] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation SE - OSDI '06*, pages 335–350, 2006. ISSN <null>. URL citeulike-article-id:6502774{%}5Cnhttp://portal.acm.org/citation.cfm?id=1298487.

[18] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993. ISSN 00010782. doi: 10.1145/163298.163303. URL http://dl.acm.org/citation.cfm?id=163298.163303{%}5Cnhttp://hdl.handle.net/2060/19920024932.

[19] L. Suresh, D. Malkhi, P. Gopalan, I. P. Carreiro, and Z. Lokhandwala. Stable and Consistent Membership at Scale with Rapid. 2018. URL http://arxiv.org/abs/1803.03620.

[20] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006. ISSN 01782770. doi: 10.1007/s00446-006-0005-x.

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. ISSN 07342071. doi: 10.1145/279227.279229. URL http://portal.acm.org/citation.cfm?doid=279227.279229.

[22] Red Hat. JGroups - The JGroups Project, 2017. URL http://www.jgroups.org/.

[23] JBoss. JGroups Services. URL https://docs.jboss.org/jbossclustering/cluster{_}guide/5.1/html/jgroups.chapt.html.

[24] Bela Ban. Performance tests JGroups 2.5, 200. URL http://www.jgroups.org/perf/perf2007/Report.html.

[25] LinkedIn. Norbert is a cluster manager and networking layer built on top of Zookeeper. URL https://github.com/rhavyn/norbert.

[26] Uber. Ringpop — Ringpop 0.1.0 documentation. URL https://ringpop.readthedocs.io/en/latest/.

[27] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly-consistent Infection-style process group Membership protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 303–312, 2002. ISBN 0769515975. doi: 10.1109/DSN.2002.1028914.

[28] HashiCorp. Gossip Protocol - Serf by HashiCorp, . URL https://www.serf.io/docs/internals/gossip.html.

[29] A. Dadgar, J. Phillips, and J. Currey. Lifeguard: Local Health Awareness for More Accurate Failure Detection. *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W 2018*, pages 22–25, 2018. doi: 10.1109/DSN-W.2018.00017. URL http://arxiv.org/abs/1707.00788.

[30] HashiCorp. Gossip Protocol - Serf by HashiCorp, . URL https://www.serf.io/docs/internals/gossip.html.

[31] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. *Middleware'98*, pages 55–70, 1998. doi: 10.1007/978-1-4471-1283-9_4. URL http://link.springer.com/10.1007/978-1-4471-1283-9{_}4.

[32] S. Voulgaris, D. Gavidia, and M. Van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–216, 2005. ISSN 10647570. doi: 10.1007/s10922-005-4441-x.

[33] M. Matos, H. Mercier, P. Felber, R. Oliveira, and J. Pereira. EpTO. In *Proceedings of the 16th Annual Middleware Conference on - Middleware '15*, pages 100–111, New York, New York, USA, 2015. ACM Press. ISBN 9781450336185. doi: 10.1145/2814576.2814804. URL http://dl.acm.org/citation.cfm?doid=2814576.2814804.

[34] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The /spl phi/ accrual failure detector. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78, 2004. ISSN 1060-9857. doi: 10.1109/RELDIS.2004.1353004. URL http://ieeexplore.ieee.org/document/1353004/.

[35] Akka. Cluster Specification • Akka Documentation. URL http://doc.akka.io/docs/akka/snapshot/scala/common/cluster.html{#}cluster.

[36] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, page 15, 2011. doi: 10.1145/2043556.2043559. URL http://dl.acm.org/citation.cfm?doid=2043556.2043559.

[37] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data. *Source*, 2005.

[38] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003. ISSN 10636692. doi: 10.1109/TNET.2002.808407.

[39] Apache. NodeTool - Cassandra Wiki, . URL https://wiki.apache.org/cassandra/NodeTool.

[40] Apache. [CASSANDRA-7592] Ownership changes can violate consistency - ASF JIRA, . URL https://issues.apache.org/jira/browse/CASSANDRA-7592.

[41] G. Kakivaya, V. Modi, M. Mohsin, R. Kong, A. Ahuja, O. Platon, A. Wun, M. Snider, C. Daniel, D. Mastrian, Y. Li, L. Xun, A. Rao, V. Kidambi, R. Wang, A. Ram, S. Shivaprakash, R. Nair, A. Warwick, B. S. Narasimman, M. Lin, J. Chen, R. Hasha, A. B. Mhatre, P. Subbarayalu, M. Coskun, I. Gupta, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, and M. Fussell. Service fabric. *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*, (since 2007):1–15, 2018. doi: 10.1145/3190508.3190546. URL http://dl.acm.org/citation.cfm?doid=3190508.3190546.

[42] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In X. Défago, F. Petit, and V. Villain, editors, *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems*, number 11, page 17 pages, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 9781489979933. doi: 10.1007/978-3-642-24550-3. URL http://link.springer.com/10.1007/978-3-642-24550-3.

[43] S. Parafina. Improved Docker Container Integration with Java 10 - Docker Blog. URL https://blog.docker.com/2018/04/improved-docker-container-integration-with-java-10/.

[44] W. Working Group. Web Services Architecture, 2004. URL https://www.w3.org/TR/ws-arch/{#}whatis.

[45] Google. Compute Engine — Google Cloud Platform, 2014. URL https://cloud.google.com/products/compute-engine/.

[46] A. Zookeeper. ZooKeeper Programmer's Guide, 2012. URL https://zookeeper.apache.org/doc/r3.3.5/zookeeperProgrammers.html{#}ch{_}zkWatches.

[47] Docker integration · Issue #13 · lalithsuresh/rapid, 2018. URL https://github.com/lalithsuresh/rapid/issues/13.

[48] HashiCorp. Gossip Protocol - Serf by HashiCorp, . URL https://www.serf.io/docs/internals/gossip.html.

[49] Google. Google Compute Engine – Cloud Computing & IaaS – Google Cloud Platform, 2014. URL https://cloud.google.com/compute/pricing.