# PRIME : PRobabilistIc MEmbership – Large Scale Membership and Consistency

Extended Abstract

Francisco Santos
Advisors: Miguel Matos and Rodrigo Rodrigues
Instituto Superior Técnico, Universidade de Lisboa

***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 scales. 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.**

**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.**

## I. 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.

This shift in business model took less than a decade and led small clusters using view synchrony [1] into large ones using weaker models, to achieve the desired performance. New architectures like microservices [2] also aim to scale more. The popularity of these models led to the creation of technologies like Amazon DynamoDB [3] and Akka[1].

Apache Cassandra[2] [4] is an example of a storage system, used in real-world companies, like Apple [5] or Netflix [6]. Cassandra is decentralized fault-tolerant NoSQL database built on top of a Distributed Hash Table (DHT), where data is replicated among nodes according to a specified replication factor.

Cassandra has open tickets, in particular, one [7] 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.

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.

Membership is a fundamental building block of some distributed systems. We present **PRIME**. An decentralized highly scalable membership service with the probabilistic consistency, maintaining a total view and efficient. It features churn [8] tolerance and catastrophic failures resilience.

## II. Background

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. These systems ensure a correct total view of the system at the expense of performance.

Norbert [9] is a membership protocol, made by LinkedIn, built using Apache ZooKeeper. However, Norbert scalability is compromised due to ZooKeeper's design. In addition, Horus [10] and JGroups [11] fall into this category.

On the other hand, membership systems sacrifice consistency to improve performance. This means that each node

[1]https://akka.io/
[2]http://cassandra.apache.org/

may have different views of the same system, which can be undesirable for some applications. Due to their high scalability and resilience, these protocols are widely used in nowadays systems like Cassandra [4], Ringpop [12] and Serf [13] which is based on an optimized implementation of SWIM [14] [15].

New alternative approaches try to merge the two philosophies described above, aiming to develop a membership which is consistent and efficient, like Rapid [16]. We fall into that category, but following a different approach.

We take advantage of EpTO [17], a total order dissemination algorithm that features probabilistic agreement. The agreement probability can be tuned to the point that it is more likely for a hardware failure to happen than a total order violation.

## III. PRIME's Architecture

In this section, we present PRIME's architecture. It assumes a crash failure model, asynchronous nodes and network and fair-lossy links.

### A. Architectural Overview

PRIME has three main components. The **Membership Manager** is the component responsible for maintaining the system's total view as a ring. The **Hole Manager** has a role in handling holes caused by EpTO's algorithm. Finally, the **Failure Manager** handles all failure detection and the decision of a dead node's removal of the system.

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 Peer Sampling Service (PSS), which in practice is an implementation of CYCLON [18].

A simple graphical representation of the interaction between components can be seen in Figure 1.

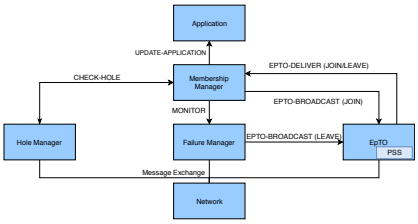


Fig. 1: Architecture Overview

### B. 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 III-B2.

*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 III-C1.

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 and since the view has changed the nodes monitored may change as well, so the Failure Manager 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 and behave like the other nodes.

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 I 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 [19], 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.

*2) Hole Handling:* 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.

**Algorithm 1:** Membership Manager (node n)

```
1  initially total-view ← {}
2  initialized ← false
3  update-queue ← []
4  ooo-control ← {}
5  SEND DISCOVERY(n) TO random-node // A random
     node already in the system
6
7
8  upon receiving DISCOVERY (node)
9      if total-view.initialized = true then
10         update.command ← JOIN
11         update.node ← node
12         update.hash-sent ← HASH(total-view)
13         EPTO-BROADCAST (update)
14 procedure PROCESS-UPDATE (update,
     is-out-of-order)
15     CHECK-HOLE (update)
16     if update.command = LEAVE then
17         total-view ← total-view \ {update}
18         ooo-control ← ooo-control ∪ {update}
19     else
20         if not is-out-of-order ∧ update ∈ ooo-control
             then
21             total-view ← total-view ∪ {update}
22     UPDATE-APPLICATION (update)
23     MONITOR (total-view)
24 procedure SEND-VIEW (node)
25     if p = FIRST-GUARDIAN (node) then
26         SEND VIEW(total-view) TO node
27 upon receiving VIEW (view-delivered)
28     total-view ← view-delivered
29     forall update in update-queue do
           // Process all updates that were stored in the
              queue, while a view's copy was not sent
30         PROCESS-UPDATE (update, false)
31     MONITOR (view)
32     total-view.initialized ← true
33 upon EPTO-DELIVER (update)
34     if total-view.initialized = true then
35         PROCESS-UPDATE (update, false)
36         SEND-VIEW (update.node)
37     else
38         update-queue ← update
39 upon EPTO-DELIVER-OUT-OF-ORDER (update)
40     if total-view.initialized = true then
41         PROCESS-UPDATE (update, true)
42     else
43         update-queue ← update
```

**Algorithm 2:** Membership Manager (node n) 2

```
1  procedure UPDATE-APPLICATION ()
       // Updates application
2  procedure FIRST-GUARDIAN (node)
       // Returns first guardian of node
```

All updates have a hash of the view that was installed when it was broadcasted. Using that remote hash and following Algorithm 3, 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 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).

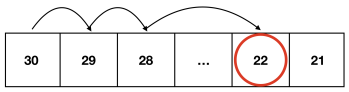


Fig. 2: Finding the missing timestamp

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 3, 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.

### C. Monitoring

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.

TABLE I: Out of order deliver cases

| 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. |

*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 and node $p$ as an example, blue nodes are Guardians monitoring $p$ and red nodes are Protegees monitored by $p$.

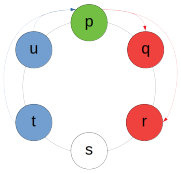


Fig. 3: Logical ring from the point of view of node $p$: $\#guardians = \#protegees = 2$

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

*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 4, 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 4, line 6).

Whenever a node is detected as failed by a Guardian's Failure Detector (FD), it sends a SUSPICION message to the other guardians of the suspected node (Algorithm 4, line 24).

All Guardians keep collecting SUSPICION messages until either they receive an ALIVE message (Algorithm 5, line 5), meaning a Guardian detected that the node is not failed, or more than half of the number of guardians of SUSPICION messages (Algorithm 5, 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.

## IV. Implementation

PRIME is implemented as a Java 10 library and we present PRIME's API in Listing 1. It is open-sourced under an Apache 2 license[3].

```
// 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 1: API

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.

## V. Optimizations

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

### A. Failure Detector State Transfer

The approach described in III-C can be optimized to reduce removal times. Each Protegee has a corresponding FD keeping state about it. This state can be kept for Protegees that remain after the new view is installed. Thus, the suspicion mechanism is triggered almost immediately making the removal of failed nodes faster.

### B. 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, it was inefficient, so we changed the original implementation into a TreeSet since it orders elements and it avoids duplicates by default.

[3]It is private at the moment. https://github.com/francisco-polaco/prime-probabilistic-membership

**Algorithm 3:** 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
```

**Algorithm 4:** 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 5:** 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
```

### C. View's Hash Caching

Hashing the current view is something often calculated, since it is the hole detection mechanism, as explained in Section III-B2. 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.

### D. Monitor

The MONITOR method described at Algorithm 4 can be extremely inefficient. 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.

## VI. EVALUATION

We choose metrics and testing environments that have two goals. On the one hand, they help to show that PRIME approach is correct, that is 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 [13], Rapid [16] and Norbert [9] representing strong and weak consistency protocols.

### A. 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 usage and CPU usage.

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 – some metrics considered are similar to ones in other works.

### B. 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[4] using n1-standard-1 [20] instances 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 VI-C7c, we make some considerations about resource usage.

[4]https://cloud.google.com/

### C. Results

In this section, we present data comparing Serf, Rapid, Norbert and PRIME. We denote each experiment as System Size, for instance, PRIME 100 is a PRIME experience with 100 nodes.

*1) Update Installation During Bootstrap:* In Figure 4, 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%[5].

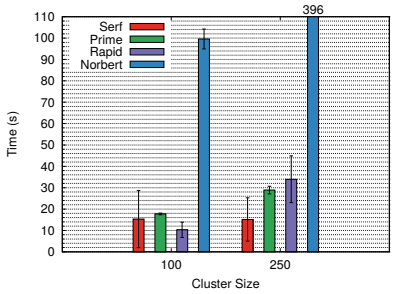


Fig. 4: Bootstrap Time Comparison

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 VI-C6, 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 4 and the Figure 5, PRIME has a small variation when bootstrapping, with values close to the Serf ones.

Figure 5 shows a Cumulative Distribution Function (CDF) with respect to bootstrap time. Note that the values are different, since we are showing the average and confidence interval, whereas in Figure 5 we are presenting its best results. For instance, Rapid 250 nodes bootstrap values are significantly worse in Figure 4.

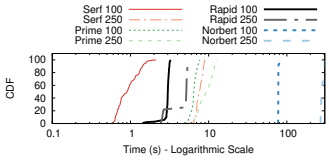


Fig. 5: Bootstrap CDF

*2) Reaction Results:* Reaction scenarios assess the systems' speed of removing a single failed node. In Table II we present the reaction results for runs with 100 and 250 nodes.

[5]Every degree of confidence in this thesis has a value of 95%.

As expected, PRIME has a values in between Serf and Norbert. PRIME's value will be decomposed and explained in Section VI-C5.

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

TABLE II: Reaction times in seconds

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

*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 6 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 7 and Figure 8.

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

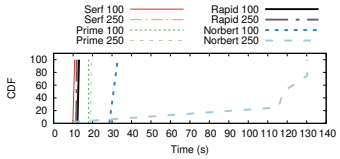


Fig. 6: Churn 1% Removal CDF

*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%.

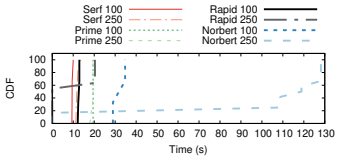


Fig. 7: Churn 2% Removal CDF

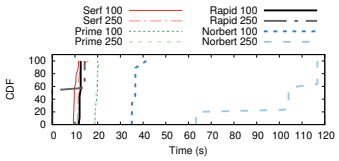


Fig. 8: Churn 5% Removal CDF

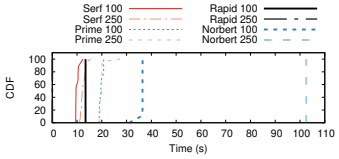


Fig. 9: Catastrophic 10% Removal CDF

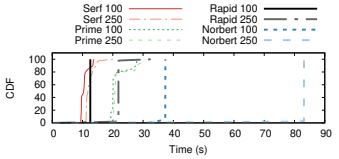


Fig. 10: Catastrophic 20% Removal CDF

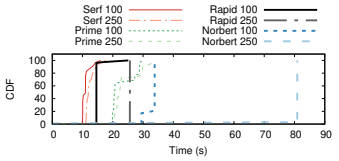


Fig. 11: Catastrophic 30% Removal CDF

Serf's excellent results are caused by the suspicion timeout mechanism built into SWIM [14], 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 9, it did not converge[6].

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

*5) PRIME's Node Removal Time Decomposition:* The time taken to remove a dead node from the view since it died can be decomposed into two segments:

(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[7].

(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 12, 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.

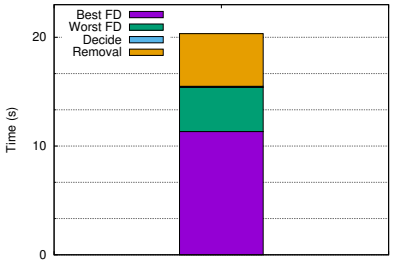


Fig. 12: Failure Detection Decomposition

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.

*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 13, 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.

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

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

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.

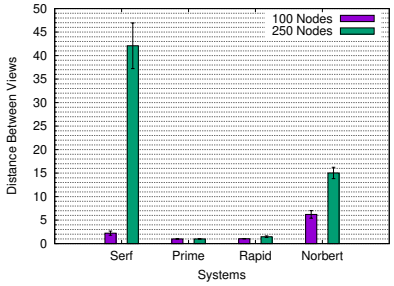


Fig. 13: Distance Between Views

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 $N^{\text{th}}$ 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 [21].

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 [22].

*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.

*a) 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 [23].

With 250 nodes, the memory usage interval for Rapid varies from 10 to 28 MB, Norbert from 9 to 71 MB and PRIME from 10 to 75 MB. Serf uses an average of 2,5 MB.

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.

*b) 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 14.

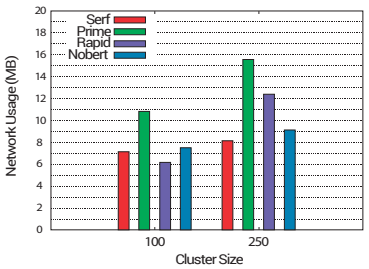


Fig. 14: Network Usage Comparison

Serf, Norbert and PRIME almost did not degrade from the system size increase. However, Rapid suffered quite a lot, as presented in Figure 15. Although PRIME's network usage is larger than the others, its grow is sublinear with the system's size, making it scalable.

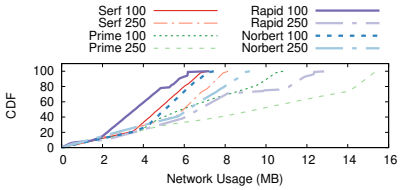


Fig. 15: Network Usage CDF

*c) 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 [22] – gRPC. Table III 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.

TABLE III: Resources difference between PRIME and Rapid

| 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 |

So 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 to function properly. In practice, a company deploying an application like this, it would double the expenses in Google Cloud [24].

Considering Table III again, we conclude that PRIME is between 60% and 70% less expensive than Rapid.

### *D. 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 16, to evaluate the behavior of PRIME.

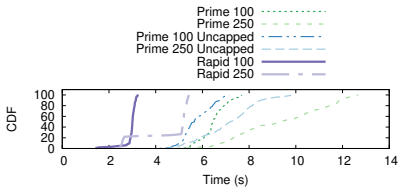


Fig. 16: Bootstrap Time Comparison Without PRIME's CPU cap

The results of PRIME does not change significantly, because it does not suffer from CPU bottleneck as Rapid. Instead, as detailed in Section VI-C7, PRIME suffers from network bottleneck.

### E. Summary

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

TABLE IV: Summary of Results

| System | Consistency | Speed | Resource Usage[8] |
|---|---|---|---|
| Serf | $--$ | $++$ | $++$ |
| Norbert | $-$ | $--$ | $+$ |
| Rapid | $+$ | $++$ | $--$ |
| **PRIME** | $++$ | $+$ | $+$ |

## VII. Conclusion

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.

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.

## References


[1] O. Babaoglu, A. Bartoli, and G. Dini, "Enriched view synchrony: a programming paradigm for partitionable asynchronous distributed systems," *IEEE Transactions on Computers*, vol. 46, no. 6, pp. 642–658, June 1997.

[2] J. Lewis, "Microservices - A definition of this new architectural term," pp. 1–16, 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[3] 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*, vol. 41, no. 6, p. 205, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1323293.1294281

[4] A. Lakshman and P. Malik, "Cassandra," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 35, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1773912.1773922

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

[6] Netflix, "NoSQL at Netflix," *Netflix*, 2011. [Online]. Available: https://medium.com/netflix-techblog/nosql-at-netflix-e937b660b4c

[7] Apache, "[CASSANDRA-9667] strongly consistent membership and ownership - ASF JIRA." [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-9667

[8] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," *Proceedings of the 6th ACM SIGCOMM on Internet measurement - IMC '06*, no. August, p. 189, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1177080.1177105

[9] LinkedIn, "Norbert is a cluster manager and networking layer built on top of Zookeeper." [Online]. Available: https://github.com/rhavyn/norbert

[10] 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*, vol. 1, no. 1, pp. 29–36, 1993.

[11] Red Hat, "JGroups - The JGroups Project," 2017. [Online]. Available: http://www.jgroups.org/

[12] Uber, "Ringpop — Ringpop 0.1.0 documentation." [Online]. Available: https://ringpop.readthedocs.io/en/latest/

[13] HashiCorp, "Gossip Protocol - Serf by HashiCorp." [Online]. Available: https://www.serf.io/docs/internals/gossip.html

[14] 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*, 2002, pp. 303–312.

[15] 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*, pp. 22–25, 2018. [Online]. Available: http://arxiv.org/abs/1707.00788

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

[17] M. Matos, H. Mercier, P. Felber, R. Oliveira, and J. Pereira, "EpTO," in *Proceedings of the 16th Annual Middleware Conference on - Middleware '15*. New York, New York, USA: ACM Press, 2015, pp. 100–111. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2814576.2814804

[18] S. Voulgaris, D. Gavidia, and M. Van Steen, "CYCLON: Inexpensive membership management for unstructured P2P overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–216, 2005.

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free Replicated Data Types," in *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds., no. 11. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, p. 17 pages. [Online]. Available: http://link.springer.com/10.1007/978-3-642-24550-3

[20] Google, "Compute Engine — Google Cloud Platform," 2014. [Online]. Available: https://cloud.google.com/products/compute-engine/

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

[22] "Docker integration · Issue #13 · lalithsuresh/rapid," 2018. [Online]. Available: https://github.com/lalithsuresh/rapid/issues/13

[23] HashiCorp, "Gossip Protocol - Serf by HashiCorp." [Online]. Available: https://www.serf.io/docs/internals/gossip.html

[24] Google, "Google Compute Engine – Cloud Computing & IaaS – Google Cloud Platform," 2014. [Online]. Available: https://cloud.google.com/compute/pricing


[8] $++$ is better.