Stable and Consistent Membership at Scale with Rapid

Lalith Suresh[†], Dahlia Malkhi[†], Parikshit Gopalan[†], Ivan Porto Carreiro[◇], Zeeshan Lokhandwala[‡]

[†]VMware Research, [‡]VMware, [◇]One Concern

Abstract

We present the design and evaluation of Rapid, a distributed membership service. At Rapid's core is a scheme for *multi-process cut detection* (CD) that revolves around two key insights: (i) it suspects a failure of a process only after alerts arrive from multiple sources, and (ii) when a group of processes experience problems, it detects failures of the entire group, rather than conclude about each process individually. Implementing these insights translates into a simple membership algorithm with low communication overhead.

We present evidence that our strategy suffices to drive unanimous detection almost-everywhere, even when complex network conditions arise, such as one-way reachability problems, firewall misconfigurations, and high packet loss. Furthermore, we present both empirical evidence and analyses that proves that the almost-everywhere detection happens with high probability. To complete the design, Rapid contains a leaderless consensus protocol that converts multi-process cut detections into a view-change decision. The resulting membership service works both in fully decentralized as well as logically centralized modes.

We present an evaluation of Rapid in moderately scalable cloud settings. Rapid bootstraps 2000 node clusters 2-5.8x faster than prevailing tools such as Memberlist and ZooKeeper, remains stable in face of complex failure scenarios, and provides strong consistency guarantees. It is easy to integrate Rapid into existing distributed applications, of which we demonstrate two.

1 Introduction

Large-scale distributed systems today need to be provisioned and resized quickly according to changing demand. Furthermore, at scale, failures are not the exception but the norm [21, 30]. This makes membership management and failure detection a critical component of any distributed system.

Our organization ships standalone products that we do not operate ourselves. These products run in a wide range of enterprise data center environments. In our experience, many failure scenarios are not always crash failures, but commonly involve misconfigured firewalls, one-way connectivity loss, flip-flops in reachability, and some-but-not-all packets being dropped (in line with observations by [49, 19, 37, 67, 41]). We find that existing membership solutions struggle with these common failure scenarios, despite being able to cleanly detect crash faults. In particular, existing tools take long to, or never converge to, a stable state where the faulty processes are removed (§2.1).

We posit that despite several decades of research and production systems, stability and consistency of existing membership maintenance technologies remains a challenge. In this paper, we present the design and implementation of *Rapid*, a scalable, distributed membership system that provides both these properties. We discuss the need for these properties below, and present a formal treatment of the service guarantees we require in §3.

Need for stability. Membership changes in distributed systems trigger expensive recovery operations such as failovers and data migrations. Unstable and flapping membership views therefore cause applications to repeatedly trigger these recovery workflows, thereby severely degrading performance and affecting service availability. This was the case in several production incidents reported in the Cassandra [10, 9] and Consul [26, 25, 27] projects. In an end-to-end experiment, we also observed a 32% increase in throughput when replacing a native system's failure detector with our solution that improved stability (see §7 for details).

Furthermore, failure recovery mechanisms may be faulty *themselves* and can cause catastrophic failures when they run amok [42, 40]. Failure recovery workflows being triggered ad infinitum have led to Amazon EC2 outages [5, 6, 4], Microsoft Azure outages [7, 48], and "killer bugs" in Cassandra and HBase [39].

Given these reasons, we seek to avoid frequent oscillations of the membership view, which we achieve through stable failure detection.

Need for consistent membership views. Many systems require coordinated failure recovery, for example, to correctly handle data re-balancing in storage systems [3, 28]. Consistent changes to the membership view simplify reasoning about system behavior and the development of dynamic reconfiguration mechanisms [65].

Conversely, it is challenging to build reliable clustered services on top of a weakly consistent membership ser-

vice [11]. Inconsistent view-changes may have detrimental effects. For example, in sharded systems that rely on consistent hashing, an inconsistent view of the cluster leads to clients directing requests to servers that do not host the relevant keys [12, 3]. In Cassandra, the lack of consistent membership causes nodes to duplicate data re-balancing efforts when concurrently adding nodes to a cluster [11] and also affects correctness [12]. To work around the lack of consistent membership, Cassandra ensures that only a single node is joining the cluster at any given point in time, and operators are advised to wait *at least two minutes* between adding each new node to a cluster [11]. As a consequence, bootstrapping a 100 node Cassandra cluster takes three hours and twenty minutes, thereby significantly slowing down provisioning [11].

For these reasons, we seek to provide strict consistency, where membership changes are driven by agreement among processes. Consistency adds a layer of safety above the failure detection layer and guarantees the same membership view to all non-faulty processes.

Our approach

Rapid is based on the following fundamental insights that bring stability and consistency to *both* decentralized and logically centralized membership services:

Expander-based monitoring edge overlay. To scale monitoring load, Rapid organizes a set of processes (a *configuration*) into a stable failure detection topology comprising *observers* that monitor and disseminate reports about their communication *edges* to their *subjects*. The monitoring relationships between processes forms a directed expander graph with strong connectivity properties, which ensures with a high probability that healthy processes detect failures. We interpret multiple reports about a subject's edges as a high-fidelity signal that the subject is faulty.

Multi-process cut detection. For stability, processes in Rapid (i) suspect a faulty process p only upon receiving alerts from multiple observers of p, and (ii) delay acting on alerts about different processes until the churn stabilizes, thereby converging to detect a global, possibly multi-node *cut* of processes to add or remove from the membership. This filter is remarkably simple to implement, yet it suffices by itself to achieve *almost-everywhere agreement* – unanimity among a large fraction of processes about the detected cut.

Practical consensus. For consistency, we show that converting almost-everywhere agreement into full agreement is practical even in large-scale settings. Rapid's consensus protocol drives configuration changes by a low-overhead, leaderless protocol in the common case: every process simply validates consensus by counting the number of identical cut detections. If there is a quorum containing three-quarters of the membership set with the same cut, then without a leader or further communication, this is a safe consensus decision.

Rapid thereby ensures all participating processes see a strongly consistent sequence of membership changes to the cluster, while ensuring that the system is stable in the face of a diverse range of failure scenarios.

In summary, we make the following key contributions:

- Through measurements, we demonstrate that prevailing membership solutions guarantee neither stability nor consistency in the face of complex failure scenarios.
- We present the design of Rapid, a scalable membership service that is robust in the presence of diverse failure scenarios while providing strong consistency. Rapid runs both as a decentralized as well as a logically centralized membership service.
- In system evaluations, we demonstrate how Rapid, despite offering much stronger guarantees, brings up 2000 node clusters 2-5.8x faster than mature alternatives such as Memberlist and ZooKeeper. We demonstrate Rapid's robustness in the face of different failure scenarios such as simultaneous node crashes, asymmetric network partitions and heavy packet loss. Rapid achieves these goals at a similar cost to existing solutions.
- Lastly, we report on our experience running Rapid to power two applications; a distributed transactional data platform and a service discovery use case.

2 Motivation and Related work

Membership solutions today fall into two categories. They are either managed for a cluster through an auxiliary service [15, 43], or they are gossip-based and fully decentralized [45, 44, 8, 69, 62, 59, 70, 64].

We studied how three widely adopted systems behave in the presence of network failure scenarios: *(i)* of the first category, ZooKeeper [15], and of the second, *(ii)* Memberlist [47], the membership library used by Consul [45] and Serf [44] and *(iii)* Akka Cluster [69] (see §7 for the detailed setup). For ZooKeeper and Memberlist, we bootstrap a 1000 process cluster with standalone agents that join and maintain membership using these solutions (for Akka Cluster, we use 400 processes because it began failing for cluster sizes beyond 500). We then drop 80% of packets for 1% of processes, simulating high packet loss scenarios described in the literature [19, 49] that we have also observed in practice.

Figure 1 shows a timeseries of membership sizes, as viewed by each non-faulty process in the cluster (every dot indicates a single measurement by a process). Akka Cluster is unstable as conflicting rumors about processes propagate in the cluster concurrently, even resulting in benign processes being removed from the membership. Memberlist and ZooKeeper resist removal of the faulty processes from the membership set but are unstable over

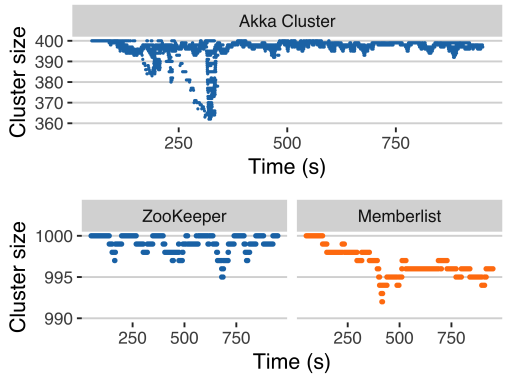

Figure 1: Akka Cluster, ZooKeeper and Memberlist exhibit instabilities and inconsistencies when 1% of processes experience 80% packet loss (similar to scenarios described in [19, 49]). Every process logs its own view of the cluster size every second, shown as one dot along the time (X) axis. Note, the y-axis range does not start at 0. X-axis points (or intervals) with different cluster size values represent inconsistent views among processes at that point (or during the interval).

a longer period of time. We also note extended periods of inconsistencies in the membership view.

Having found existing membership solutions to be unstable in the presence of typical network faults, we now proceed to discuss the broader design space.

2.1 Comparison of existing solutions

There are three membership service designs in use today, each of which provides different degrees of resiliency and consistency.

Logically centralized configuration service. A common approach to membership management in the industry is to store the membership list in an auxiliary service such as ZooKeeper [15], etcd [33], or Chubby [23].

The main advantage of this approach is simplicity: a few processes maintain the ground truth of the membership list with strong consistency semantics, and the remaining processes query this list periodically.

The key shortcoming here is that relying on a small cluster reduces the overall resiliency of the system: connectivity issues to the cluster, or failures among the small set of cluster members themselves, may render the service unavailable (this led Netflix to build solutions like Eureka [58, 61]). As the ZooKeeper developers warn, this also opens up new failure modes for applications that depend on an auxiliary service for membership [17].

Gossip-based membership. van Renesse et al. [72, 71] proposed managing membership by using gossip to spread positive notifications (keepalives) between all processes. If a process p fails, other processes eventually remove p after a timeout. SWIM [29] was proposed as a variant of that approach that reduces the communication overhead; it uses gossip to spread "negative" alerts, rather than regular positive notifications.

Gossip-based membership schemes are widely adopted in deployed systems today, such as Cassandra [8], Akka [69], ScyllaDB [64], Serf [44], Redis Cluster [62], Orleans [59], Uber's Ringpop [70], Netflix's Dynomite [56], and some systems at Twitter [55].

The main advantage of gossip-based membership is resiliency and graceful degradation (they tolerate $N-1$ failures). The key disadvantages include their weak consistency guarantees and the complex emergent behavior that leads to stability problems.

Stability is a key challenge in gossip-based membership: When communication fails between two processes which are otherwise live and correct, there are repeated accusations and refutations that may cause oscillations in the membership views. As our investigation of leading gossip-based solutions showed (Figure 1), these conflicting alerts lead to complex emergent behavior, making it challenging to build reliable clustered services on top of. Indeed, stability related issues with gossip are also observed in production settings (see, e.g., Consul [26, 25, 27] and Cassandra [11, 12, 10]).

Lastly, FireFlies [50] is a decentralized membership service that tolerates Byzantine members. FireFlies organizes monitoring responsibilities via a randomized k-ring topology to provide a robust overlay against Byzantine processes. While the motivation in FireFlies was different, we believe it offers a solution for stability; accusations about a process by a potentially Byzantine monitor are not acted upon until a conservative, fixed delay elapses. If a process does not refute an accusation about it within this delay, it is removed from the membership. However, the FireFlies scheme is based on a gossip-style protocol involving accusations, refutations, rankings, and disabling (where a process p announces that a monitor should not complain about it). Furthermore, FireFlies' refutations resist process removals as much as possible, which is undesirable in non-Byzantine settings. For example, in the 80% packet loss scenario described in Figure 1, a faulty process p may still succeed in disseminating refutations, thereby resisting removal from the membership. As we show in upcoming sections, our scheme is simple in comparison and requires little bookkeeping per process. Unlike FireFlies, we aggregate reports about a process p from multiple sources to decide whether to remove p, enabling timely and coordinated membership changes with low overhead.

Group membership. By themselves, gossip-based membership schemes do not address consistency, and allow the membership views of processes to diverge. In this sense, they may be considered more of failure detec-

tors, than membership services.

Maintaining membership with strict consistency guarantees has been a focus in the field of fault tolerant state-machine replication (SMR), starting with early foundations of SMR [52, 60, 63], and continuing with a variety of group communication systems (see [24] for a survey of GC works). In SMR systems, membership is typically needed for selecting a unique primary and for enabling dynamic service deployment. Recent work on Census [28] scales dynamic membership maintenance to a locality-aware hierarchy of domains. It provides fault tolerance by running the view-change consensus protocol only among a sampled subset of the membership set.

These methods may be harnessed on top of a stable failure detection facility, stability being orthogonal to the consistency they provide. As we show, our solution uses an SMR technique that benefits from stable failure detection to form fast, leaderless consensus.

3 The Rapid Service

Our goal is to create a membership service based on techniques that apply equally well to both decentralized as well as logically centralized designs. For ease of presentation, we first describe the fully decentralized Rapid service and its properties in this section, followed by its design in §4. We then relax the resiliency properties in §5 for the logically centralized design.

API Processes use the membership service by using the Rapid library and invoking a call JOIN(HOST:PORT, SEEDS, VIEW-CHANGE-CALLBACK). Here, HOST:PORT is the process' TCP/IP listen address. Internally, the join call assigns a unique logical identifier for the process (ID). If a process departs from the cluster either due to a failure or by voluntarily leaving, it rejoins with a new ID. This ID is internal to Rapid and is not an identifier of the application that is using Rapid. SEEDS is an initial set of process addresses known to everyone and used to contact for bootstrapping. VIEW-CHANGE-CALLBACK is used to notify applications about membership change events.

Configurations A configuration in Rapid comprises a configuration identifier and a *membership-set* (a list of processes). Each process has a local *view* of the configuration. All processes use the initial seed-list as a bootstrap configuration C_0. Every configuration change decision triggers an invocation of the VIEW-CHANGE-CALLBACK at all processes, that informs processes about a new configuration and membership set.

At time t, if C is the configuration view of a majority of its members, we say that C is the *current configuration*. Initially, once a majority of C_0 start, it becomes current.

Failure model We assume that every pair of correct processes can communicate with each other within a known transmission delay bound (an assumption required for failure detection). When this assumption is violated for a pair of (otherwise live) processes, there is no obvious definition to determine which one of them is faulty (though at least one is). We resolve this using the parameters L and K as follows. Every process p (a *subject*) is monitored by K *observer* processes. If L-of-K correct observers cannot communicate with a subject, then the subject is considered *observably unresponsive*. We consider a process faulty if it is either crashed or observably unresponsive.

Cut Detection Guarantees Let C be the current configuration at time t. Consider a subset of processes $F \subset C$ where $\frac{|F|}{|C|} < \frac{1}{2}$. If all processes in $C \setminus F$ remain non-faulty, we guarantee that the multi-process cut will eventually be detected and a view-change $C \setminus F$ installed [1]:

- Multi-process cut detection: With high probability, every process in $C \setminus F$ receives a multi-process cut detection notification about F. In Rapid, the probability is taken over all the random choices of the observer/subject overlay topology, discussed in §4. The property we use is that with high probability the underlying topology remains an expander at all times, where the expansion is quantified in terms of its second eigenvalue.

A similar guarantee holds for joins. If at time t a set J of processes join the system and remain non-faulty, then every process in $C \cup J$ is notified of J joining.

Joins and removals can be combined: If a set of processes F as above fails, and a set of processes J joins, then $(C \setminus F) \cup J$ is eventually notified of the changes.

- View-Change: Any view-change notification in C is by consensus, maintaining *Agreement* on the view-change membership among all correct processes in C; and *Liveness*, provided a majority of $C \cup J$ ($J = \emptyset$ if there are no joiners) remain correct until the VC configuration becomes current.

Our requirements concerning configuration changes hold when the system has quiesced. During periods of instability, intermediate detection(s) may succeed, but there is no formal guarantee about them.

Hand-off Once a new configuration C_{j+1} becomes current, we abstractly abandon C_j and start afresh: New failures can happen within C_{j+1} (for up to half of the membership set), and the Cut Detection and View Change guarantees must hold.

We note that liveness of the consensus layer depends on a majority of both C_j and C_{j+1} remaining correct to perform the 'hand-off': Between the time when C_j becomes current and until C_{j+1} does, no more than a minority fail in either configuration. This dynamic model

[1] The size of cuts $|F|$ we can detect is a function of the monitoring topology. The proof is summarized in §8, and a full derivation appears in a tech report [51].

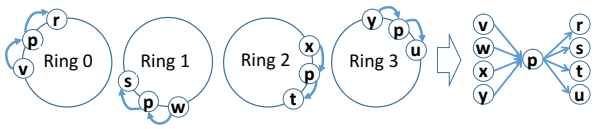

Figure 2: p's neighborhood in a $K = 4$-Ring topology. p's observers are $\{v,w,x,y\}$; p's subjects are $\{r,s,t,u\}$.

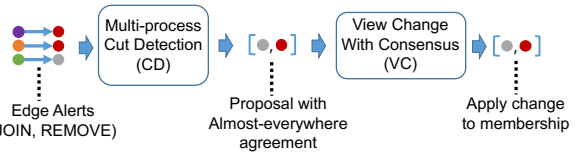

Figure 3: Solution overview, showing the sequence of steps at each process for a single configuration change.

borrows the dynamic-interplay framework of [35, 66].

4 Decentralized Design

Rapid forms an immutable sequence of configurations driven through consensus decisions. Each configuration may drive a single configuration-change decision; the next configuration is logically a new system as in the virtual synchrony approach [22]. Here, we describe the algorithm for changing a known current configuration $\mathscr{C}$, consisting of a membership set (a list of process identities). When clear from the context, we omit $\mathscr{C}$ or explicit mentions of the configuration, as they are fixed within one instance of the configuration-change algorithm.

We start with a brief overview of the algorithm, breaking it down to three components: (1) a monitoring overlay; (2) an almost-everywhere multi-process cut detection (CD); and (3) a fast, leaderless view-change (VC) consensus protocol. The response to problems in Rapid evolves through these three components (see Figure 3).

Monitoring We organize processes into a monitoring topology such that every process monitors K peers and is monitored by K peers. A process being monitored is referred to as a *subject* and a process that is monitoring a subject is an *observer* (each process therefore has K subjects and K observers). The particular topology we employ in Rapid is an *expander graph* [38] realized using K pseudo-random rings [34]. Other observer/subject arrangements may be plugged into our framework without changing the rest of the logic.

Importantly, this topology is deterministic over the membership set $\mathscr{C}$; every process that receives a notification about a new configuration locally determines its subjects and creates the required monitoring channels.

There are two types of alerts generated by the monitoring component, REMOVE and JOIN. A REMOVE alert is broadcast by an observer when there are reachability problems to its subject. A JOIN alert is broadcast by an observer when it is informed about a subject joiner request. In this way, both types of alerts are generated by multiple sources about the same subject. Any best-effort broadcast primitive may be used to disseminate alerts (we use gossip-based broadcast).

Multi-process cut detection (CD) REMOVE and JOIN alerts are handled at each process independently by a multi-process cut detection (CD) mechanism. This mechanism collects evidence to support a single, stable multi-process configuration change proposal. It outputs the same cut proposal *almost-everywhere*; i.e., unanimity in the detection among a large fraction of processes.

The CD scheme with a K-degree monitoring topology has a constant per-process per-round communication cost, and provides stable multi-process cut detection with almost-everywhere agreement.

View change (VC) Finally, we use a consensus protocol that has a fast path to agreement on a view-change. If the protocol collects identical CD proposals from a *Fast Paxos quorum* (three quarters) of the membership, then it can decide in one step. Otherwise, it falls back to Paxos to form agreement on some proposal as a view-change.

We note that other consensus solutions could use CD as input and provide view-change consistency. VC has the benefit of a fast path to decision, taking advantage of the identical inputs almost-everywhere.

We now present a detailed description of the system.

4.1 Expander-based Monitoring

Rapid organizes processes into a monitoring topology that is an *expander graph* [38]. Specifically, we use the fact that a random K-regular graph is very likely to be a good expander for $K \geq 3$ [34]. We construct K pseudo-randomly generated rings with each ring containing the full list of members. A pair of processes (o, s) form an observer/subject *edge* if o precedes s in a ring. Duplicate edges are allowed and will have a marginal effect on the behavior. Figure 2 depicts the neighborhood of a single process p in a 4-Ring topology.

Topology properties. Our monitoring topology has three key properties. The first is *expansion*: the number of edges connecting two sets of processes reflects the relative sizes of the set. This means that if a small subset F of processes V are faulty, we should see roughly $\frac{|V|-|F|}{|V|}$ fraction of monitoring edges to F emanating from the set $V \setminus F$ of healthy processes. This ensures with high probability that healthy processes detect failures, as long as the set of failures is not too large. The size of failures we can detect depends on the expansion of the topology as quantified by the value of its second eigenvalue (§ 8). Second, every process monitors K subjects, and is monitored by K observers. Hence, monitoring incurs $O(K)$ overhead

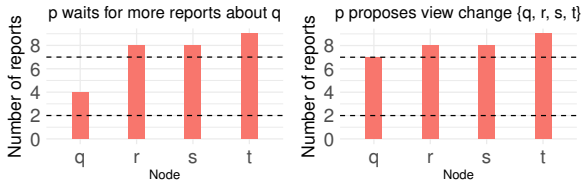

Figure 4: Almost everywhere agreement protocol example at a process p, with tallies about q,r,s,t and $K = 10, H = 7, L = 2$. K is the number of observers per subject. The region between H and L is the unstable region. The region between K and H is the stable region. **Left:** $stable = \{r,s,t\}$; $unstable = \{q\}$. **Right:** q moves from $unstable$ to $stable$; p proposes a view change $\{q,r,s,t\}$.

per process per round, distributing the load across the entire cluster. The fixed observer/subject approach distinguishes Rapid from gossip-based techniques, supporting prolonged monitoring without sacrificing failure detection scalability. At the same time, we compare well with the overhead of gossip-based solutions (§7). Third, every process join or removal results only in $2 \cdot K$ monitoring edges being added or removed.

Joins New processes join by contacting a list of K temporary observers obtained from a seed process (deterministically assigned for each joiner and $\mathscr{C}$ pair, until a configuration change reflects the join). The temporary observers generate independent alerts about joiners. In this way, multiple JOIN alerts are generated from distinct sources, in a similar manner to alerts about failures.

Pluggable edge-monitor. A monitoring edge between an observer and its subject is a pluggable component in Rapid. With this design, Rapid can take advantage of diverse failure detection and monitoring techniques, e.g., history-based adaptive techniques as used by popular frameworks like Hystrix [57] and Finagle [68]; phi-accrual failure detectors [31]; eliciting indirect probes [29]; flooding a suspicion and allowing a timeout period for self-rebuttal [50]; using cross-layer information [54]; application-specific health checks; and others.

Irrevocable Alerts When the edge-monitor of an observer indicates an existing subject is non-responsive, the observer broadcasts a REMOVE alert about the subject. Given the high fidelity made possible with our stable edge monitoring, these alerts are considered *irrevocable*, thus Rapid prevents spreading conflicting reports. When contacted by a subject, a temporary observer broadcasts JOIN alert about the subject.

4.2 Multi-process Cut Detection

Alerts in Rapid may arrive at different orders at each process. Every process independently aggregates these alerts until a stable multi-process cut is detected. Our approach aims to reach agreement *almost everywhere* with regards to this detection. Our mechanism is based on a simple key insight: A process defers a decision on a single process until the alert-count it received on all processes is considered stable. In particular, it waits until there is no process with an alert-count above a low-watermark threshold L and below a high-watermark threshold H.

Our technique is simple to implement; it only requires maintaining integer counters per-process and comparing them against two thresholds. This state is reset after each configuration change.

Processing REMOVE and JOIN alerts Every process ingests broadcast alerts by observers about edges to their subjects. A REMOVE alert reports that an edge to the subject process is faulty; a JOIN alert indicates that an edge to the subject is to be created. By design, a JOIN alert can only be about a process not in the current configuration $\mathscr{C}$, and REMOVE alerts can only be about processes in $\mathscr{C}$. There cannot be JOIN and REMOVE alerts about the same process in $\mathscr{C}$.

Every process p tallies up distinct REMOVE and JOIN alerts in the current configuration view as follows. For each observer/subject pair (o, s), p maintains a value $M(o, s)$ which is set to 1 if an alert was received from observer o regarding subject s; and it is set to (default) 0 if no alert was received. A *tally(s)* for a process s is the sum of entries $M(*, s)$.

Stable and unstable report modes We use two parameters H and L, $1 \leq L \leq H \leq K$. A process p considers a process s to be in a *stable report mode* if $|\text{tally}(s)| \geq H$ at p. A stable report mode indicates that p has received at least H distinct observer alerts about s, hence we consider it "high fidelity"; A process s is in an *unstable report mode* if tally(s) is in between L and H. If there are fewer than L distinct observer alerts about s, we consider it noise. Recall that Rapid does not revert alerts; hence, a stable report mode is permanent once it is reached. Note that, the same thresholds are used for REMOVE and JOIN reports; this is not mandatory, and is done for simplicity.

Aggregation Each process follows one simple rule for aggregating tallies towards a proposed configuration change: *delay proposing a configuration change until there is at least one process in stable report mode and there is no process in unstable report mode*. Once this condition holds, the process announces a configuration change proposal consisting of all processes in stable report mode, and the current configuration identifier. The proposed configuration change has the almost-everywhere agreement property, which we analyze in §8 and evaluate in §7. Figure 4 depicts the almost everywhere agreement mechanism at a single process.

Ensuring liveness: implicit detections and reinforcements There are two cases in which a subject process might get stuck in an unstable report mode and not accrue H observer reports. The first is when the observers themselves are faulty. To prevent waiting for stability forever, for each observer o of s, if both o and s are in the unstable report mode, then an *implicit-alert* is applied from o to s (i.e., an implicit REMOVE if s is in $\mathscr{C}$ and a JOIN otherwise; o is by definition always in $\mathscr{C}$).

The second is the case when a subject process has good connections to some observers, and bad connections to others. In this case, after a subject s has been in the unstable reporting mode for a certain timeout period, each observer o of s *reinforces* the detection: if o did not send a REMOVE message about s already, it broadcasts a REMOVE about s to echo existing REMOVEs.

4.3 View-change Agreement

We use the result of each process' CD proposal as input to a consensus protocol that drives agreement on a single view-change.

The consensus protocol in Rapid has a fast, leaderless path in the common case, that has the same overhead as simple gossip. The fast path is built around the Fast Paxos algorithm [53]. In our variation, we use the CD result as initial input to processes, instead of having an explicit proposer populating the processes with a proposal. Fast Paxos reaches a decision if there is a quorum larger than three quarters of the membership set with an identical proposal. Due to our prudent almost-everywhere CD scheme, with high probability, all processes indeed have an identical multi-process cut proposal. In this case, the VC protocol converges simply by counting the number of identical CD proposals.

The counting protocol itself uses gossip to disseminate and aggregate a bitmap of "votes" for each unique proposal. Each process sets a bit in the bitmap of a proposal to reflect its vote. As soon as a process has a proposal for which three quarters of the cluster has voted, it decides on that proposal.

If there is no fast-quorum support for any proposal because there are conflicting proposals, or a timeout is reached, Fast Paxos falls back to a recovery path, where we use classical Paxos [52] to make progress.

In the face of partitions [36], some applications may need to maintain availability everywhere (AP), and others only allow the majority component to remain live to provide strong consistency (CP). Rapid guarantees to reconfigure processes in the majority component. The remaining processes are forced to logically depart the system. They may wait to rejoin the majority component, or choose to form a separate configuration (which Rapid facilitates quickly). The history of the members forming a new configuration will have an explicit indication of these events, which applications can choose to use in any manner that fits them (including ignoring).

5 Logically Centralized Design

We now discuss how Rapid runs as a logically centralized service, where a set of auxiliary nodes S records the membership changes for a cluster $\mathscr{C}$. This is a similar model to how systems use ZooKeeper to manage membership: the centralized service is the ground truth of the membership list.

Only three minor modifications are required to the protocol discussed in §4:

1. Nodes in the current configuration $\mathscr{C}$ continue monitoring each other according to the k-ring topology (to scale the monitoring load). Instead of gossiping these alerts to all nodes in $\mathscr{C}$, they report it only to all nodes in S instead.
2. Nodes in S apply the CD protocol as before to identify a membership change proposal from the incoming alerts. However, they execute the VC protocol only among themselves.
3. Nodes in $\mathscr{C}$ learn about changes in the membership through notifications from S (or by probing nodes in S periodically).

The resulting solution inherits the stability and agreement properties of the decentralized protocol, but with reduced resiliency guarantees; the resiliency of the overall system is now bound to that of S ($F = \frac{S}{2} - 1$) – as with any logically centralized design. For progress, members of $\mathscr{C}$ need to be connected to a majority partition of S.

6 Implementation

Rapid is implemented in Java with 2362 lines of code (excluding comments and blank lines). This includes all the code associated with the membership protocol as well as messaging and failure detection. In addition, there are 2034 lines of code for tests. Our code is open-sourced under an Apache 2 license [2].

Our implementation uses gRPC and Netty for messaging. The counting step for consensus and the gossip-based dissemination of alerts are performed over UDP. Applications interact with Rapid using the APIs for joining and receiving callbacks described in §3. The logical identifier (§3) for each process is generated by the Rapid library using UUIDs. The join method allows users to supply edge failure detectors to use. Similar to APIs of existing systems such as Serf [44] and Akka Cluster [69], users associate application-specific metadata with the process being initialized (e.g., *"role":"backend"*).

Our default failure detector has observers send probes to their subjects and wait for a timeout. Observers mark an edge faulty when the number of communication exceptions they detect exceed a threshold (40% of the last 10 measurement attempts fail). Similar to Memberlist

Figure 5: Bootstrap convergence measurements showing the time required for all nodes to report a cluster size of N. Rapid bootstraps a 2000 node cluster 2-2.32x faster than Memberlist, and 3.23-5.81x faster than ZooKeeper.

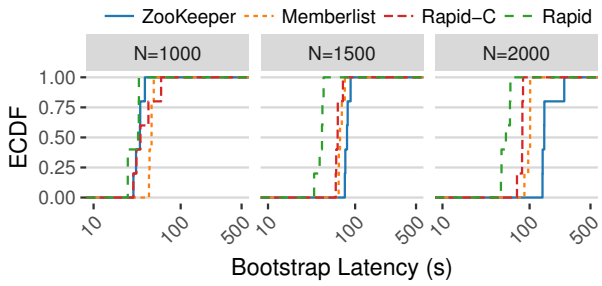

Figure 6: Bootstrap latency distribution for all systems.

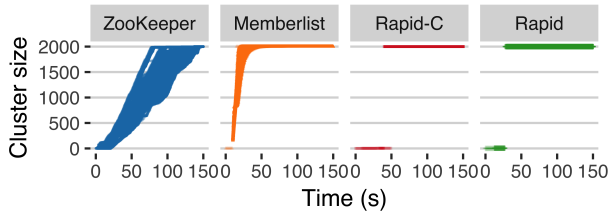

Figure 7: Timeseries showing the first 150 seconds of all three systems bootstrapping a 2000 node cluster.

and Akka Cluster, Rapid batches multiple alerts into a single message before sending them on the wire.

7 Evaluation

Setup We run our experiments on a shared internal cloud service with 200 cores and 800 GB of RAM (100 VMs). We run multiple processes per VM, given that the workloads are not CPU bottlenecked. We vary the number of processes (N) in the cluster from 1000 to 2000.

We compare Rapid against *(i)* ZooKeeper [15] accessed using Apache Curator [13], *(ii)* Memberlist [47], the SWIM implementation used by Serf [44] and Consul [45]. For Rapid, we use the decentralized variant unless specified otherwise (Rapid-C, where a 3-node ensemble manages the membership of N processes).

We also tested Akka Cluster [69] but found its bootstrap process to not stabilize for clusters beyond 500 processes, and therefore do not present further (see §2.1 and Figure 1). All ZooKeeper experiments use a 3-node ensemble, configured according to [16]. For Memberlist, we use the provided configuration for single data center settings (called *DefaultLANConfig*). Rapid uses $\{K,H,L\} = \{10,9,3\}$ for all experiments and we also show a sensitivity analysis. We seek to answer:

- How quickly can Rapid bootstrap a cluster?
- How does Rapid react to different fault scenarios?
- How bandwidth intensive is Rapid?
- How sensitive is the almost-everywhere agreement property to the choice of K,H,L?
- Is Rapid easy to integrate with real applications?

Bootstrap experiments We stress the bootstrap protocols of all three systems under varying cluster sizes. For Memberlist and Rapid, we start each experiment with a single seed process, and after ten seconds, spawn a subsequent group of $N-1$ processes (for ZooKeeper, the 3-node ZooKeeper cluster is brought up first). Every process logs its observed cluster size every second. Every measurement is repeated five times per value of N. We measure the time taken for all processes to converge to a cluster size of N (Figure 5). For $N = 2000$, Rapid improves bootstrap latencies by 2-2.32x over Memberlist, and by 3.23-5.8x over ZooKeeper.

ZooKeeper suffers from herd behavior during the bootstrap process (as documented in [18]), resulting in its bootstrap latency increasing by 4x from when N=1000 to when N=2000. Group membership with ZooKeeper is done using watches. When the i^{th} process joins the system, it triggers $i-1$ watch notifications, causing $i-1$ processes to re-read the full membership list and register a new watch each. In the interval between a watch having triggered and it being replaced, the client is not notified of updates, leading to clients observing different sequences of membership change events [17]. This behavior with watches leads to the eventually consistent client behavior in Figure 7. Lastly, we emphasize that this is a 3-node ZooKeeper cluster being used *exclusively* to manage membership for a single cluster. Adding even one extra watch per client to the group node at N=2000 inflates bootstrap latencies to 400s on average.

Memberlist processes bootstrap by contacting a seed. The seed thereby learns about every join attempt. However, non-seed processes need to periodically execute a push-pull handshake with each other to synchronize their views (by default, once every 30 seconds). Memberlist's convergence times are thereby as high as 95s on average when $N = 2000$ (Figure 7).

Similar to Memberlist, Rapid processes bootstrap by contacting a seed. The seed aggregates alerts until it bootstraps a cluster large enough to support a Paxos quorum (minimum of three processes). The remaining processes are admitted in a subsequent one or more view

System	N=1000	N=1500	N=2000
ZooKeeper	1000	1500	2000
Memberlist	901	1383	1858
Rapid-C	9	10	7
Rapid	4	8	4

Table 1: Number of unique cluster sizes reported by processes in bootstrapping experiments.

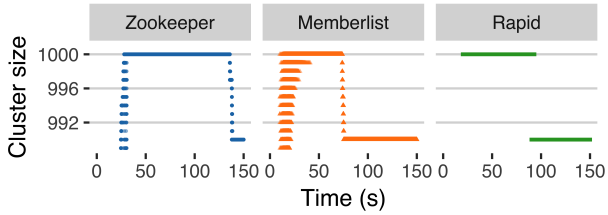

Figure 8: Experiment with 10 concurrent crash failures.

changes. For instance, in Figure 7, Rapid transitions from a single seed to a five node cluster, before forming a cluster of size 2000. We confirm this behavior across runs in Table 1, which shows the number of unique cluster sizes reported for different values of N. In the ZooKeeper and Memberlist experiments, processes report a range of cluster sizes between 1 and N as the cluster bootstraps. Rapid however brings up large clusters with very few intermediate view changes, reporting four and eight unique cluster sizes for each setting. Our logically centralized variant Rapid-C, behaves similarly for the bootstrap process. However, processes in Rapid-C periodically probe the 3-node ensemble for updates to the membership (the probing interval is set to be 5 seconds, the same as with ZooKeeper). This extra step increases bootstrap times over the decentralized variant; in the latter case, all processes participate in the dissemination of votes through aggregate gossip.

Crash faults We now set $N = 1000$ to compare the different systems in the face of crash faults. At this size, we have five processes per-core in the infrastructure, leading to a stable steady state for all three systems. We then fail ten processes and observe the cluster membership size reported by every other process in the system.

Figure 8 shows the cluster size timeseries as recorded by each process. Every dot in the timeseries represents a cluster size recording by a single process. With Memberlist and ZooKeeper, processes record several different cluster sizes when transitioning from N to $N-F$. Rapid on the other hand concurrently detects all ten process failures and removes them from the membership using a 1-step consensus decision. Note, our edge failure detector performs multiple measurements before announcing a fault for stability (§6), thereby reacting roughly 10 seconds later than Memberlist does. The results are identical when the ten processes are partitioned away completely

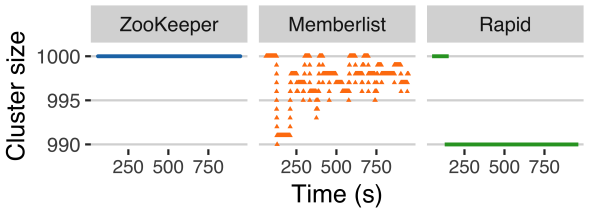

Figure 9: Asymmetric network failure with one-way network partition on the network interface of 1% of processes (ingress path).

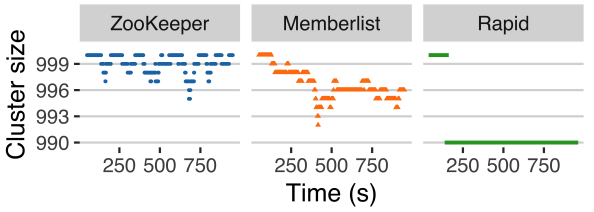

Figure 10: Experiment with 80% ingress packet loss on the network interface of 1% of processes.

from the cluster (we do not show the plots for brevity).

Asymmetric network failures We study how each system responds to common network failures that we have seen in practice. These scenarios have also been described in [49, 19, 37, 67, 41].

Flip-flops in one-way connectivity. We enforce a "flip-flopping" asymmetric network failure. Here, 10 processes lose all packets that they receive for a 20 second interval, recover for 20 seconds, and then repeat the packet dropping. We enforce this by dropping packets in the `iptables INPUT` chain. The timeseries of cluster sizes reported by each process is shown in Figure 9.

ZooKeeper does not react to the injected failures because clients do not receive packets on the ingress path, but send heartbeats to the ZooKeeper nodes. Reversing the direction of connectivity loss as in the next experiment does cause ZooKeeper to react. Memberlist never removes all the faulty processes from the membership, and oscillates throughout the duration of the failure scenario. We also find several intervals of inconsistent views among processes. Unlike ZooKeeper and Memberlist, Rapid detects and removes the faulty processes.

High packet loss scenario. We now run an experiment where 80% of outgoing packets from the faulty processes are dropped. We inject the fault at $t = 90s$. Figure 10 shows the resulting membership size timeseries. ZooKeeper reacts to the failures at $t = 200s$, and does not remove all faulty processes from the membership. Figure 10 also shows how Memberlist's failure detector is conservative; even a scenario of sustained high packet loss is insufficient for Memberlist to conclusively remove a set of processes from the network. Further-

	KB/s (received / transmitted)		
System	Mean	p99	max
ZooKeeper	0.43 / 0.01	17.52 / 0.33	38.86 / 0.67
Memberlist	0.54 / 0.64	5.61 / 6.40	7.36 / 8.04
Rapid	0.71 / 0.71	3.66 / 3.72	9.56 / 11.37

Table 2: Mean, 99^{th} percentile and maximum network bandwidth utilization per process.

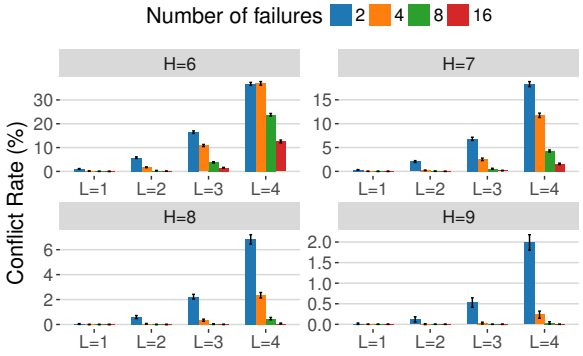

Figure 11: Almost-everywhere agreement conflict probability for different combinations of H, L and failures F when K=10. Note the different y-axis scales.

more, we observe view inconsistencies with Memberlist near $t = 400s$. Rapid, again, correctly identifies and removes only the faulty processes.

Memory utilization. Memberlist (written in Go) used an average of 12MB of resident memory per process. With Rapid and ZooKeeper agents (both Java based), GC events traced using `-XX:+PrintGC` report min/max heap utilization of 10/25MB and 3.5/16MB per process.

Network utilization. Table 2 shows the mean, 99th and 100th percentiles of network utilization per second across processes during the crash fault experiment (1000 processes). Rapid has a peak utilization of 9.56 KB/s received (and 11.37 KB/s transmitted) versus 7.36 KB/s received (8.04 KB/s transmitted) for Memberlist. Rapid therefore provides stronger guarantees than Memberlist for a similar degree of network bandwidth utilization. ZooKeeper clients have a peak ingress utilization of 38.86 KB/s per-process on average to learn the updated view of the membership.

K, H, L sensitivity study We now present the effect of K, H and L on the almost-everywhere agreement property of our multi-process detection technique. We initialize 1000 processes and select F random processes to fail. We generate alert messages from the F processes' observers and deliver these alerts to each process in a uniform random order. We count the number of processes that announce a membership proposal that did not include all F processes (a *conflict*). We run all parameter combinations for $H = \{6,7,8,9\}, L = \{1,2,3,4\}, F = \{2,4,8,16\}$ with 20 repetitions per combination.

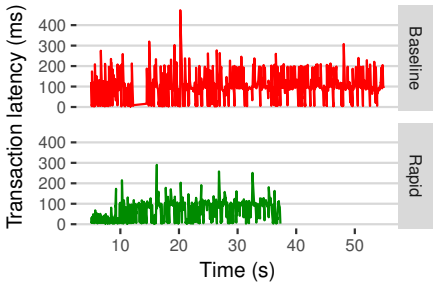

Figure 12: Transaction latency when testing an in-house gossip-style failure detector and Rapid for robustness against a communication fault between two processes. The baseline failure detector triggers repeated failovers that reduce throughput by 32%.

Figure 11 shows the results. As our analysis (§8) predicts, the conflict rate is highest when the gap between H and L is lowest ($H = 6, L = 4$) and the number of failures F is 2. This setting causes processes to arrive at a proposal without waiting long enough. As we increase the gap $H - L$ and increase F, the algorithm at each process waits long enough to gather all the relevant alerts, thereby diminishing the conflict probability. Our system is thereby robust across a range of values; for $H - L = 5$ and $F = 2$, we get a 2% conflict rate for different values of H and L. Increasing to $H - L = 6$ drops the probability of a conflict by a factor of 4.

Experience with end-to-end workloads We integrated Rapid within use cases at our organization that required membership services. Our goal is to understand the ease of integrating and using Rapid.

Distributed transactional data platform. We worked with a team that uses a distributed data platform that supports transactions. We replaced the use of its in-house gossip-based failure detector that uses all-to-all monitoring, with Rapid. The failure detector recommends membership changes to a Paxos-based reconfiguration mechanism, and we let Rapid provide input to the re-configuration management instead. Our integration added 62 and removed 25 lines of code. We also ported the system's failure detection logic such that it could be supplied to Rapid as an edge failure detector, which involved an additional 123 lines of code.

We now describe a use case in the context of this system where stable monitoring is required. For total ordering of requests, the platform has a transaction serialization server, similar to the one used in Google Megastore [20] and Apache Omid [14]. At any moment in time, the system has only one active serialization server, and its failure requires the cluster to identify a new candidate server for a failover. During this interval, workloads are paused and clients do not make progress.

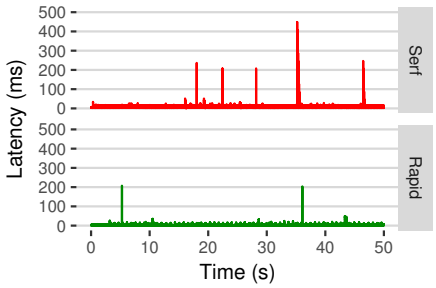

Figure 13: Service discovery experiment. Rapid's batching reduces the number of configuration reloads during a set of failures, thereby reducing tail latency.

We ran an experiment where two update-heavy clients (read-write ratio of 50-50) each submit 500K read/write operations, batched as 500 transactions. We injected a failure that drops all packets between the current serialization server and one other data server (resembling a packet blackhole as observed by [41]). Note, this fault does not affect communication links between clients and data servers. We measured the impact of this fault on the end-to-end latency and throughput.

With the baseline failure detector, the serialization server was repeatedly added and removed from the membership. The repeated failovers caused a degradation of end-to-end latency and a 32% drop in throughput (Figure 12). When using Rapid however, the system continued serving the workload without experiencing any interruption (because no node exceeded L reports).

Service discovery. A common use case for membership is service discovery, where a fleet of servers need to be discovered by dependent services. We worked with a team that uses Terraform [46] to drive deployments where a load balancer discovers a set of backend servers using Serf [44]. We replaced their use of Serf in this workflow with an agent that uses Rapid instead (the Rapid specific code amounted to under 20 lines of code). The setup uses nginx [1] to load balance requests to 50 web servers (also using nginx) that serve a static HTML page. All 51 machines run as t2.micro instances on Amazon EC2. Both membership services update nginx's configuration file with the list of backend servers on every membership change. We then use an HTTP workload generator to generate 1000 requests per-second. 30 seconds into the experiment, we fail ten nodes and observe the impact on the end-to-end latency (Figure 13).

Rapid detects all failures concurrently and triggers a single reconfiguration because of its multi-node membership change detection. Serf, which uses Memberlist, detects multiple independent failures that result in several updates to the nginx configuration file. The load balancer therefore incurs higher latencies when using Serf at multiple intervals (t=35s and t=46s) because nginx is reconfiguring itself. In the steady state where there are no failures, we observe no difference between Rapid and Serf, suggesting that Rapid is well suited for service discovery workloads, despite offering stronger guarantees.

8 Summary of Proofs

In the interest of space, we report the complete proof of correctness in a tech report [51], and only present the key take aways here. Our consensus engine is standard, and borrows from known literature on consensus algorithms [53, 32, 52]. We do not repeat its proof of Agreement and Liveness.

It is left to prove that faced with F failures in a configuration C, the stable failure detector detects and outputs F at all processes with high probability. We divide the proof into two parts.

Detection guarantee For parameters L and K, we can detect a failure of a set F as long as $|F|$ is bounded by the relationship $\frac{|F|}{|C|} \leq (1 - \frac{L}{K} - \frac{\lambda}{2K})$. Here, λ is the second eigenvalue of the underlying monitoring topology, and is tied to the expansion properties of the topology. In our experiments, with $K = 10$, we have observed consistently that $\frac{\lambda}{2K} < 0.45$ (which, for $L = 3$, yields $\frac{|F|}{|C|} \leq 0.25$).

Almost-everywhere agreement Second, we prove the almost-everywhere agreement property about our multi-process cut protocol. We assume that there are t failures, and that nodes receive alerts about these failures in a uniform random order. Let $\Pr[B(z)]$ be the probability that the CD protocol at z outputs a subset of t that differs from the output at other nodes. We show that if multiple processes fail simultaneously, $\Pr[B(z)]$ *exponentially* decreases with increasing K.

9 Conclusions

In this paper, we demonstrated the effectiveness of detecting cluster-wide multi-process cut conditions, as opposed to detecting individual node changes. The high fidelity of the CD output prevents frequent oscillations or incremental changes when faced with multiple failures. It achieves unanimous detection almost-everywhere, enabling a fast, leaderless consensus protocol to drive membership changes. Our implementation successfully bootstraps clusters of 2000 processes 2-5.8x times faster than existing solutions, while being stable against complex network failures. We found Rapid easy to integrate end-to-end within a distributed transactional data platform and a service discovery use case.

References

[1] Nginx. http://nginx.org/en/docs/http/load_balancing.html.

[2] Rapid source code. https://github.com/lalithsuresh/rapid, 2018.

[3] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 739–753.

[4] AMAZON. Summary of the Amazon EC2, Amazon EBS, and Amazon RDS Service Event in the EU West Region. https://aws.amazon.com/message/2329B7/, 2011.

[5] AMAZON. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/, 2011.

[6] AMAZON. Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region. https://aws.amazon.com/message/680587/, 2012.

[7] AMAZON. Summary of Windows Azure Service Disruption on Feb 29th, 2012. https://azure.microsoft.com/en-us/blog/summary-of-windows-azure\-service-disruption-on-feb-29th-2012/, 2012.

[8] APACHE CASSANDRA. Apache Cassandra. http://cassandra.apache.org/, 2007.

[9] APACHE CASSANDRA. Cassandra-3831: scaling to large clusters in GossipStage impossible due to calculatePendingRanges. https://issues.apache.org/jira/browse/CASSANDRA-3831, 2012.

[10] APACHE CASSANDRA. Cassandra-6127: vnodes don't scale to hundreds of nodes. https://issues.apache.org/jira/browse/CASSANDRA-6127, 2013.

[11] APACHE CASSANDRA. Cassandra-9667: strongly consistent membership and ownership. https://issues.apache.org/jira/browse/CASSANDRA-9667, 2015.

[12] APACHE CASSANDRA. Cassandra-11740: Nodes have wrong membership view of the cluster. https://issues.apache.org/jira/browse/CASSANDRA-11740, 2016.

[13] APACHE CURATOR. Apache Curator. https://curator.apache.org/, 2011.

[14] APACHE OMID. Apache Omid. http://omid.incubator.apache.org/, 2011.

[15] APACHE ZOOKEEPER. Apache Zookeeper. https://zookeeper.apache.org/, 2010.

[16] APACHE ZOOKEEPER. ZooKeeper Administrator's Guide. https://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html, 2010.

[17] APACHE ZOOKEEPER. ZooKeeper Programmer's Guide. https://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html, 2010.

[18] APACHE ZOOKEEPER. ZooKeeper Recipes. https://zookeeper.apache.org/doc/trunk/recipes.html, 2010.

[19] BAILIS, P., AND KINGSBURY, K. The network is reliable. *Queue 12*, 7 (July 2014), 20:20–20:32.

[20] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[21] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2013.

[22] BIRMAN, K. Replication. Springer-Verlag, Berlin, Heidelberg, 2010, ch. A History of the Virtual Synchrony Replication Model, pp. 91–120.

[23] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[24] CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. Group communication specifications: A comprehensive study. *ACM Comput. Surv. 33*, 4 (Dec. 2001), 427–469.

[25] CONSUL PROJECT. Issue 1212: Node health flapping - EC2. https://github.com/hashicorp/consul/issues/1212, 2015.

[26] CONSUL PROJECT. Issue 916: Frequent membership loss for 50-100 node cluster in AWS. https://github.com/hashicorp/consul/issues/916, 2015.

[27] CONSUL PROJECT. Issue 916: node health constantly flapping in large cluster. https://github.com/hashicorp/consul/issues/1337, 2015.

[28] COWLING, J., PORTS, D. R. K., LISKOV, B., POPA, R. A., AND GAIKWAD, A. Census: Location-aware membership management for large-scale distributed systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 12–12.

[29] DAS, A., GUPTA, I., AND MOTIVALA, A. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on* (2002), IEEE, pp. 303–312.

[30] DEAN, J., AND BARROSO, L. A. The Tail At Scale. *Communications of the ACM 56* (2013), 74–80.

[31] DEFAGO, X., URBAN, P., HAYASHIBARA, N., AND KATAYAMA, T. The ϕ accrual failure detector. In *RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology* (2004), pp. 66–78.

[32] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (Apr. 1988), 288–323.

[33] ETCD. etcd. https://github.com/coreos/etcd, 2014.

[34] FRIEDMAN, J., KAHN, J., AND SZEMERÉDI, E. On the second eigenvalue of random regular graphs. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1989), STOC '89, ACM, pp. 587–598.

[35] GAFNI, E., AND MALKHI, D. Elastic Configuration Maintenance via a Parsimonious Speculating Snapshot Solution. In *DISC 2015* (Tokyo, France, Oct. 2015), Y. Moses and M. Roy, Eds., vol. LNCS 9363 of *29th International Symposium on Distributed Computing*, Toshimitsu Masuzawa and Koichi Wada, Springer-Verlag Berlin Heidelberg.

[36] GILBERT, S., AND LYNCH, N. Perspectives on the cap theorem. *Computer 45*, 2 (Feb. 2012), 30–36.

[37] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 350–361.

[38] GOLDREICH, O. Studies in complexity and cryptography. Springer-Verlag, Berlin, Heidelberg, 2011, ch. Basic Facts About Expander Graphs, pp. 451–464.

[39] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 7:1–7:14.

[40] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 1–16.

[41] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 139–152.

[42] GUO, Z., MCDIRMID, S., YANG, M., ZHUANG, L., ZHANG, P., LUO, Y., BERGAN, T., MUSUVATHI, M., ZHANG, Z., AND ZHOU, L. Failure recovery: When the cure is worse than the disease. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems* (Berkeley, CA, 2013), USENIX.

[43] HADOOP, A. http://hadoop.apache.org/.

[44] HASHICORP. Serf. https://www.serf.io/, 2013.

[45] HASHICORP. Consul. https://www.consul.io/, 2014.

[46] HASHICORP. Terraform. https://www.terraform.io/, 2014.

[47] HASHICORP. memberlist. https://github.com/hashicorp/memberlist, 2015.

[48] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS '17, ACM, pp. 150–155.

[49] JEFF DEAN. Designs, Lessons and Advice from Building Large Distributed Systems. http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf, 2009.

[50] JOHANSEN, H. D., RENESSE, R. V., VIGFUSSON, Y., AND JOHANSEN, D. Fireflies: A secure and scalable membership and gossip service. *ACM Trans. Comput. Syst. 33*, 2 (May 2015), 5:1–5:32.

[51] L. SURESH, D. MALKHI, P. GOPALAN, I. PORTO CARREIRO, Z. LOKHANDWALA. Consistent and Stable Membership at Scale with Rapid. https://github.com/lalithsuresh/rapid/blob/master/docs/tech-report.pdf, 2018.

[52] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[53] LAMPORT, L. Fast paxos. *Distributed Computing 19* (October 2006), 79–103.

[54] LENERS, J. B., WU, H., HUNG, W.-L., AGUILERA, M. K., AND WALFISH, M. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *SOSP* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 279–294.

[55] MCCAFFREY, C. Building Scalable Stateful Services. https://speakerdeck.com/caitiem20/building-scalable-stateful-services, 2015.

[56] NETFLIX. Dynomite. https://github.com/Netflix/dynomite, 2011.

[57] NETFLIX. Introducing Hystrix for Resilience Engineering. http://goo.gl/h9brP0, 2012.

[58] NETFLIX. Eureka. https://github.com/Netflix/eureka, 2014.

[59] NEWELL, A., KLIOT, G., MENACHE, I., GOPALAN, A., AKIYAMA, S., AND SILBERSTEIN, M. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (April 2016), ACM - Association for Computing Machinery.

[60] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (1988), ACM, pp. 8–17.

[61] PETER KELLEY. Eureka! Why You Shouldn't Use ZooKeeper for Service Discovery. https://tech.knewton.com/blog/2014/12/eureka-shouldnt-use\-zookeeper-service-discovery/, 2014.

[62] REDIS. Redis. http://redis.io/, 2009.

[63] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[64] SCYLLA. ScyllaDB. http://www.scylladb.com/, 2013.

[65] SHRAER, A., REED, B., MALKHI, D., AND JUNQUEIRA, F. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 39–39.

[66] SPIEGELMAN, A., KEIDAR, I., AND MALKHI, D. Dynamic reconfiguration: Abstraction and optimal asynchronous solution.

[67] TURNER, D., LEVCHENKO, K., MOGUL, J. C., SAVAGE, S., AND SNOEREN, A. C. On failure in managed enterprise networks. *HP Labs HPL-2012-101* (2012).

[68] TWITTER. Finagle: A Protocol-Agnostic RPC System. https://goo.gl/ITebZs, 2011.

[69] TYPESAFE. Akka. http://akka.io/, 2009.

[70] UBER. Ringpop. https://ringpop.readthedocs.io/en/latest/index.html, 2015.

[71] VAN RENESSE, R., BIRMAN, K. P., AND VOGELS, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst. 21*, 2 (May 2003), 164–206.

[72] VAN RENESSE, R., MINSKY, Y., AND HAYDEN, M. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (London, UK, UK, 1998), Middleware '98, Springer-Verlag, pp. 55–70.