Local Health Awareness for More Accurate Failure Detection

Armon Dadgar James Phillips Jon Currey
{armon,james,jc}@hashicorp.com

***Abstract*—SWIM is a peer-to-peer group membership protocol, with attractive scaling and robustness properties. However, our experience supporting an implementation of SWIM shows that a high rate of false positive failure detections (healthy members being marked as failed) is possible in certain real world scenarios, and that this is due to SWIM's sensitivity to slow message processing. To address this we propose a set of extensions to SWIM (together called Lifeguard), which employ heuristic measures of a failure detector's *local health*. In controlled tests, Lifeguard is able to reduce the false positive rate by more than 50x. Real world deployment of the extensions has significantly reduced support requests and observed instability. The need for this work points to the fail-stop failure model being overly simplistic for large datacenters, where the likelihood of some nodes experiencing transient CPU starvation, IO flakiness, random packet loss, and other non-crash problems becomes high. With increasing attention being given to these *gray failures*, we believe the local health abstraction may be applicable in a broad range of settings, including other kinds of distributed failure detectors.**

I. Introduction

Group membership is an intuitive abstraction that can be used to address discovery, failure detection, and load balancing of components of a distributed system. SWIM [1] is a group membership protocol with a simple peer-to-peer design. Its use of randomized communication make it highly scalable, robust to both node and network failures, and easy to manage.

We are aware of three mature open source implementations of SWIM. Butterfly [2] is part of Habitat [3], a popular software automation platform. Ringpop [4] was built to support the applications of a global transportation technology company. memberlist[5] is our implementation of SWIM, which underpins Consul[6], a popular service discovery and management tool, and Nomad[7], a high-availability, data center scale scheduler. Through our relationship with customers, we know of hundreds of thousands of running instances of Consul, and deployments with more than 6,000 members in a single group.

The SWIM paper identifies sensitivity to slow message processing as an issue with the basic SWIM protocol. Slow message processing can be caused by a wide variety of factors, including CPU contention, network delay or loss, and can lead SWIM to declare healthy members as faulty - so called *false positive failure detection*. To counter this, the SWIM paper proposes a Suspicion subprotocol, that trades increased failure detection latency for fewer false positives.

However, our experience supporting Consul and Nomad shows that, even with the Suspicion subprotocol, slow message processing can still lead healthy members being marked as failed in certain circumstances. When the slow processing occurs intermittently, a healthy member can oscillate repeatedly between being marked as failed and healthy. This 'flapping' can be very costly if it induces repeated failover operations, such as provisioning members or re-balancing data.

Debugging these scenarios led us to insights regarding both a deficiency in SWIM's handling of slow message processing, and a way to address that deficiency. The approach used is to make each instance of SWIM's failure detector consider its own health, which we refer to as *local health*. We implement this via a set of extensions to SWIM, which we call Lifeguard. Lifeguard is able to significatly reduce the false positive rate, in both controlled and real-world scenarios.

Looking at the bigger picture, we see that SWIM follows the majority of the literature of distributed failure detectors, in assuming the fail-stop failure model, where processes can only be completely healthy or crash-stopped. We note the increasing attention being given to non-fail stop failures (so called *gray failures* [8]), and that the failure detector slowness that Lifeguard defends against is a kind of gray failure. We believe the local health abstraction may be applicable in a broad range of settings, including other kinds of distributed failure detectors.

SWIM's failure detector is described in Section II. Our insights and Lifeguard are described in Section III. The evaluation of Lifeguard is discussed in Section IV. Related work is discussed in Section V. In Section VI we conclude and consider future work.

II. The SWIM Failure Detector

The SWIM protocol is comprised of a failure detector and a dissemination component. The protocol is peer-to-peer, with every group member running its own instance of the failure detector, and participating in dissemination. The SWIM paper first defines a basic version of the protocol, where these two components are independent.

The basic failure detector operates as follows. Once per round of activity (defined by a configurable *protocol period*), each member **A** selects another member **B** at random, and sends it a `ping` message (in a so called *direct probe*). If an `ack` message is not received back from **B** in a configurable amount of time, **A** selects a configurable number of other members (**C**,**D**,...) and sends each of them a `ping-req` message, to request that they ping **B** (in a so called *indirect probe*). If any of those members recives an `ack` from **B**, it forwards it to **A**. If **A** receives no `ack` messages regarding **B** in a configurable time interval, it declares **B** failed, via the dissemination component.

The dissemination component is used to propagate member status updates: `alive` messages are sent for a member joining the group, `confirm` messages when one is confirmed failed. The communication style is randomized epidemic gossip [9]. When a member originates or receives a status update message, it forwards the message to a set of randomly selected peers. The number of peers chosen is scaled logarithmically with the size of the known group. The messages are piggy-backed on the failure detector messages to reduce message load and dissemination latency. See Section 4.1 of [1] for details of the implementation and its convergence properties.

However, the SWIM authors themselves observe many false positive failure detections with this basic protocol, citing *slow processing* of messages as one of the primary causes. This is intuitive: Failure to deliver or process certain combinations of the messages from in Figure 1 in a timely manner could cause member A to declare B as failed when it is not.

To address this, the SWIM paper introduces the *Suspicion* subprotocol. The Suspicion-enabled failure detector operates in the same way as the basic failure detector up to the point where **A** fails to receive any `ack` messages about **B**. At that point, rather than then immediately declaring **B** as failed, **A** propagates a new `suspect` message. If **B** receives the `suspect` message, it responds with an `alive` message, which acts as a refutation of its failure.

Crucially, both the `suspect` and `alive` messages of the Suspicion subprotocol are sent via the dissemination component. The dissemination component's epidemic infection style makes it resilient to the presence of slow members, which just fail to participate in the dissemination in a timely manner. Hence the Suspicion subprotocol will work around those members, and deliver the messages to healthy members in a timely manner, up to quite a large proportion of the group being slow members.

III. Lifeguard Insights And Design

The Suspicion subprotocol is described as an extension, but in practice it is a necessary part of SWIM, included in every implementation we have seen. However, our experience supporting Consul and Nomad shows that, even with Suspicion, there are situations where periods of slow message processing can lead to a high rate of false positive failure reports. Users reported this issue in diverse scenarios, including: overloaded web servers; edge servers experiencing a DDoS attack; oversubscribed video transcode servers; burstable Performance Instances (*e.g.* AWS T2 instances), with exhausted CPU credits. It was seen on bare metal and virtualized systems, in private data centers and public clouds.

Debugging these scenarios led to the key insight that *the Suspicion subprotocol is also sensitive to slow message processing:* If the suspected member does not process the `suspect` message in a timely manner, or *just one* of the other members that received the `suspect` message does not process the refuting `alive` message in a timely manner, the suspected member will be falsely reported as failed.

While investigating possible solutions, we observed that: i) Because the problematic members are experiencing slow message processing, sending them more messages will not help. ii) However, *comparing the messages it has recently processed against those expected*[1] gives a member an indication that it may be experiencing slow message processing. We think of this as an indication of the *local health* of that member's failure detector (and the member in general). iii) An episode of slow message processing at a member is likely to impact multiple of its interactions with other members in a short period of time.

From these insights we designed a set of extensions to SWIM to make it more resilient to slow processing, that collectively we call Lifeguard. Our approach is to turn SWIM into an adaptive protocol, where the fixed (but configurable) protocol timeouts are replaced with ones adjusted dynamically at each member. The adjustment is driven by heuristic measures of the *local health* of that member. Multiple types of interactions update a saturating counter which estimates the changing health of the local detector over time. As well as being about the health of the local member, the metrics are calculated using only locally available indicators, including comparing the state of the member's interaction with its peers to the expected state.

Lifeguard has the following components:

- **Local Health Aware Probe** which makes SWIM's *protocol period* and *probe timeout* adaptive.
- **Local Health Aware Suspicion** which makes SWIM's *suspicion timeout* adaptive.
- **Buddy System** which prioritizes delivery of `suspect` messages to suspected members.

These components are described in detail in the Lifeguard whitepaper[10]. Here we note only aspects that are discussed further in this paper. Local Health Aware Suspicion uses a heuristic which starts the Suspicion timeout at a high value, and lowers it each time a `suspect` message is processed that indicates an *independent suspicion* of the same suspected member by some other member. The maximum that the Suspicion timeout starts at, the minimum that it drops to, and the number independent suspicions required to make it drop to the minimum (K) are configurable parameters of Lifeguard. Requiring K independent suspicions also reduces sensitivity to concurrent slow processing by other members, since the probability of multiple slow members falsely suspecting the same member reduces exponentially as K increases.

IV. Experimental Evaluation

We evaluate Lifeguard according to the criteria used in [1] to evaluate SWIM:

- **Failure Detection False Positives** The number of healthy members mistakenly marked as failed.

[1]For example, being suspected of failure indicates that a local failure detector may be degraded, but successful completion of a peer probe indicates timely process.

- **Detection and Dissemination Latency** The time to first detection/full dissemination of true failures.
- **Message Load** The number of messages and bytes sent.

We evaluate Lifeguard both in a controlled environment, where the duration and frequency of delays in protocol message processing (each of which we refer to as an *anomaly*) can be varied in a fine-grained manner, and in a public cloud environment where CPU oversubscription can be varied in a way that approximates many of the scenarios reported by users. The three Lifeguard components outlined in Section III are evaluated individually (referred to in this section and figures as `L1`, `L2`, and `L3`, respectively), and together (referred to as `L123`). The pre-Lifeguard performance of Consul (referred to as `L0`) is used as a baseline.

The effect that Lifeguard has on the level of false positive failure detections and on the detection and dissemination latencies for true failures are connected. *It is possible to reduce both the false positive failure detection rate and the detection and disemmination latencies simultaneously, and to tune the balance of the reduction.* The tuning is related to the minimum and maximum Suspicion timeouts, described in Section III. In the most conservative case, where the minimum Suspicion timeout matches the non-Lifeguard fixed level, false positives are reduced by 98% (more than 50x) compared to `L0`, while median detection and dissemination latencies increases less than 0.1%, and 99th and 99.9th percentile latencies are still reasonable, at around 6-9% above `L0`. However, if the minimum suspicion timeout is set to half of the L0 baseline, median detection and dissemination latency are reduced by around 45% while the false positive rate is reduced by 68% compared to the L0 level.

Figure 1 shows how the total number of false positive events (occurring at any member, including ones experiencing anomalies) varies with the number of concurrent anomalies. The number of false positives increases signifincantly with the number of concurrent anomalies, but at every concurrency level, full Lifeguard (`L123`) reduces the false positives by a factor of between 50x and 100x compared to `L0`. Figure 2 shows the number of false positives at healthy members, from the same experiments as Figure 1. Once again, full Lifeguard (`123`) reduces the number of false positives by a factor of between 10x and 100x, at every level of concurrent anomalies. At some levels, zero false positives occurred at healthy nodes during repeated testing. In both cases, Local Health Aware Suspicion (`L2`) has the greatest effect, followed by Local Health Aware Probe (`L1`), but their complementary nature means the best reduction is seen with full Lifeguard (`L123`).

Lifeguard leads to an average increase of around 11% in the number of discrete messages sent. The average amount of data sent actually decreases by around 2%, because the reduction in failure detections leads to less state updates. Note that this is across the full range of anomalous scenarios tested, which include up to 25% of the members experiencing anomalies in a sustained manner. In practice, with less than 10% of members experiencing anomalies, the increase in both messages and bytes sent is around 5%.

The Lifeguard whitepaper[10] contains full details of the experimental method and results.

V. Related Work

Chandra and Toueg [11] introduce unreliable failure detectors, and the *local failure detector module* pattern, which SWIM's per-member failure detector conforms to. However, they use these unreliable failure detectors as is and do not explore making them more reliable. Chen *et al.* [12] propose an adaptive failure detector that, like Lifeguard, adjusts its timeouts based on recent message loss and delay. A line of work (detailed in [10]) refines the approach, but in all cases the focus is distinguishing the behavior of the network from that of the remote system being monitored. The health of the local failure detector itself is not discussed.

Gupta *et al.* [13] introduce adaptivity into the gossip literature. Like Lifeguard, they leverage local knowledge about peer failure and message loss, and uses it to take remedial action (in their case to transition to a different dissemination sub-protocol). However, there is no consideration of slowness, either of message delivery or members themselves.

Johansen *et al.* [14] propose an adaptive gossip-based protocol that has many similarities to SWIM and Lifeguard. Like SWIM, it uses probe-based failure detection, has a suspicion phase and a gossip-based dissemination subprotocol. Like Lifeguard, its probe timeouts are adaptive. But once again, the tuning does not consider the possibility of slow message processing by the failure detector. In fact, the assumption that all correct (non-Byzantine and non-failed) members can run their probe and update dissemination subprotocols in a timely manner is explicitly stated (in section 4.1). The adaptive tuning is present only to accommodate unreliable message delivery.

Dzung *et al.* [15] revisit classical distributed failure detector results with a probabalistic and temporal treatment. But their system model explicitly assumes negligible processing delay, as well as an upper bound on propagation delay.

The prevalence of the fail-stop failure model, where a component is either correct or stopped, explains why none of the above work considers the health of the local failure detector module. Recently, Huang *et al.* [8] and Mogul *et al.* [16] have reported the experience of two of the largest cloud providers around large scale system availability. Both call for a more nuanced view of availability, beyond the fail-stop model, and point to availability's relationship with performance. Both also indicate that while Byzantine Fault Tolerance considers non-crash faults, its complexity and overhead stop it being deployed widely in production systems. This leaves the need for new approaches that consider more nuanced failures without the complexity of arbitrary or adversarial failures.

Our experience supporting SWIM at scale aligns exactly with these reports. [8] focuses on the handling of *gray failures*, examples of which include: severe performance degradation, random packet loss, flaky I/O, memory thrashing, capacity pressure, and non-fatal exceptions. Lifeguard can be viewed as

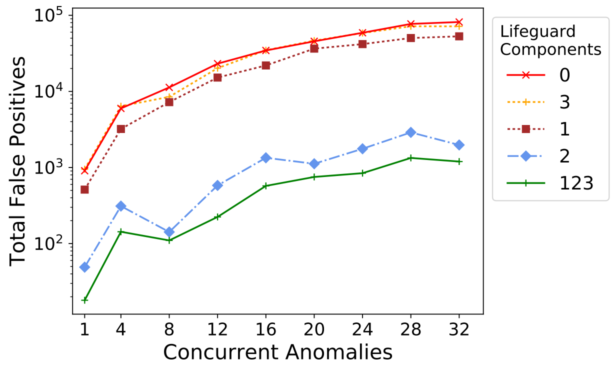

Fig. 1. **Total false positives (at healthy and anomalous members) versus number of concurrent anomalies.**

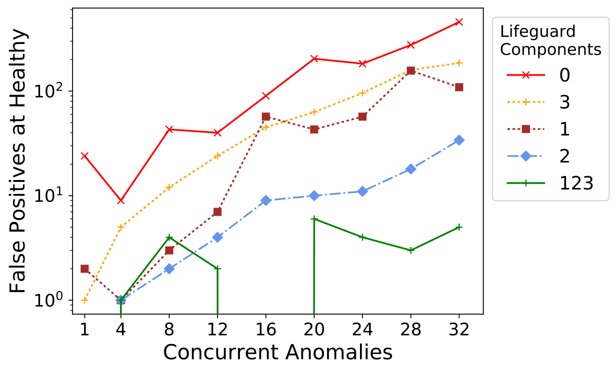

Fig. 2. **False positives at healthy members versus number of concurrent anomalies.**

addressing SWIM's sensitivity to gray failures, and Lifeguard's approach aligns with [8]'s intuitions about how to proceed. Slowing the timeouts of failure detectors with poor local health is a way of closing the observation gap (between local failure detector module instances, as well as between the distributed failure detector and applications). The local health heuristics harness temporal patterns. However, [8] and [16] both still focus on the identification and isolation of slow system components by other, healthy ones. Local health attempts to let slow components detect their own slowness, so they can take remedial action of their own. Local health is therefore potentially an additional, complementary technique that can be used in concert with other approaches.

VI. Conclusions and Future Work

Our goal with Lifeguard was to reduce the rate of false positive failures compared to that of SWIM, without a large increase in detection latencies or message load. We are able to achieve this, and even allow the false positive rate and detection latencies to be reduced simultaneously.

Lifeguard was developed as a modification to an existing SWIM implementation, which was already widely deployed and depended upon. Therefore we took a conservative design approach, with a one-to-one correspondence between SWIM failure detector stages and Lifeguard components. Now having confidence in Lifeguard's benefit and stability, future work can consider replacing its heuristics with either a localized model (used independently by each member) or new heuristics derived from a localized model. The localized model could potentially be derived from a global model, as in [17], or from a utility function, as in [18].

The local health abstraction may be applicable more broadly, as part of the effort to formalize the handling of gray failures. First steps could include investigating the use of measures of local health to adjust timeouts for heartbeat-based failure detectors, or to adjust the number of observers and their level of redundancy in structured (such as ring or hierachical) failure detectors. It could also be deployed directly in non-failure detector subsystems that are capable of slow processing, to help them degrade gracefully.

References

[1] 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*, ser. DSN '02.

[2] (2017, Nov.) Butterfly documentation. Chef. [Online]. Available: https://www.habitat.sh/docs/internals/#supervisor-internals

[3] (2017, Nov.) Habitat. Chef. [Online]. Available: https://www.habitat.sh

[4] (2017, Nov.) Ringpop documentation. Uber Technologies Inc. [Online]. Available: https://ringpop.readthedocs.io/en/latest/architecture_design.html

[5] HashiCorp, "memberlist Project," https://github.com/hashicorp/memberlist, 2017, [Online; accessed 28-Feb-2018].

[6] ——, "Consul Project," https://www.consul.io/, 2017, [Online; accessed 28-Feb-2018].

[7] ——, "Nomad Project," https://www.nomadproject.io/, 2017, [Online; accessed 28-Feb-2018].

[8] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray failure: The achilles' heel of cloud-scale systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17.

[9] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[10] A. Dadgar, J. Phillips, and J. Currey, "Lifeguard : Swim-ing with local health awareness," vol. abs/1707.00788, 2017. [Online]. Available: http://arxiv.org/abs/1707.00788

[11] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*.

[12] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, 2000.

[13] I. Gupta, A. M. Kermarrec, and A. J. Ganesh, "Efficient epidemic-style protocols for reliable and scalable multicast," in *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, 2002.

[14] H. Johansen, A. Allavena, and R. van Renesse, "Fireflies: Scalable support for intrusion-tolerant network overlays," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06, 2006.

[15] D. Dzung, R. Guerraoui, D. Kozhaya, and Y. A. Pignolet, "Never say never – probabilistic and temporal failure detectors," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[16] J. C. Mogul, R. Isaacs, and B. Welch, "Thinking about availability in large service infrastructures," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17.

[17] J. A. Patel, I. Gupta, and N. Contractor, "Jetstream: Achieving predictable gossip dissemination by leveraging social network principles," in *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)*, 2006.

[18] G. He, R. Zheng, I. Gupta, and L. Sha, "A framework for time indexing in sensor networks," *ACM Trans. Sen. Netw.*, 2005.