Medley: A Novel Distributed Failure Detector for IoT Networks

Rui Yang, Shichu Zhu, Yifei Li, and Indranil Gupta
{ry2,szhu28,yifeili3,indy}@illinois.edu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

Abstract

Efficient and correct operation of an IoT network requires the presence of a failure detector and membership protocol amongst the IoT nodes. This paper presents a new failure detector for IoT settings where nodes are connected via a wireless ad-hoc network. This failure detector, which we name *Medley*, is fully decentralized, allows IoT nodes to maintain a local membership list of other alive nodes, detects failures quickly (and updates the membership list), and incurs low communication overhead in the underlying ad-hoc network. In order to minimize detection time and communication, we adapt a failure detector originally proposed for datacenters (SWIM), for the IoT environment. In Medley each node picks a medley of ping targets in a randomized and skewed manner, preferring nearer nodes. Via analysis and NS-3 simulation we show the right mix of pinging probabilities that simultaneously optimize detection time and communication traffic. We have also implemented Medley for Raspberry Pis, and present deployment results.

CCS Concepts • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

Keywords Failure detection, Internet of Things, Membership

ACM Reference Format:
Rui Yang, Shichu Zhu, Yifei Li, and Indranil Gupta. 2019. Medley: A Novel Distributed Failure Detector for IoT Networks. In *20th International Middleware Conference (Middleware '19), December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361556

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361556

1 Introduction

The IoT market is expected to reach 500 Billion dollars in size by 2022 [20]. For instance, during just the second quarter of 2018, Amazon Echo + Dot sold 3.6 million units, while Google Home + Mini sales were 3.1 million units [29]. IoT deployments in smart buildings, smart homes, smart hospitals, smart forests, battlefield scenarios, etc., are proliferating. While today's deployments in smart homes are typically a few tens of devices, tomorrow's vision, in smart buildings and cities, is for hundreds or thousands of devices communicating with each other.

Such large IoT deployments are in essence *distributed systems* of devices. As such, there is a need to provide familiar abstractions and a similar substrate of distributed group operations as those which exist in internet-based distributed systems like datacenters, peer-to-peer systems, clouds, etc. In other words, a *distributed group communication substrate* is required for IoT settings, atop which management functions and distributed programs can then be built. This is critical in order to build large-scale IoT deployments that are truly autonomous, self-healing, and self-sufficient.

One of the first problems that such a substrate needs to solve is detecting failures (we consider only fail-stop failures in this paper [1]). At large scale, failures are the norm rather than the exception. When a device fails, other affected devices need to know about it and take appropriately corrective action, and in some cases inform the human user. This is a very common way of building internet-based and datacenter-based distributed systems. In the IoT environment, examples of corrective actions after failure include (but are not limited to): backup actions to ensure user needs are met (e.g., maintain sufficient lighting in an area), re-initiating and re-replicating device schedules that were stored on failed devices (e.g., timed schedules), informing the upper management layer, informing the user, etc.

Existing techniques in IoT literature detect failures either centralized or semi-centralized [9, 18, 27, 30]. These typically provide a central clearinghouse where information is maintained about currently-alive nodes. Yet, they require access to a cloud or a cloudlet, but this is not always feasible. For instance, IoT deployments may span remote scenarios (e.g., battlefields, forests, etc.), and in some cases sending data

[1] Malicious/Byzantine failures are outside our scope.

to the cloud may be prohibited by laws (e.g., GDPR [24] or HIPAA [1] laws for data from smart hospitals). Additionally, if the centralized service becomes inaccessible (e.g., to due to failures or message losses), the IoT devices no longer have access to the failure detection and membership service.

In this paper, we present *Medley*, which is the first fully-decentralized membership service for IoT distributed systems running over a wireless ad-hoc network. The Medley membership service maintains at each IoT node, an up-to-date membership list containing a list of currently alive nodes in the system. The membership service's critical goal is to detect device failures (crashes) and update membership lists at non-faulty nodes—this is the responsibility of the *failure detector* component, which is the focus of this paper.

Maintaining full membership lists at devices is scalable and feasible in an IoT setting. Consider a system with 5K devices (sufficient to densely populate a multi-storey office building). Suppose each membership list entry uses 20 B (16 B for IPv6 address + 4 B for sequence number). This entails 100 KB of memory for the membership list. For Raspberry Pis which currently have 512 MB of memory, the membership list would occupy only 0.02% of the memory.

Classical distributed systems literature builds a wide swath of distributed algorithms over a full membership list (at each node). Examples include multicast, coordination, leader election, mutual exclusion, virtual synchrony etc. [5]. Essentially, full membership offers maximal flexibility in designing arbitrary distributed algorithms on top of it. It also helps make analysis tractable. For IoT networks, Medley opens the door for similar algorithms to be built on top of it. For instance, to build a multicast tree, one algorithm could choose only nearby nodes, or alternatively a mix of near and far nodes. Both can be built atop a full membership algorithm.

Failure detector protocols for internet-based distributed systems fall into two categories: heartbeat-based (or lease-based), and ping-based. Heartbeat-based protocols [2, 3, 28] have each node send periodic heartbeats to one or more other monitor nodes; when a node n_i dies, its heartbeats stop, the monitors time out, and detect the node n_i as failed. Ping-based protocols [11, 16] have each node periodically ping randomly-selected target nodes from the system. Analysis in [16] has shown that compared to heartbeat protocols, ping-based protocols are faster at detecting failures and impose less network traffic, and can completely detect failures.

We thus adopt a ping-based approach for our IoT failure detection protocol Medley. The key challenge for Medley is that existing ping-based protocols [11, 14] select ping targets uniformly at random across the system. Randomized selection is attractive due to its fast detection, congestion avoidance and load balancing. Yet in a wireless ad-hoc IoT network, uniform random selection leads to large volumes of network traffic that span major portions of the IoT network.

Medley solves this by proposing a new *spatial* ping-target selection strategy which prefers nearer nodes but also has some probability of pinging farther nodes. Compared to fully randomized pinging, always picking nearby nodes as ping targets localizes and reduces network traffic. But this always-local selection leads to high detection times due to lowered randomness of pinging. It also causes non-detection of failures when multiple simultaneous failures occur (e.g., failures caused by a circuit breaker tripping), because all nearby pingers of a failed node have also failed.

Medley attempts to gain the advantages of both approaches by using a hybrid of the uniform-random and the always-local target selection. It utilizes a mix (medley) of nearer and farther ping targets. The best way to mix these targets is the key question we answer via both mathematical analysis as well as experimental results.

We implemented and evaluated Medley via both simulation and real deployment. Our simulation is in the NS-3 simulator. Our Medley implementation is in Raspberry Pi. Our evaluation shows that compared to the classical random pinging schemes, Medley provides comparable failure detection times, lowers bandwidth by 35% (for a given failure detection time), and gives false positive rates as low as 2% under high 20% packet drop rates.

The contributions of this paper are:

1. A new fully-decentralized failure detector protocol, Medley, for wireless ad-hoc IoT networks.
2. Analysis of the key parameter (exponent) in spatial pinging, in order to optimize detection time as well as communication traffic.
3. An optimization to provide time-bounded detection of failures in Medley.
4. Evaluation of Medley via simulation in NS-3.
5. Implementation of Medley for Rasperry Pi, and associated deployment experiments.

2 Background

System Model We consider the fail-stop model: once a node crashes it executes no further instructions or operations. Fail-recovery models can be seen as a special case (with nodes rejoining under a new id or incarnation number). Byzantine failures [12] are beyond our current scope (but represent an interesting future direction).

The network is asynchronous, and messages may be delayed or dropped. Multiple nodes may fail simultaneously. Nodes are allowed to join and voluntarily leave the system. We use N to denote the number of nodes in the system.

Each node maintains a membership list consisting of entries for all other nodes in the system–our membership protocol's goal is to delete entries for failed/departed nodes soon after their failure departure, and to add entries for joining nodes soon after they join.

Our protocol makes no assumptions about clock synchronization, but our analysis assumes (for tractability) that clock speeds are similar.

Failure Detector Properties Failure detectors have three desirable properties. The two desirable correctness properties are called [7] *Completeness* and *Accuracy*. Completeness requires that every failure is detected by at least one non-faulty node. Accuracy means that no failure detections are about healthy nodes, i.e., there are no false positives. In their seminal paper [7] Chandra and Toueg proved that it is impossible to design a failure detector for asynchronous networks, to satisfy both completeness and accuracy. Due to the need to perform corrective recovery actions after a failure, today's failures navigate this impossibility by always guaranteeing completeness, while attempting to maximize accuracy (i.e., minimize false positive rate).

Besides the above two properties, failure detectors also aim to minimize *detection time*, i.e., time between failure and first non-faulty node discovering this failure. Finally, scalability and load balancing are often goals of failure detectors.

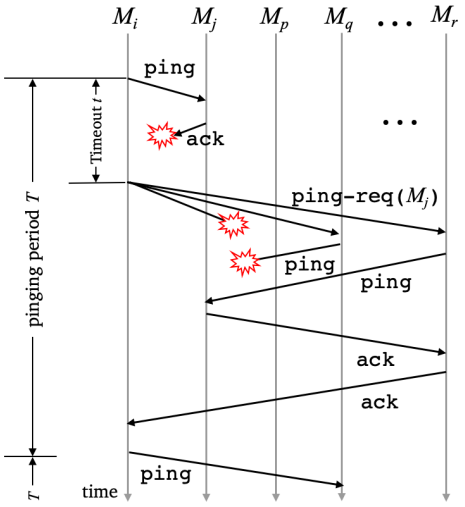


Figure 1. One pinging round in SWIM failure detection (from [16]).

SWIM Failure Detector Our Medley system is adapted from the failure detector and dissemination component of the SWIM protocol [11, 16]. SWIM is popular and various versions of it are today widely deployed in datacenters and in open-source software, including at Uber [19], and HashiCorp's Serf [26] and Consul [10].

We next describe the base SWIM protocol to set the context for Medley. The SWIM membership protocol handles failure detection and dissemination separately. The former detects failures, while the latter multicasts to the system information about node joins, leaves, and detected failures.

Fig. 1 (from [11, 16]) depicts the SWIM failure detector. Each node M_i periodically runs the following protocol every T time units. T is fixed at all nodes but nodes run their periods asynchronously from each other. Each period consists of a *direct pinging* phase and an optional *indirect pinging* phase.

At the start of a period, M_i picks a member from its membership list, uniformly at random, and sends it a ping message. Any node M_j receiving a ping responds immediately with an ack. If M_i receives the ack within a small timeout t (based on message RTT), then M_i is satisfied and does nothing else in this period. Otherwise, M_i picks k other nodes (denoted as indirect pingers), also at random, and asks each of them to ping M_j. If any of these k nodes hears back an ack from M_j, they pass on the ack back to M_i. If M_i receives at least one such ack before the end of the period, it is satisfied and does nothing else in this period. Otherwise, i.e., if M_i hears no acks, then it marks M_j as failed at the end of this period. Pings and acks carry unique identifiers to avoid confusion with other rounds and pingers.

Indirect pinging essentially gives a "second chance" to pinged nodes that might have been congested or slow during the initial ping. It also avoids potential network congestion on the direct $M_i - M_j$ network path. Both of these reduce false positive rates.

Analysis in [16] shows that even without the indirect pinging, failures are detected within $O(1)$ protocol periods on expectation. In addition, the SWIM protocol guarantees eventual detection of all failures (eventual completeness).

SWIM Dissemination Component SWIM nodes continuously piggyback the information about node join/leave/failure atop the messages they send out, namely ping, ack, and indirect ping request for quick dissemination. In addition, a receiving node records new information in the message and reacts accordingly.

This "infection-style" dissemination provides a gossip-like behavior for disseminating information about node failures, joins, and voluntary leaves. Analysis [16] shows that in a system with N nodes, information spreads with high probability to all nodes within $O(log(N))$ time periods.

3 Medley: Design and Analysis

3.1 Spatial Pinging

We target settings where IoT devices are connected via a wireless ad-hoc network. In such scenarios, the SWIM failure detector described in Section 2 is inefficient because it picks ping targets uniformly at random. This spreads pings and acks across far distances in the ad-hoc network. Far pings and acks require more routing hops, incurring higher communication overhead on intermediate nodes, longer latency, and create congestion and packet losses.

Thus, we propose in Medley a way to replace the randomized target selection in SWIM with a *skewed randomized* mechanism which takes distance to target into account. We call this as *spatial target selection.*

Spatial Target Selection: In Medley, a node chooses to ping a given target with probability proportional to $\frac{1}{r^m}$, where r is the distance to the target and m is a fixed exponent.

An example is shown in Fig. 2. M_i has in its membership list nodes M_p, M_q, and M_r at distances $d, 2d$, and $4d$ respectively. In a period of the SWIM protocol at M_i, it has the highest probability ($\propto \frac{1}{d^m}$) of pinging M_p. Similarly, the probabilities for pinging M_q, M_r are respectively $\propto \frac{1}{(2d)^m}$, and $\propto \frac{1}{(4d)^m}$. Using appropriate normalization constants, we depict two points in the space of m. If $m = 1$, then the respective ping probabilities to M_p, M_q, M_r are 0.57, 0.28, and 0.15. However, increasing the exponent m to 2 localizes pings more–the changed ping probabilities are respectively 0.75, 0.2, and 0.05. M_p with probability 0.75 will be pinged even more frequently.

The above calculations indicate that higher values of m localize ping-ack traffic more and incur lower communication overhead. At the same time, more localized pinging reduces the randomness of pinging and thus increases the detection time. We wish to find "good" values for m that optimize both network traffic and detection time. We do so in the next section (Analysis).

We point out that Spatial Pinging (Medley) is a generalization of SWIM. When $m = 0$, spatial pinging degenerates to SWIM with uniform target selection. $m = \infty$ means that each member uniformly pings to its closest neighbours.

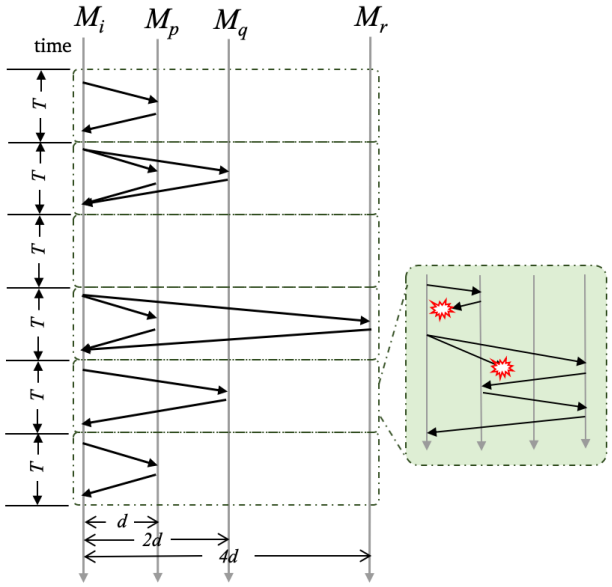


Figure 2. Example of ping target selection in Medley.

Other components: Just like SWIM, Medley disseminates information by piggybacking atop pings, acks, and indirect pings (Sec. 2, "SWIM Dissemination Component"). This is a gossip style of dissemination and is also used to disseminate node join/leave information.

Medley is able to seamlessly borrow optimizations from SWIM. One such important optimization is suspicion, which allows mistakenly-detected alive nodes a second chance to disprove their false detection. Here a detected node is not marked as failed but instead is suspected and this suspicion gossiped to other nodes (via pings and acks). If another node successfully pings the suspected node via normal pinging, before the suspicion times out, the suspected node rejuvenates in membership lists and is not deleted from membership lists. More details can be found in the SWIM paper [11].

3.2 Analysis

We analyze Medley's spatial pinging under certain idealized assumptions. For tractability, we assume that: i) the N nodes are uniformly spread with a density of D, and ii) a pinging node picks targets only up to a distance of R away.

First, to minimize detection time we wish to maximize the expected number of pings a given node receives during a pinging period. We denote this expected number as E[Pings received per period] or $EP(m)$, where:

$$\begin{aligned} EP(m) &= \left(\int_0^R \frac{1}{r^m} D(2\pi r) \cdot dr\right) \cdot \frac{1}{\pi R^2 D} \\ &= \left(\int_0^R \frac{1}{r^{m-1}} \cdot dr\right) \cdot \frac{2}{R^2} \end{aligned} \tag{1}$$

In the first line of the equation, the integral term contains the probability of being picked as a ping target($\frac{1}{r^m}$), multiplied by the number of nodes in an annulus at radius r ($D(2\pi r) \cdot dr$). The term beyond parentheses is a normalizing constant to ensure that when $m = 0$, which is the uniform default SWIM, Equation 1 comes to an expected 1 received ping.

Second (along with maximizing ping probability), we simultaneously wish to minimize communication cost $C(m)$ incurred by pings received at a given node. A message transits over multiple hops in the underlying ad-hoc network. Assuming a fixed size for messages, $C(m)$ is proportional to the number of hops incurred by the message. Again for tractability, we calculate a message's cost as proportional to the distance between its sender and receiver (as this is correlated with hop count). We obtain:

$$\begin{aligned} C(m) &= \left(\int_0^R \frac{1}{r^m} D(2\pi r) \cdot r \cdot dr\right) \\ &= \left(\int_0^R \frac{1}{r^{m-2}} \cdot dr\right) \cdot (2\pi D) \end{aligned} \tag{2}$$

This is obtained by multiplying the expected number of pings in the annulus of radius r (similar to Equation 1), by the communication cost incurred by the multi-hop network, which is proportional to the target distance r.

In order to simultaneously minimize $C(m)$) and maximize $EP(m)$, we define our optimization function that we wish to maximize as: $Ratio(m) = \frac{EP(m)}{C(m)}$.

Theorem 1. *Medley's spatial pinging: (i) provides completeness, and (ii) optimizes $Ratio(m)$ at the following values of exponent m:*

1. *If the ratio of deployment area dimension to inter-node distance is high, then $m = \infty$ is optimal;*
2. *If the ratio of deployment area dimension to inter-node distance is low, then $m = 3$ is optimal.*

Proof. First, to prove completeness, consider a failure of node M_j. We observe that with at least one non-faulty node M_i in the system, M_i has a non-zero probability of pinging M_j during any protocol period subsequennt to M_j's failure. Because of the (biased) randomness of picking ping targets, M_i is guaranteed to *eventually* pick M_j as a ping target in a future period. M_j will be unresponsive (because it is failed), and thus M_i will mark M_j as failed.

Second we find the optimal value of m. Explanding the expected ping count $EP(m)$ gives us:

$$EP(m) = \begin{cases} \frac{R^{2-m}}{2-m} \cdot \frac{2}{R^2} & m < 2 \\ \log(\frac{R}{d}) \cdot \frac{2}{R^2} & m = 2 \\ (\frac{1}{d^{m-2}} - \frac{1}{R^{m-2}}) \cdot \frac{1}{m-2} \cdot \frac{2}{R^2} & m > 2 \end{cases} \quad (3)$$

where d represents the distance to the nearest node (for a 2-dimensional deployment, $d \propto \frac{1}{\sqrt{D}}$).
Similarly, for communication cost $C(m)$:

$$C(m) = \begin{cases} \frac{R^{3-m}}{2-m} \cdot (2\pi D) & \text{when } m < 3 \\ \log(\frac{R}{d}) \cdot (2\pi D) & \text{when } m = 3 \\ (\frac{1}{d^{m-3}} - \frac{1}{R^{m-3}}) \cdot \frac{1}{m-3} & \text{when } m > 3 \end{cases} \quad (4)$$

Table 1 shows the variation of $Ratio(m) = \frac{EP(m)}{C(m)}$ as m is increased (second column). To make comparisons tractable, the third column shows the normalized value $\frac{Ratio(\infty)}{Ratio(m)}$. We wish to minimize this value (in order to maximize $Ratio(m)$).

From the table, we can eliminate some possibilities for optimizing $Ratio(m)$:

1. $m = 0$ can be ignored as $Ratio(m = 1)$ is higher than $Ratio(m = 0)$,
2. $m = 2$ can be ignored as $\frac{x}{\log(x)}$ has a minimum of $e(> 1)$,
3. $m = 1$ can be ignored as $\frac{Ratio(3)}{Ratio(1)} = \frac{x}{2\log(x)}$ has a minimum at $\frac{e}{2} > 1$.

Therefore, the choice for optimizing $Ratio(m)$ boils down to either $m = 3$ or $m = \infty$. Next we observe that:

1. If $R \gg d$ (in particular $\frac{R}{d} > e \simeq 2.718$ or $\log(\frac{R}{d}) > 1$, then $m = \infty$ is optimal. In other words, *if the dimension of the IoT installation area is much larger than inter-node distances, local pinging is optimal.*
2. If $\frac{R}{d} < e \simeq 2.718$, $m = 3$ is optimal. In other words, *for small installation areas (e.g., a room or a floor, where R is small), or areas of low node density (where inter-node distance d is high), Medley with $m = 3$ is optimal.*

Table 1. Ratio of expected number of pings (need to maximize) to Communication (need to minimize). Here $x = \frac{R}{d}$. Constants elided.

m	$Ratio(m)$	$\frac{Ratio(\infty)}{Ratio(m)}$
0	$\frac{3}{R^3}$	$\frac{2}{3} \cdot x$
1	$\frac{4}{R^3}$	$\frac{1}{2} \cdot x$
2	$\frac{2\log(\frac{R}{d})}{R^3}$	$\frac{x}{\log(x)}$
3	$\frac{2}{dR^2\log(\frac{R}{d})}$	$\log(x)$
>3	$\frac{2 \cdot (m-3)}{(m-2) \cdot dR^2}$	Increasing, tends to 1

□

Theorem 2. *In an area with symmetric pinging (e.g., large deployment, or 3 dimensional area), when Medley is configured to have each node send 1 ping per period [2], it achieves an O(1) expected time for failure detection, while imposing an O(1) message load.*

Proof. Consider a system of N nodes $M_1, M_2, \ldots M_N$. Without loss of generality, let M_1 be the node failing. Denote as $PP_m(i)$ the probability of M_i pinging M_1 in a given period, according to the normalized spatial ping distribution and m. Because each Medley node sends 1 ping period, by symmetry, a node M_1 will also receive an expected 1 ping per period. This means that $\sum_{k=2}^{N} PP_m(k) = 1$, for all values of spatial exponent m we may choose.

Now the probability that *at least one* of the nodes $M_2, \ldots M_N$ picks M_i as ping target in a protocol period (and thus detects its failure) is $FP(m) = 1 - \prod_{k=2}^{N}(1 - PP_m(k))$. Because the product of terms with a fixed sum ($\prod_{k=2}^{N}(1 - PP_m(k))$) is *maximized* when all terms ($PP_m(k)$) are identical, we have for all m, $FP(m) \geq FP(m = 0)$.

When $m = 0$ (the default uniform SWIM), each of the nodes $M_2, \ldots M_N$ pings M_i per period with identical probability $\frac{1}{N-1}$. Thus, $FP(0) = 1-(1-\frac{1}{N-1})^{N-1} \simeq 1-e^{-1}$ for large N. This is equivalent to tossing a coin with heads probability $(1 - e^{-1})$ per period. Thus: i) the expected detection time at $m = 0$ is $O(\frac{1}{1-e^{-1}})$ periods, which is O(1); and ii) the time

[2] Note that this is a different assumption from the analysis in Equation 3, but is closer to our real implementation.

for the failure to be detected with high probability (w.h.p.) $(1 - \frac{1}{N})$ is $log_{(\frac{1}{1-e^{-1}})}(N)$ periods.

Since $FP(m) \geq FP(0)$, the expected detection time and w.h.p. detection time for spatial pinging are both $\leq$ the corresponding values for $m = 0$.

□

4 Time-Bounded Failure Detection

Theorem 1 was able to prove that detection is eventual. In practice this could still mean particularly long detection times in IoT scenarios. Consider a node M_i that is "far" from most other nodes. Because ping probabilities to M_i are low, when M_i fails, the biased target selection choices imply that it may be an arbitrarily (and indeterminately) long time for the first non-faulty node to pick M_i as ping target.

We now present an optimization that preserves the biased randomness of the Medley's spatial pinging from Section 3, but is additionally able to specify an absolute time bound on how long a failed node takes to be detected.

4.1 Design of Time-Bounded Medley

The key idea is to ping via a round-robin mechanism which is weighted by ping probability.

Algorithm 1 Time-bounded target selection in a super round from a single node M_i's view.

Require: Runtime Probability List $\mathcal{P}_{M_i}$

▷ *Super round:* Create initial bag $\mathcal{BAG}_{M_i}$

1: **for** each p_j in $\mathcal{P}_{M_i}$ **do**
2: Put $\lceil \frac{p_j}{p_{min}} \rceil$ into $\mathcal{BAG}_{M_i}$
3: **end for**
4: Create an empty set $onePassTargets$

5: ▷ Start target selection
6: **while** $\mathcal{BAG}_{M_i}$ is not all zeros **do**
7: ▷ Initialize new *Pass* if needed
8: **if** $onePassTargets$ is empty **then**
9: $onePassTargets$ = $\{M_j\}$ for all j that $Count_j > 0$ in $\mathcal{BAG}_{M_i}$
10: **end if**
11: ▷ One *Period*
12: Randomly pick one node in $onePassTargets$ as PING target
13: Remove M_j from $onePassTargets$
14: Reduce $\mathcal{BAG}_{M_i}(M_j)$ by one
15: **end while**

Consider a member (node) M_i with membership list $\mathcal{ML}_i$, currently containing K entries $(M_1, M_2, \ldots, M_K)$. M_i also maintains a runtime probability list $\mathcal{P}_{M_i} = [p_1, p_2, p_3 \ldots p_K]$, where p_j is the pinging probability of respective member M_j from $\mathcal{ML}_i$.

The p_j values in $\mathcal{P}_{M_i}$ are calculated using the spatial ping probabilities of Section 3. The pseudocode for our approach is shown in Algorithm 1. We explain below.

Let $p_{min} = \min\{\mathcal{P}_{M_i}\}$, the lowest probability among all non-faulty members in $\mathcal{ML}_i$. Now, denote $Count_j = \lceil \frac{p_j}{p_{min}} \rceil$. We create a initial *bag list* as $\mathcal{BAG}_{M_i} = [Count_1, Count_2, Count_3 \ldots Count_K]$ (Line 1 - 3).

The weighted round-robin pinging at node M_i creates a *bag* B_i which consists of $Count_j$ instances of node M_j for each $M_j \in \mathcal{ML}_i$. This can be thought of as a bag of balls, with $Count_j$ balls of color M_j.

During each period, M_i picks one ball from this bag (without replacement), and uses the corresponding member as ping target for that period. The bag is created at the start of a *super round* (which consists of multiple periods), and a super round completes when the bag is empty. Thus, a super-round consists of $(\Sigma_1^K Count_j)$ number of protocol periods.

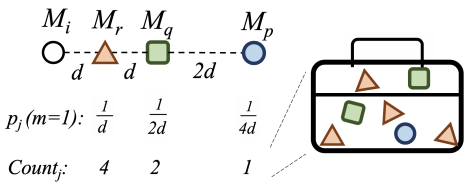
Picking these balls (targets) uniformly at random from the bag may lead to a high variance in detection times. To reduce this variation, we introduce the notion of *passes*. Algorithm 1 depicts how M_i selects targets in a super round. At M_i, ping target selection is done randomly but in multiple passes through the bag. Each pass consists of multiple periods. In each pass at M_i, every node M_j (in M_i's bag), which has at least one leftover instance in the bag, is touched (removed, and pinged) only once. These instances are removed in a random order (Line 8 - Line 13).

Suppose a particular pass contains r instances (thus consisting of r protocol periods). Then during these r periods, M_i sequentially picks one instance as ping target based on the order. When the final pass is done (and no instances are left in the bag), all instances are put back in the bag, a new super round is started, and the above process is repeated.

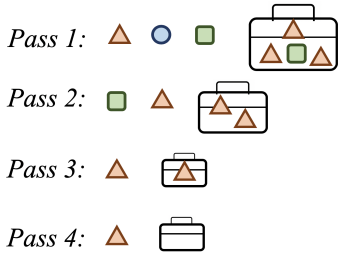
Note that the different super rounds may contain different numbers of periods, as the membership list is continuously changing (we discuss node joins and leaves in Section 4.3).

Fig. 3 shows an example of Algorithm 1 in action. There are four active members in the network, aligned topologically in a straight line. $\|M_iM_r\| = \|M_rM_q\| = \frac{1}{2}\|M_qM_p\| = d$ (as Fig. 3a). Thus, M_r, M_q, M_p are $d, 2d$ and $4d$ away from M_i respectively. When $m = 1$, $\mathcal{P}_{M_i} = [\frac{1}{d}, \frac{1}{2d}, \frac{1}{4d}]$, $p_{min} = \frac{1}{4d}$. Thus, $\mathcal{BAG}_{M_i} = [4, 2, 1]$ respectively for M_r, M_q and M_p.

At the start of this super round, there are $1 + 2 + 4 = 7$ instances in the bag at M_i. Based on our protocol, Fig. 3b shows that M_i sequentially pings M_r, M_p, M_q in the first three time periods. Then, M_p's instance is no longer in this bag (only three for M_r and one for M_q left), so in Pass 2 M_i pings M_q and M_r in the next two time periods. Only two M_r instances are left after this. Passes 3 and 4 each pick one M_r for one period each. This concludes the super round for M_i, and new bag is created again for the next super round based on the latest $\mathcal{ML}_i$.



(a) Neighbor distances and initial full bag of M_i



(b) Target selection example of M_i

Figure 3. An example of time-bounded target selection in one super round (seven time periods in total, four members in network, $m = 1$).

4.2 Time Bound

The approach above preserves relative ping selection probabilities because ping $Count_i$'s are normalized derivations of ping probabilities p_i. At the same time, this protocol provides time-bounded completeness, as we prove now.

Theorem 3. *In a system of N IoT nodes, consider a non-faulty node M_i and a faulty node M_j in M_i's membership list. Let α be the highest $Count_k$ in M_i's bag counts (i.e., $\mathcal{BAG}_{M_i}$). Then: Medley guarantees M_i detects M_j's failure within a number of pinging periods that is upper-bounded by:*

$$((N-2)\cdot\alpha)+(N-1)$$

Proof. The worst case occurs when: a) M_j has the lowest count (=1) in M_i's membership list (bag), i.e., M_j is the farthest node from M_i, b) all other $(N-2)$ nodes in M_i's membership list share the same $Count_k$ value of α, and in the execution run: c) M_i creates a new bag, and the *first* node it picks to ping is M_j, and d) this first ping succeeds but M_j fails right afterwards.

From this point onwards: *(i)* M_i will spend the rest of this super round by executing $(N-2)\cdot\alpha$ periods pinging nodes other than M_j. At the start of the next super round, when M_i creates a new bag, the first pass will pick every node once, including M_j. Thus, the worst case occurs when M_j is picked *last* at the end of this first pass (in this next bag). This means: *(ii)* M_i will take another $(N-2)$ protocol periods to get around to pinging M_j. Finally: *(iii)* one additional protocol period is needed where M_i actually pings M_j.

Adding *(i), (ii), and (iii)*, the worst-case detection time of faulty node M_j at M_i is (in protocol periods):

$$((N-2)\cdot\alpha)+(N-1)$$

□

4.3 Node Joins and Removals

If a new node M_j is added to, or removed from, M_i's membership list just as the bag is about to be refilled, then all the members' ping probabilities (and thus *Counts*) are recalculated and normalized to reflect the changed membership. Additionally, Medley also allows node joins and removals in the midst of passes–the only rule required for correctness (to preserve relative ping probabilities) is to normalize the ping probability (and thus Counts) of the added/removed nodes to match current super round progress, based on the leftover nodes in the bag. When the bag becomes empty next, probabilities (and thus Counts) of all other members are recalculated and re-normalized anew.

5 System Design

We now discuss practical considerations that were needed in order to implement Medley in a real IoT network.

***Distance Metric*:** The analysis in Section 3.2 is based on physical distances. However, exact physical locations are hard to calculate; furthermore, physical distance may not be proportional to end to end (multi-hop) routing latency. As a result, our Medley implementation replaces the use of physical distance in the ping-probability equations (Section 4) with the metric of *hop-distance*. The hop-distance is the actual total distance that a message travels between two nodes, i.e., sum of distances of all intermediate hops.

For instance, if the locations of nodes M_1, M_2 and M_3 form an isosceles right triangle with $\|M_1M_2\| = \|M_2M_3\| = 1$. Suppose M_1 pings M_3 through M_2: Medley considers the distance between M_1 and M_3 as $\|M_1M_2\| + \|M_2M_3\| = 2$ instead of $\sqrt{2}$ which would have been the physical distance.

In our deployment experiments (Section 6), for comparison, we also implemented two alternative distance metrics. These are: 1) *latency metric*: actual end-to-end latency, and 2) *hop-number metric*: count of number of hops. We found that: a) the performance of Medley with latency metric was comparable to using the hop-distance metric, and b) Medley with hop-number metric behaves similar to hop-distance metric under grid topology. Thus hereafter we only show results using the hop-distance metric, with a few differing results shown using the hop-number metric.

***Other Medley Features*:** We clarify a few other features of Medley. First, the spatial probabilities we just described are for selecting not only ping targets, but also indirect pings. (Section 2).

Second, the rejoin of a failed node is considered as a new node. We denote the ID of each node with its IP address and local timestamp when it joins the network. Two IDs with the same IP but different join timestamps are considered as two incarnations. If M_i receives an active update for M_j with ID (ip_j, ts_1) that is different from its local record for $M_j : (ip_j, ts)$, M_i will consider the old incarnation as failed and continue with the latest ID for M_j. In practice this scenario occurs rarely as Medley dissemination times are fast.

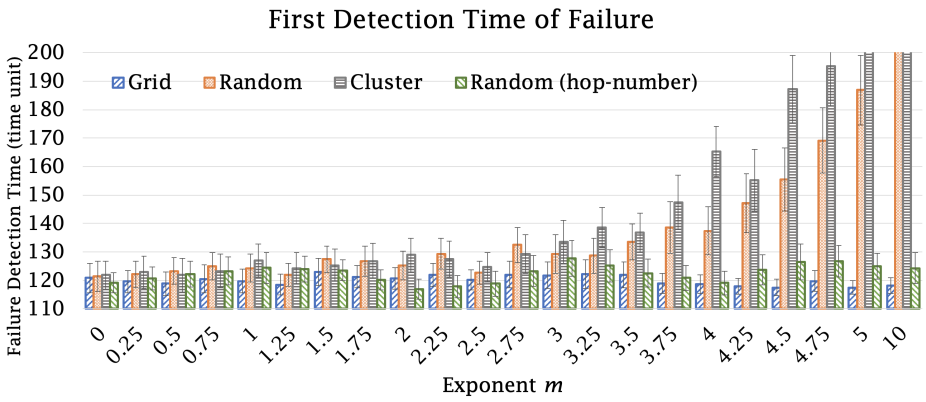


Figure 4. First failure detection time under different m and for 3 topologies. All use hop-distance metric, except Random (hop-number) which uses the hop-number metric.

6 Experiments

We present experimental results from both NS-3 simulation and Raspberry Pi deployments.

6.1 NS-3 Simulation Results

The theoretical analysis of Section 3 made simplifying assumptions about uniformity and used physical distances. In this section, we explore realistic node layouts and pinging, and measure the behavior and performance of our real Medley system.

We implemented and simulated Medley in NS-3 simulator (v3.27), and evaluated in three topologies: Random, Grid, and Cluster (multiple clusters in the area), each with 25 nodes deployed in $50m \times 50m$ area. The default number of members chosen as indirect pinger was $K = 3$, and protocol period was 20 time units. The suspicion timeout was set as $3\lceil \log(N+1) \rceil = 80$ time units in the experiments (based on [16]). Each data point reflects data from 50 independent runs with the same parameters and different seed values.

In the simulations below, unless otherwise specified, Medley uses the hop-distance metric from Section 5.

6.1.1 Failure Detection and Dissemination Latency

We define *first detection time* as the time gap between a failure occurring and the first non-faulty node detecting this failure (after suspicion time out). Figure 4 shows how the exponent m affects first detection time, with averages across 50 runs, and square root of standard deviation. Apart from the three topologies (using the hop-distance metric), the Random (hop-number) bars show the runs with hop-number metric (Section 5) under Random topology.

First, we note that due to initial detection timeout of 20 time units and the suspicion timeout $to_{suspect} = 80$ time units, the minimum possible failure detection time is 100 time units. Across the three topologies using the hop-distance metric, Grid has the lowest detection time, with Random next, and Cluster the worst. Grid is more deterministic in assigning every node at least a small set of neighbors at short distances. On the other hand, there might be "unlucky" nodes in Cluster and Random topologies, whose neighbors all have their own closer neighbor(s). When m is high and pings stay local, unlucky nodes have fewer pingers, thus prolonging their detection times. As a result, the detection time in Grid is more stable and largely independent of m. This is a different result from the analysis in Section 3.2, and this occurred because: i) the node layout and density assumptions were different there, and ii) we are using the hop-distance metric (Section 5) to calculate ping probabilities.

For Cluster and Random topologies, *first detection time* (defined at start of this section) stays low for $m \leq 3.5$ and rises quickly when $m \geq 3.5$. This is due to two factors: i) a quick increase (with rising m) in initial bag size increases the duration of a super round (Section 4), and ii) unlucky node's lower probability (fewer instances in bag) to be as a ping target by any of its neighbours. When m is low enough (below 3.5), the bag sizes are manageable and nodes have sufficient pingers for fast failure detection. Beyond $m = 3.5$, the bag size increases quickly, and thus super round length. It takes much longer for a pinger to pick a failed unlucky node, which could be as long as a super-round in the worst case (Theorem 3).

We also observe from Figure 4 that Medley using the hop-number metric in the Random topology behaves similar to Grid topology, with relatively stable first detection time. The reason is that each member has at least one one-hop (shortest distance) neighbour as a pinger, making unlucky nodes rarer than when using hop-distance metric.

Simultaneous Failures: We test up to half the nodes being failed randomly (12 out of 25) in the Random topology. Figure 5 shows the average first detection time. The lower error bar is the *earliest* time any failure is detected, averaged across runs. The higher bar is the *latest* detection time for a non-faulty node detecting all failures, averaged across runs.

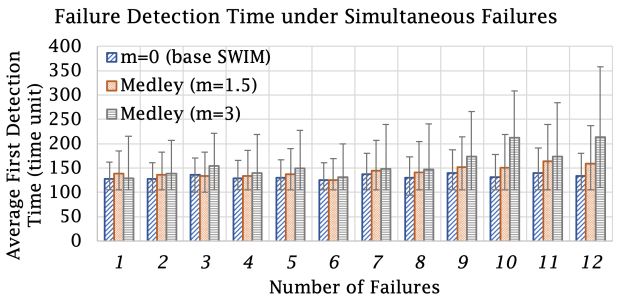


Figure 5. Failure detection time under simultaneous failures (Random topology).

When $m = 0$, each failed node has a good probability (at least $1 - (\frac{23}{24})^{12} \simeq 40.0\%$) of being picked as ping target in the round after it fails, as long as fewer than 50% nodes fail. Thus, the failure detection time stays stable when $m = 0$.

The first detection time tends to increase gently across incrementing of both m and the number of failures. For $m = 1.5$, the time to detect massive failures only takes 15% more time than one failure, and for $m = 3$, this difference is about 60%. This slight increase is caused by the ping localization that Medley brings about, and the limitation of one ping target per period. Under a given number of simultaneous failures, as m rises, a failed node waits longer to become a ping target because pings are localized. Additionally, more unlucky nodes might be involved under more failures, thus increasing average first detection time.

Domain Failure Next we explore the effect of massive failures in an area (e.g., connected to a power breaker). 25 nodes are located in three clusters in the square area of interest. Each run randomly fails a whole cluster.

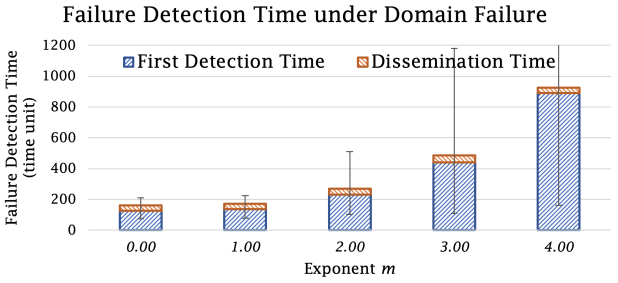


Figure 6. Failure detection time under domain failures (Cluster topology).

From Figure 6 (bars similar to Figure 5) we observe that the average first detection time stays low when $m < 2$, and as expected it increases as m rises. The dissemination time of failure information is much stabler. The increase in detection time with m is because of the ping localization under higher m, implying that the typical way a detection proceeds at higher m is from the edges of the failed cluster towards the cluster's middle. In comparison, lower values of m would detect nodes near the middle of the failed cluster much quicker due to the higher probability of far-away non-faulty pingers.

6.1.2 First Detection Distance

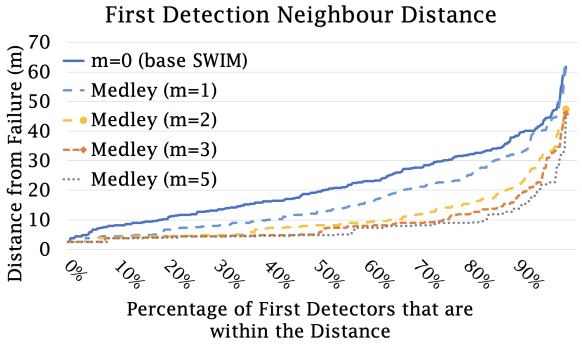


Figure 7. CDF of distances between the failed node and the first member detecting this failure (Random topology).

Figure 7 shows how far the first detector is from the failed node. Higher m exponents lead to more local detections. For $m = 3$, 90% of detections are within 20 m, which is 32% of the farthest node distance. At $m = 5$, the 90% detection distance is within 15 m (24 % of farthest node distance). Hence, higher values of m, while prohibitive for detection time, may be preferable for applications that prefer the first detecting node be closer to the faulty node (e.g., to re-replicate file replicas, or respond to local events).

6.1.3 Communication Cost

Figure 8 shows the CDF of communication cost over distance. Point (x, y) shows that y% of messages travel less than distance x. As expected, lower m (uniform pinging) incurs far more traffic across the network while Medley with higher m localizing traffic. Medley's higher values of m reduce traffic over base SWIM's $m = 0$.

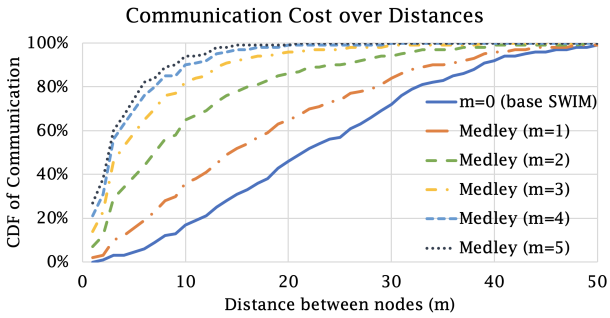


Figure 8. CDF of communication cost (number of messages) vs. distance, for different m (Random topology).

Because Medley's goal is to minimize both communication cost (bytes sent, counting multiple hops) and detection time,

we measure the square root of their product in Figure 9, for Random topology. The product cost (lower is better) falls quickly as m goes from 0 to 2.5, and then flattens out. This is because Medley pings far nodes more frequently when m is low. Due to increased ping localization and multi-hop routing, increasing m results in reduced communication, and increased detection time (Figure 4), but the former drops much quicker (hence the product falls). At higher m these two balance out. Compared to base SWIM ($m = 0$), the product cost of Medley is $\frac{Product\_Cost(m=0)}{Product\_Cost(m=3)} - 1 = 35\%$ lower.

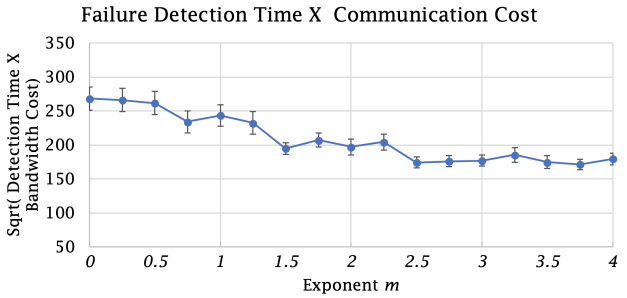


Figure 9. First failure detection rate × Communication cost, Square Root (Random topology). Lower is better.

6.1.4 False Positive Rate

We measure the rate of false detections, which are non-faulty nodes mistakenly detected as failed (this may occur due to slow nodes, dropped packets, etc.). In Table 2, we drop a random fraction r_{loss} of packets (on hops). We measure false positive rate as the fraction of time, over the entire run, that a false positive detection persists, i.e., fraction of time that at least one non-faulty node is considered failed by at least one other non-faulty node.

In Table 2, higher packet loss rates imply higher false positive rates, as expected. We also observe that false positive rate drops with increasing m (for a given packet loss rate). This is because at lower m, pings and acks have to transit more hops, thus increasing the chances that at least one of the hops will drop the packet, and a non-faulty node will be detected as failed due to a timeout. Further, at higher m, the suspicion (Section 3.1) arising from a failure detection has a higher chance of being resolved due to the more repetitive and localized nature of pings.

6.2 Deployment Evaluation

We implemented a prototype of Medley in the Raspberry Pi (RP) [22] environment. Our Java implementation was 1427 lines of code, under Raspbian 4.13. We deployed Medley in a network of 7 IoT devices with the topology shown in Figure 10. Each device was a Raspberry Pi 2 B model, with 32 GB memory, 1GB RAM and 900MHz quad-core ARM Cortex-A7 CPU. Since Raspberry 2 does not provide a built-in Wifi module, we used a TP-Link USB Wifi adapter WN725N v2.0 with

Table 2. False positive rate under different packet loss rates (r_{loss}) and exponents m.

r_{loss} \ m	0	1	2	3	4	5
10%	1.07%	0.43%	0.69%	0.08%	0.08%	0.00%
20%	2.32%	2.35%	2.05%	1.49%	1.36%	1.39%

the 2.4GHz 802.11ac standard in ad-hoc mode. We used UDP as communication protocol in the Netty framework [21].

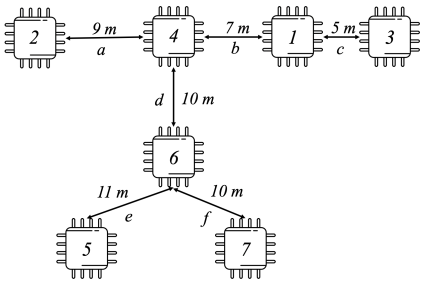
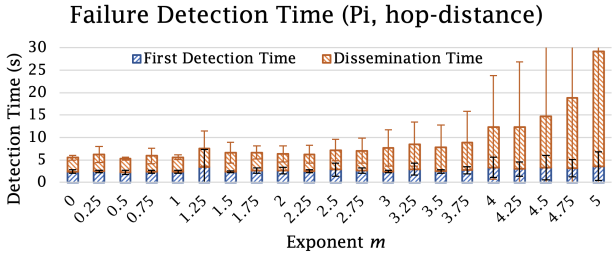
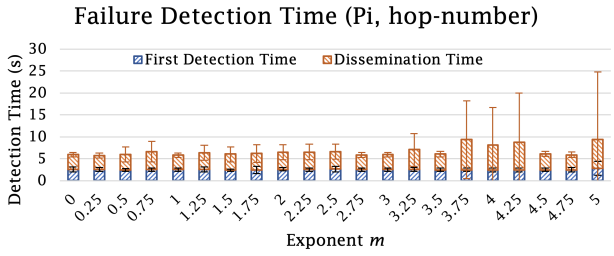


Figure 10. Topology of Raspberry Pi deployment.

6.2.1 Failure Detection and Dissemination Latency



(a) Hop-distance metric.



(b) Hop-number metric.

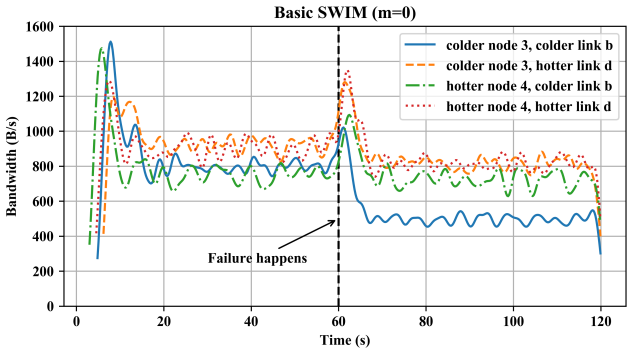
Figure 11. First failure detection time and dissemination time for Raspberry Pi experiments for 2 distance metrics.

From Figure 11 (15 data points per failure, with average and standarad deviation), we observe that the failure detection time under both hop-distance and hop-number metrics are relatively stable as m varies. The limited number of Pi

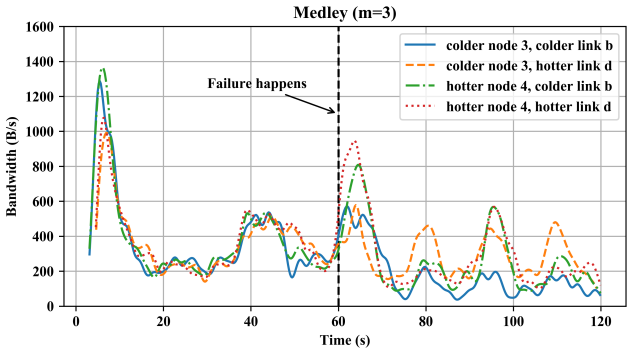
nodes restricts the number of unlucky nodes (those with no nearby neighbors), leading to low first failure detection time.

The dissemination time for hop-number metric is stable, while the dissemination time for hop-distance increases when m becomes larger (beyond m>3.5). This is because disseminating failure information "out" of a region of isolated nodes (e.g., nodes 1, 3 in Figure 10) takes a while since spatial pinging (hence piggybacking of failure information) stays largely local especially when m is high. Using the hop-number metric, each node has multiple closest neighbors, allowing faster spread. Further, the *bag* size is smaller (as counts vary less across nodes), limiting super-round time and thus dissemination time.

6.2.2 Bandwidth Cost over Time



(a) Base SWIM (m = 0). Hop-number metric.



(b) Medley (m = 3). Hop-number metric.

Figure 12. Bandwidth change timeline. Failures occur at t = 0. Using the hop-number metric.

We denote links that lie on more routing paths (of node pairs) as *hotter links*, and those on fewer paths as *colder links*. Figure 12 plots real-time bandwidth on a hotter link d and a colder link b. In each run a node fails at time 60 (hotter node 4 or colder node 3). Compared to m = 0, m = 3 consumes lower bandwidth on average (61.8% less for link d, and 52.9% less for link b), but fluctuates inside a super round.

Both far pings and local pings tend to go through hotter links. When m increases, far pings reduce dramatically, saving bandwidth. Bandwidth cost is high right after new nodes join (time 5 to 10) and right after failures occur (time 60 to 70)—this is due to increase in indirect pings. Larger exponent values (m) mean that a failure will cause bandwidth to rise more (3× at m = 3 and 1.5× at m = 0). Yet the peak bandwidth consumption in Medley (m = 3) stays lower than base SWIM's (m = 0).

This difference between SWIM and Medley's post-failure bandwidth is largely due to the spatial target selection. While SWIM spreads pings, acks and indirect pings uniformly, Medley's common case traffic (pings and acks) is largely local. Indirect pings that jump up after a failure consume more bandwidth, but because they are also chosen spatially, Medley's post-failure bandwidth stays lower than SWIM's.

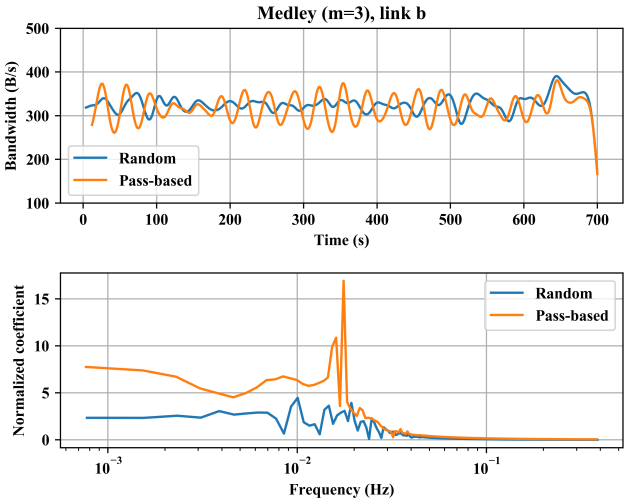


Figure 13. Run-time bandwidth on random vs pass target selection under Medley (m = 3). Failed colder node 3, measuring link b. Top: Bandwidth. Bottom: Bandwidth's FFT.

When m is relatively high, bandwidth usage has a periodic behavior caused by the cyclical nature of the super-round. Figure 13 depicts the bandwidth pattern on link b under m = 3 for the two strategies of random and pass-based from the bag (Section 4.1). The plot shows: 1) bandwidth over time (top figure), and 2) its Fast Fourier transform (FFT) (lower figure). This is a 700 second run with no failures.

We first observe that the bag selection strategy does not affect the *average* bandwidth. Second, the random selection from bag has lower bandwidth fluctuation over time, while pass-based has bigger amplitudes. This is because in the pass-based approach (Algorithm 1), the pings in the second half of each super-round tend to focus on close neighbors (a small group of nodes which have higher counts in the bag), leading to temporally unbalanced communication load on links. In comparison, selecting from the bag targets at random (rather than via passes) has less pronounced periodicity.

Although random strategy benefits from balanced bandwidth, it has longer detection times: $2 \times ((N-2) \cdot \alpha) + 1$

periods, almost twice as pass-based (for high m). If the application prefers reducing detection time than minimizing bandwidth, the pass-based approach is preferable.

7 Discussion

***Partial Membership Lists*:** Medley maintains full membership lists, useful for building a swath of distributed algorithms (Section 1). Nevertheless, full membership lists can be "pared" down to partial membership lists, without affecting properties, while reducing overhead. Two examples follow. Ex. 1: If a multicast tree (built atop Medley) uses only nearby neighbors, the partial membership list can maintain mostly nearby neighbors. Ex. 2: It is well-known that uniformly-randomly-selected partial membership lists give identical properties as a full list, for gossip multicast applications [15]. For this case, Medley's partial membership lists could be built in one of two ways: i) apply a uniform-random selection strategy to pick the partial membership list, and use spatial pinging, or ii) apply the spatial distribution to pick the partial membership list, and use uniform-random pinging. In both these cases, the results of [15] would extend, meaning that gossip over Medley with partial lists would behave identically as gossip over Medley with full lists.

***Topology Optimizations*:** An open direction is leveraging knowledge of network topology and optimizing indirect ping target selection. For instance, one could avoid intersecting routes, route pings/acks avoiding failure domains, etc. Another possibility is randomizing (uniformly) the indirect ping selections, to reduce going via failed nodes in correlated failure scenarios.

***Byzantine failures*:** We have focused on crash failures. Classical techniques for Byzantine failures are quite different from those for crash failures, focusing on tolerance rather than detection [6]. This is an open direction for IoT networks.

***Code Availability*:** Medley code is available at: https://github.com/RuiRitaYang/Medley.git.

8 Related Work

Failure Detection Techniques Failure detection in datacenter environments is a well-studied area. Heartbeating [3] is one of the earliest failure detectors where each process periodically sends a heartbeat message to either every node, or a subset thereof. Each receiver maintains the list of heartbeat counts, and if one has not been updated for a timeout, it is marked as failed. Authors of [28] describe a gossip-style way of disseminating heartbeats. However, this still incurs a super-quadratic increase in network load as the system scale increases. Authors of [17] use a hierarchical gossiping protocol and an adaptive multicast dissemination framework to reduce network overheads. SWIM [11] solves the network overhead problem by separating the failure detection operations from membership update dissemination and by using pinging for failure detection. SWIM provably gives the optimal tradeoff between detection time and bandwidth [16]. FUSE [14] uses applications to disseminate failure information in distributed systems to reduce network costs.

Failure Detection in IoT Networks Existing IoT failure detection schemes largely focus on data anomalies and can be used orthogonally with Medley. Sympathy [23] uses flooding to calculate next-hops and neighbors, and then analyzes these in a centralized way. It aggregates distributed data at the sink and detects failure by finding insufficient flow of incoming data. [25] proposes a distributed sensor node failure detection method called Memento, wherein sensor nodes detect problematic nodes by cooperatively monitoring each other. However, Memento structured the network nodes in a tree-like way, which limits the scalability of the design. Such an algorithm can detect the failures quickly, but the aggregation behavior bottlenecks performance and causes high bandwidth cost closer to the root of the tree. For both Sympathy and Memento, our Medley-style pinging can be used to improve them.

Network layer information such as packet traces or delays is used in [8] and [13] to detect failed nodes. Such designs, while efficient, are hard to analyze mathematically and also do not generalize easily to IoT settings. DICE [9] uses context (e.g., sensor correlation, state transition probabilities) to identify faulty sensors by finding real-time values that violate the precomputed context. To process the context, the information of sensors are aggregated to aggregators, which restricts DICE's scalability. Once again, DICE can be improved by combining it with Medley. Asim et al. [4] manage faults in a wireless sensor network environment where the network is homogeneous and nodes are equal in resources. They partition the network into a virtual grid of cells to support scalability and perform fault detection. Failures within a cell are detected by entities within it and are forwarded across cells. Unlike this work, Medley does not require a homogeneous network.

9 Conclusion

We have presented design, analysis, and implementation of Medley, a decentralized membership service for distributed IoT systems running atop wireless ad-hoc networks. Our key idea is a spatial failure detector, that prefers pinging nearby nodes with an exponentially higher probability. Compared to classical SWIM, Medley detects failures just as quickly, while lowering the product of failure detection time and communication cost by 35%, and incurring low false positive rates around 2% even with 20% dropped packets.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Fred Douglis, for their invaluable help. This work was supported in part by NSF CNS 1908888 and in part by a gift from Microsoft.

References

[1] Accountability Act. 1996. Health insurance portability and accountability act of 1996. *Public Law* 104 (1996), 191.

[2] Tilak Agerwala, Joanne L. Martin, Jamshed H. Mirza, David C. Sadler, Daniel M. Dias, and Marc Snir. 1995. SP2 system architecture. *IBM Systems Journal* 34, 2 (1995), 414–446.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 1997. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *International Workshop on Distributed Algorithms*. Springer, 126–140.

[4] Muhammad Asim, Hala Mokhtar, and Madjid Merabti. 2008. A fault management architecture for wireless sensor network. In *International Wireless Communications and Mobile Computing Conference (IWCMC'08)*. IEEE, 779–785.

[5] K. Birman and T. Joseph. 1987. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM, New York, NY, USA, 123–138. https://doi.org/10.1145/41457.37515

[6] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186. http://dl.acm.org/citation.cfm?id=296806.296824

[7] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.

[8] Bor-Rong Chen, Geoffrey Peterson, Geoff Mainland, and Matt Welsh. 2008. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *International Conference on Distributed Computing in Sensor Systems (DCOSS'08)*. Springer, 79–98.

[9] Jiwon Choi, Hayoung Jeoung, Jihun Kim, Youngjoo Ko, Wonup Jung, Hanjun Kim, and Jong Kim. 2018. Detecting and identifying faulty IoT devices in smart home with context extraction. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*. IEEE, 610–621.

[10] Consul. 2014. Consul by HashiCorp: A distributed service mesh to connect, secure, and configure services across any runtime platform and public or private cloud. https://www.consul.io/

[11] Abhinandan Das, Indranil Gupta, and Ashish Motivala. 2002. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the 32nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'02)*. IEEE, 303–312.

[12] Kevin Driscoll, Brendan Hall, Michael Paulitsch, Phil Zumsteg, and Hakan Sivencrona. 2004. The real Byzantine generals. In *The 23rd Digital Avionics Systems Conference (DASC'04)*, Vol. 2. IEEE, 6–D.

[13] Ravindra Navanath Duche and Nisha P Sarwade. 2014. Sensor node failure detection based on round trip delay and paths in WSNs. *IEEE Sensors Journal* 14, 2 (2014), 455–464.

[14] John Dunagan, Nicholas JA Harvey, Michael B Jones, Dejan Kostic, Marvin Theimer, and Alec Wolman. 2004. FUSE: Lightweight guaranteed distributed failure notification. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*.

[15] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. 2003. Peer-to-Peer Membership Management for Gossip-Based Protocols. *IEEE Trans. Comput.* 52, 2 (Feb. 2003), 139–149. https://doi.org/10.1109/TC.2003.1176982

[16] Indranil Gupta, Tushar D Chandra, and Germán S Goldszmidt. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*. ACM, 170–179.

[17] Indranil Gupta, Anne-Marie Kermarrec, and Ayalvadi J Ganesh. 2002. Efficient epidemic-style protocols for reliable and scalable multicast. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*. IEEE, 180–189.

[18] Krasimira Kapitanova, Enamul Hoque, John A Stankovic, Kamin Whitehouse, and Sang H Son. 2012. Being SMART about failures: assessing repairs in SMART homes. In *Proceedings of the 13th ACM Conference on Ubiquitous Computing (UbiComp'12)*. ACM, 51–60.

[19] Lucie Lozinski. 2016. How Ringpop from Uber Engineering helps distribute your application. Retrieved Apr. 2019 from https://eng.uber.com/intro-to-ringpop/

[20] MarketsandMarket™. 2018. Internet of Things (IoT) market by software solution (real-time streaming analytics, security solution, data management, remote monitoring, and network bandwidth management), service, platform, application area, and region - global forecast to 2022. https://www.marketsandmarkets.com/PressReleases/iot-m2m.asp

[21] Netty. 2003. Netty project: An asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients. https://netty.io/

[22] Raspberry Pi. 2016. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[23] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. 2005. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International Conference On Embedded Networked Sensor Systems (SenSys'05)*. ACM, 255–267.

[24] General Data Protection Regulation. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46. *Official Journal of the European Union (OJ)* 59, 1-88 (2016), 294.

[25] Stanislav Rost and Hari Balakrishnan. 2006. Memento: A health monitoring system for wireless sensor networks. In *The 3rd Annual IEEE Communications Society on Sensor and Adhoc Communications and Networks (SECON'06)*, Vol. 2. IEEE, 575–584.

[26] Serf. 2014. Serf by HashiCorp: Decentralized cluster membership, failure detection, and orchestration. https://www.serf.io/

[27] Amit Kumar Sikder, Hidayet Aksu, and A Selcuk Uluagac. 2017. 6thsense: A context-aware sensor-based attack detector for smart devices. In *The 26th USENIX Security Symposium (USENIX Security'17)*. 397–414.

[28] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. 2009. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'09)*. Springer-Verlag, 55–70.

[29] David Watkins. 2018. Global smart speaker shipments and revenue by model and price band: Q2 2018. https://www.strategyanalytics.com/access-services/devices/connected-home/smart-speakers-and-screens/market-data/report-detail/global-smart-speaker-shipments-and-revenue-by-model-and-price-band-q2-2018

[30] Juan Ye, Graeme Stevenson, and Simon Dobson. 2016. Detecting abnormal events on binary sensors in smart home environments. *Pervasive and Mobile Computing* 33 (2016), 32–49.