Self-Healing of Byzantine Faults

Jeffrey Knockel, George Saad and Jared Saia
Department of Computer Science, University of New Mexico
email: {jeffk,gwagdy,saia}@cs.unm.edu.

Abstract

Recent years have seen significant interest in designing networks that are *self-healing* in the sense that they can automatically recover from adversarial attack. Previous work shows that it is possible for a network to automatically recover, even when an adversary repeatedly deletes nodes in the network. However, there have not yet been any algorithms that self-heal in the case where an adversary takes over nodes in the network. In this paper, we address this gap.

In particular, we describe a communication network over n nodes that ensures the following properties, even when an adversary controls up to $t \leq (1/4 - \epsilon)n$ nodes, for any positive ϵ. First, the network provides point-to-point communication with bandwidth and latency costs that are asymptotically optimal. Second, $O(t(\log^* n)^2)$ message corruptions occur in expectation, before the adversarially controlled nodes are effectively quarantined so that they cause no more corruptions. We present empirical results showing that our approach may be practical.

"Fool me once, shame on you. Fool me twice, shame on me." - English proverb

1 Introduction

Self-healing algorithms protect critical properties of a network, even when that network is under repeated attack. Such algorithms only expend resources when it is necessary to repair damage done by an attacker. Thus, they provide significant resource savings when compared to traditional robust algorithms, which expend significant resources even when the network is not under attack.

The last several years have seen exciting results in the design of self-healing algorithms [2, 14, 7, 8, 12, 16]. Unfortunately, none of these previous results handle *Byzantine faults*, where an adversary takes over nodes in the network and can cause them to deviate arbitrarily from the protocol. This is a significant gap, since traditional Byzantine-resilient algorithms are notoriously inefficient, and the self-healing approach could significantly improve efficiency.

In this paper, we take a step towards addressing this gap. For a network of n nodes, we design self-healing algorithms for communication that tolerate up to a 1/4 fraction of Byzantine faults. Our algorithms enable any node to send a message to any other node in the network with bandwidth and latency costs that are asymptotically optimal.

Moreover, our algorithms limit the expected number of message corruptions. Ideally, each Byzantine node would cause $O(1)$ corruptions; our result is that each Byzantine node causes an expected $O((\log^* n)^2)$ corruptions.[1] Thus, we must amend our initial proverb to: *"Fool me once, shame on you. Fool me* $\omega((\log^* n)^2)$ *times, shame on me."*

1.1 Our Model

We assume an adversary that is *static* in the sense that it takes over nodes before the algorithm begins. We call the nodes controlled by the adversary *bad* and the remaining nodes *good*. The bad nodes may arbitrarily deviate from the protocol, by sending no messages, excessive numbers of messages, incorrect messages, or any com-

[1] Recall that $\log^* n$ or the iterated logarithm function is the number of times logarithm must be applied iteratively before the result is less than or equal to 1. It is an extremely slowly growing function: e.g. $\log^* 10^{10} = 5$

bination of these. The good nodes follow the protocol. We assume that the adversary knows our protocol, but is unaware of the random bits of the good nodes.

We further assume that each node has a unique ID. We say that node p has a link to node q if p knows q's ID and can thus directly communicate with node q. Also, we assume the existence of a public key digital signature scheme, and thus a computationally bounded adversary. Finally, we assume a partially synchronous communication model: any message sent from one good node to another good node requires at most Δ time steps to be sent and received, and the value Δ is known to all nodes. However, we assume a *rushing adversary*, so the bad nodes receive all messages from good nodes in a round before sending out their own messages.

Our algorithms make critical use of quorums and a quorum graph. We define a *quorum* to be a set of $\theta(\log n)$ nodes, of which at most a 1/4 fraction are bad. Many results show how to create and maintain a network of quorums [5, 9, 11, 17, 6, 1, 10]. All of these results maintain what we will call a *quorum graph* in which each vertex represents a quorum. The properties of the quorum graph are: 1) Each node is in $O(\log n)$ quorums; 2) For any quorum Q, any node in Q can communicate directly to any other node in Q; and 3) For any quorums Q_i and Q_j that are connected in the quorum graph, any node in Q_i can communicate directly with any node in Q_j and vice versa.

Communication in the quorum graph typically occurs as follows. When a node $\mathbf{s}$ sends another node $\mathbf{r}$ some message m, there is a canonical *quorum path* through the quorum graph, $Q_1, Q_2, \ldots, Q_\ell$, where $\mathbf{s} \in Q_1$ and $\mathbf{r} \in Q_\ell$. This path is determined by the ID's of both $\mathbf{s}$ and $\mathbf{r}$. A naive way to route the message is for $\mathbf{s}$ to send m to all nodes in Q_1. Then for $i = 1$ to $\ell - 1$, for all nodes in Q_i to send m to all nodes in Q_{i+1}, and for all nodes in Q_{i+1} to do majority filtering on the messages received in order to determine the true value of m. Unfortunately, this algorithm requires $O(\ell \log^2 n)$ messages. This paper shows how to reduce this cost.

1.2 Our Results

This paper provides a self-healing algorithm, *SEND*, that sends a message from a source node to a target node in the network. Our main result is summarized in the following theorem.

Theorem 1.1. Assume we have a network with n nodes and $t \leq (1/4 - \epsilon)n$ bad nodes, for any positive ϵ, and a quorum graph as described above. Then our algorithm ensures the following.

- For any call to *SEND*, the expected latency is $O(\ell)$ and the expected number of messages is $O(\ell + \log n)$, in an amortized sense.[2]
- The total number of times that a message can be corrupted in a call to *SEND* is $O(t(\log^* n)^2)$ in expectation.

1.3 Related Work

Our results are inspired by recent work on self-healing algorithms [2, 14, 7, 8, 12, 16]. A common model for these results is that the following process repeats indefinitely: an adversary deletes some nodes in the network, and the algorithm adds edges. The algorithm is constrained to never increase the degree of any node by more than a logarithmic factor from its original degree. In this model, researchers have presented algorithms that ensure the following properties: the network stays connected and the diameter does not increase by much [2, 14, 7]; the shortest path between any pair of nodes does not increase by much [8]; and expansion properties of the network are approximately preserved [12].

Our results are also similar in spirit to those of Saia and Young [15] and Young et al. [19], which both show how to reduce message complexity when transmitting a message across a quorum path of length ℓ. The first result, [15], achieves expected message complexity of $O(\ell \log n)$ by use of bipartite expanders. However, this result is impractical due to high hidden constants

[2] In particular, if we perform any number of message sends through quorum paths, where ℓ_M is the longest such path, and $\mathcal{L}$ is the sum of the quorums traversed in all such paths, then the expected total number of messages sent will be $O(\mathcal{L} + t \cdot \ell_M \log^2 n \log^* n)$. Note that, since t is fixed, for large $\mathcal{L}$ this value is $O(\mathcal{L})$.

and high setup costs. The second result, [19], achieves expected message complexity of $O(\ell)$. However, this second result requires the sender to iteratively contact a member of each quorum in the quorum path. Thus, while practical for some peer-to-peer applications, it has drawbacks in 1) *load-balancing*: for example, a single node broadcasting to all nodes through a tree of quorums must send to $\theta(n)$ messages; and 2) *anonymity:* the ID of the sender is learned by at least one node in each quorum.

As mentioned earlier, several peer-to-peer networks have been described that provably enable reliable communication, even in the face of adversarial attack [5, 3, 9, 11, 17, 1]. To the best of our knowledge, our approach applies to each of these networks, with the exception of [3]. In particular, we can apply our algorithms to asymptotically improve the efficiency of the peer-to-peer networks from [5, 9, 11, 17, 1].

1.4 Organization of Paper

The rest of this paper is organized as follows. In Section 2, we describe our algorithms. In Section 3, we prove the correctness of these algorithms; the main result of this section is a proof of Theorem 1.1. We give empirical results showing how our algorithms can improve the efficiency of the butterfly network of [5] in Section 4. Finally, we conclude and describe problems for future work in Section 5.

2 Our Algorithms

Algorithm 1 SEND($m, \mathbf{r}$)

Assumptions: Node $\mathbf{s}$ wants to send message m to node $\mathbf{r}$.

1. Node $\mathbf{s}$ calls *SEND-LEADER* $(m, \mathbf{r})$
2. With probability $1/(\log^* n)^2$, node $\mathbf{s}$ calls *CHECK* $(m, \mathbf{r})$

In this section, we describe our algorithms *SEND*, *SEND-LEADER*, *CHECK*, and *UPDATE*. The main technical challenge of our paper is in the design of the algorithm *CHECK*, which is described in Section 2.2.

Algorithm 2 SEND-LEADER($m, \mathbf{r}$)

Assumptions: m is the message to be sent, and $\mathbf{r}$ is the destination. We let $Q_1, Q_2, \ldots Q_\ell$ be the quorum path from $\mathbf{s}$ to $\mathbf{r}$ in the quorum graph.

1. Node $\mathbf{s}$ sends m to every node in Q_1
2. Each node in Q_1 sends the message it receives to the leader q_2 of quorum Q_2.
3. The leader q_2 checks for conflicting messages. If messages conflict, q_2 aborts and initiates a call to *UPDATE*.
   Otherwise, for $i = 2, \ldots, \ell - 2$ do
   (a) The leader q_i of quorum Q_i sends the message it receives to the leader q_{i+1} of quorum Q_{i+1}
4. The leader $q_{\ell-1}$ sends the message it receives to all nodes in Q_ℓ
5. The node $\mathbf{r}$ checks for conflicting messages. If messages conflict, $\mathbf{r}$ initiates a call to *UPDATE*.

2.1 Overview

Our algorithms maintain a *leader* for each quorum. We maintain the invariant that, for every quorum Q, all nodes in Q know the leader of Q. Additionally we maintain that for every quorum Q', such that Q' has an edge to Q in the quorum graph, all nodes in Q' know the leader of Q.

As described previously, we assume that when node $\mathbf{s}$ wants to send a message to a node $\mathbf{r}$, there is a canonical *quorum path* $Q_1, Q_2, \ldots, Q_\ell$, determined by the IDs of $\mathbf{s}$ and $\mathbf{r}$, such that $\mathbf{s} \in Q_1$ and $\mathbf{r} \in Q_\ell$. Our main algorithm *SEND* (Algorithm 1), has $\mathbf{s}$ call *SEND-LEADER* with the message to be sent and the ID of the node $\mathbf{r}$.

In the *SEND-LEADER* (Algorithm 2), $\mathbf{s}$ sends the message to all nodes in Q_1; these nodes send to the leader of Q_2; and the message is propagated by quorum leaders until reaching $Q_{\ell-1}$. Then the leader of $Q_{\ell-1}$ sends to all nodes in Q_ℓ, and these nodes send directly to $\mathbf{r}$.

SEND-LEADER is vulnerable to corruption.

Thus, with probability $1/(\log^* n)^2$, *SEND* next calls *CHECK* (Algorithm 3), which has the following two properties: 1) with probability at least 1/2, it determines if a message was corrupted in the previous call to *SEND-LEADER*; and 2) it is resource efficient, requiring only $O(\ell(\log^* n)^2)$ messages.

Unfortunately, while *CHECK* can determine if a corruption occurred, it does not determine the location where the corruption occurred. Thus, if *CHECK* detects a corruption, *UPDATE* (Algorithm 4) is called. When called after a corruption occurs, *UPDATE* identifies two neighboring quorums Q_i and Q_{i+1} in the path, for some $1 \leq i < \ell$, such that one of the leaders of the two quorums is bad. Then, new leaders are elected for both Q_i and Q_{i+1}.

2.2 *CHECK*

The algorithm *CHECK* is described formally as Algorithm 3. We now give an overview. *CHECK* runs for $4\log^* n$ rounds, and maintains a subset S_i for each quorum Q_i in the quorum path. Initially all S_i are empty and **s** generates a public/private key pair k_p, k_s.

In each round, for each quorum Q_i, $1 \leq i \leq \ell$, a new node is added to S_i. This new node is chosen uniformly from all nodes in Q_i. Also, in each round, a message m' is constructed, which consists of 4 items: 1) the original message m sent during *SEND-LEADER*; 2) the public key k_p generated by **s** at the start of the call to *CHECK*; 3) the ID of the receiver node, **r**; and 4) an array of random numbers, R, that are used to pick the nodes added to all the S_i sets in the current round. The message m' is signed using the private key k_s.

The node **s** sends m' to all nodes in S_1; then for all $1 \leq i \leq \ell$, all nodes in S_i send to the new node in S_{i+1} and this new node sends to all nodes in S_{i+1}. Finally, all nodes in S_ℓ send to the node **r**.

If a node has previously received the public key, k_p, then it verifies each subsequent message with it. A node initiates a call to *UPDATE* if it receives inconsistent messages or fails to receive and verify some expected message.

An example run of *CHECK* is illustrated in Figure 1. In this figure, there is a column for each quorum in the quorum path and a row for each round of *CHECK*. For a given row and column, there is a G or B in that position depending on whether the node selected in that particular round and that particular quorum is good(G) or bad(B). The left bar in each row specifies the rightmost quorum in which there is some good node that knows m'. The right bar in each row specifies the leftmost quorum in which there is some good node that does not know m'.

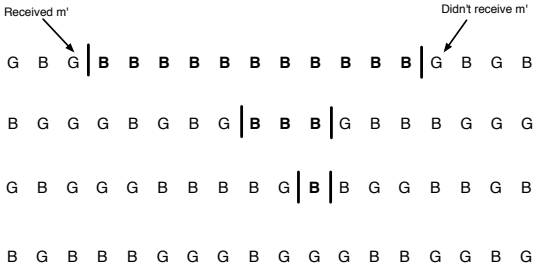

Figure 1: Example run of *CHECK*

Note that, as rounds progress, the left bar can only move rightwards, because a node that has already received k_p will call *UPDATE* unless it receives messages signed with k_p for all subsequent rounds. Further, note that the right bar can only move leftwards, since there is all-to-all communication between the nodes in the sets S_i. Finally, note that when these two bars meet, a corruption is detected.

Intuitively, the reason *CHECK* requires only $4\log^* n$ rounds is because of a probabilistic result on the maximum length run in a sequence of coin tosses. In particular, if we have a coin that takes on value "B" with probability 1/4, and value "G" with probability 3/4, and we toss it x times, then the expected length of the longest run of B's is $\log x$. Thus, if in some round, the distance between the left bar and the right bar is x, we expect in the next round this distance will shrink to $\log x$. Intuitively, we might expect that, if the quorum path is of length ℓ, then $O(\log^* \ell)$ rounds will suffice before the distance shrinks to 0. This intuition is formalized in Lemmas 1 and 2 of Section 3.

2.3 *UPDATE*

The *UPDATE* algorithm is described formally as Algorithm 4. This algorithm has each node previously involved in *SEND* broadcast all messages they have received to their own quorum and to the neighboring quorums. A pair of nodes x and y is declared to *be in conflict* if: 1) x was scheduled to send a message to y at some point in this call to *SEND*; and 2) the message that x reported that it received is different than the message that y reported that it received. *UPDATE* then finds at least one pair of nodes that are in conflict.

Moreover, in the case where a corruption occurred during the call to *SEND-LEADER*, *UPDATE* will identify a pair of neighboring quorums Q_j and Q_{j+1}, for some $1 \leq j < \ell$ such that one of the two quorums currently has a bad leader. Both leaders from Q_j and Q_{j+1} are thrown out and new leaders are elected. If these new leaders are not connected, then *UPDATE* keeps electing new leaders for these two quorums until two leaders are elected that still have an edge between them.[3] The properties of *UPDATE* are given in Lemma 3.

Since each quorum has a 3/4 fraction of good nodes, if we throw out two both leaders of Q_j and Q_{j+1}, and perform new elections, we make progress. In particular, the expected number of quorums that have good leaders will increase by a positive amount. Intuitively, we would expect that after this process repeats enough times, all quorums will have good leaders. This intuition is formalized in Lemma 4.

UPDATE makes use of a leader election protocol to 1) enable a quorum of nodes to agree on a leader; and 2) ensure that the leader agreed on is good with probability at least 3/4. We now describe how a quorum can elect a leader by using secure multiparty computation (SMPC) [13].

Let n' be the number of nodes in the quorum and let each node in the quorum be assigned a unique integer from 1 to n'. First, each node in the quorum chooses an input: an integer uniformly distributed between 1 and n'. Then, the nodes perform SMPC to find the output: the sum of all their inputs modulo n'. The node in the quorum associated with this output number becomes the new leader of the quorum.

The leader selected will be uniformly distributed provided that at least a 3/4 fraction of the nodes in the quorum are good. Finally, this leader election protocol runs in $O(1)$ time, and requires $O(\log^2 n)$ total messages.

2.4 Some Details

During the course of our algorithms, edges will be removed from the network. We assume that subsequent to the removal of an edge between node p and node q, no message is ever sent, or expected to be sent, from p to q or vice versa.

We note that in the the algorithm *SEND-LEADER*, the node $\mathbf{s}$ sends messages to all nodes in Q_1. This additional communication ensures that the nodes in Q_1 all have received a message from $\mathbf{s}$. This ensures that in the case where $\mathbf{s}$ is a bad node, it cannot cause two good leaders to be in conflict during a call to *UPDATE*. The same property holds true for the nodes in Q_ℓ and the node $\mathbf{r}$. This additional communication adds $O(\log n)$ to the message cost of *SEND-LEADER*.

3 Analysis

In this section, we prove our main result, Theorem 1.1. We first require several lemmas. Throughout this section, we let n_q represent the number of quorums in the quorum graph, and let all logarithms be base 2.

The proof of the following lemma is deferred to the appendix due to space constraints.

Lemma 1. Consider a sequence of x nodes, where each node in the sequence is bad independently with probability 1/4. Then the probability that there is any substring of length $\max(1, \log x)$ bad nodes in this sequence is no more than 1/2.

The next lemma shows that the algorithm *CHECK* catches corruptions with probability at least 1/2.

Lemma 2. Assume some bad leader has cor-

[3] No edge is ever removed between a pair of good leaders, and there are at least a 3/4 fraction of good leaders in each quorum. Thus, for each election, there is probability 9/16 of electing two good leaders. Hence, in expectation, we require only a constant number of elections before two connected leaders will be elected.

rupted a message in the last call to *SEND-LEADER*. Then when the algorithm *CHECK* is called, with probability at least 1/2, some node will call *UPDATE*.

Proof. This proof makes use of the following two facts.

Fact 1. Assume that in round i, *UPDATE* will be called if a good node chosen in round i or less at some quorum, Q_j, reliably transmits its message to a good node chosen in round i or less at some quorum, Q_k, where $1 \leq j < k \leq \ell$. Then in round $i+1$, there exist j' and k', where $j \leq j' \leq k' \leq k$,[4] such that:

- Property 1: *UPDATE* is called if a node chosen in round $i+1$ or less at $Q_{j'}$ transmits its message reliably to a node chosen in round $i+1$ or less at $Q_{k'}$.

To prove this fact, note that the good nodes in Q_j that are chosen by *CHECK* in rounds i or less, know $\mathbf{s}$'s public key, k_p. Thus they must receive uncorrupted messages signed by $\mathbf{s}$'s private key, k_s, in all rounds subsequent to i, or else *UPDATE* will be triggered.

Fact 2. Fix a round $i+1$ of *CHECK* and let j' and k' be indices satisfying Property 1 of Fact 1 that minimize the value $k'-j'$. Then in round $i+1$, *UPDATE* is called unless all nodes that are chosen between quorums $Q_{j'}$ and $Q_{k'}$ are bad.

To show Fact 2, assume by way of contradiction that it is false. Then there exists x', such that $j' < x' < k'$, where the node p' chosen in round $i+1$ at $Q_{x'}$ is good. There are two cases for what happens in round $i+1$:

- Case 1: The node p' receives the message m' sent by $\mathbf{s}$. But then the indices j' and x' satisfy Property 1 of Fact 1 and $x'-j' < k'-j'$. This contradicts the assumption that the indices j' and k' had the minimal distance among all indices satisfying Property 1 in Fact 1.

- Case 2: The node p' does not receive the message m' sent by $\mathbf{s}$. But then the indices x' and k' satisfy Property 1 of Fact 1 and $k'-x' < k'-j'$. This again contradicts the assumption that the indices j' and k' had the minimal distance among all indices satisfying Property 1 in Fact 1.

Now we can use these two facts to prove the lemma. Let X_i be an indicator random variable that it is equal to 1 if $(k'-j') \leq \log(k-j)$ and 0 otherwise. By Lemma 1, each X_i is 1 with probability at least 1/2. We require at least $\log^* n$ of the X_i random variables to be 1 in order for some node to call *UPDATE*.[5] Let $X = \sum_{i=1}^{4\log^* n} X_i$. Then $\mathbf{E}(X) = 2\log^* n$, and since the X_i's are independent, by Chernoff bounds,

$$\Pr\left(X < (1-\delta)2\log^* n\right) \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{2\log^* n}.$$

When $1-\delta = 1/2$, $\delta = \frac{1}{2}$. For $n > 16$,

$$\Pr\left(X < \log^* n\right) \leq \left(\frac{e^{\frac{1}{2}}}{(\frac{3}{2})^{\frac{3}{2}}}\right)^{2\log^* n} < \frac{1}{2}.$$

Thus the probability that *CHECK* succeeds in finding a corruption and calling *UPDATE* is at least 1/2. □

We say that a node q that is a leader of a quorum Q is *deposed* if the quorum Q elects a new leader uniformly at random with replacement. The following lemma shows that if a corruption is caught during a call to *SEND-LEADER*, *UPDATE* deposes at least one pair of leaders, and each pair contains at least one bad node. Also, *UPDATE* always removes at least one edge when it is called, and at least one endpoint of each removed edge is a bad node. The proof is deferred to the appendix.

Lemma 3. If some bad leader has corrupted a message in the last call to *SEND-LEADER*, then the algorithm *UPDATE* will 1) identify a pair of neighboring quorums Q_j and Q_{j+1}, for some $1 \leq j < \ell$ such that at least one of the two quorums currently has a bad leader; and 2) elect new leaders for these two quorums. Moreover,

[4] Note that $j' = k'$ corresponds to the case where a conflict is detected.

[5] Here we assume $\ell \leq n$. However, we can achieve the same asymptotic results assuming that ℓ is bounded by a polynomial in n.

UPDATE will always remove at least one edge from the network, and at least one endpoint of each edge removed will be a bad node.

The next lemma bounds the expected number of times that a pair of neighboring leaders is deposed. The proof is in the appendix.

Lemma 4. Assume there are j quorums in the quorum graph that have bad leaders, for any positive integer j. Then, in expectation, the number of corruptions that must be caught by *CHECK* before the leaders of all quorums are good is no more than $2j$.

We can now prove our main theorem.

Proof of Theorem. We start with resource costs. By Lemma 3, each time *UPDATE* is called, at least one edge is removed from the network. Hence, the resource costs of all calls to *UPDATE* are bounded as the number of calls to *SEND* grows large. Thus, for the amortized cost, we consider only the cost from calls to *CHECK* and *SEND-LEADER*. When sending through a path of ℓ quorums, *SEND-LEADER* has latency $O(\ell)$ and message cost $O(\ell+\log n)$. *CHECK* has latency and message cost $O(\ell(\log^* n)^2)$, but it is called only with probability $1/(\log^* n)^2$. Hence the amortized expected cost of *SEND* is $O(\ell)$ latency and $O(\ell + \log n)$ messages.

More specifically, if we perform any number of message sends through quorum paths, where ℓ_M is the longest such path, and $\mathcal{L}$ is the sum of the quorums traversed in all such paths, then the expected total number of messages sent will be $O(\mathcal{L} + t \cdot \ell_M \log^2 n \log^* n)$. This is true since each call to *UPDATE* costs $O(\ell_M \log^2 n \log^* n)$ messages, since we perform $O(\log^* n)$ Byzantine agreements over at most ℓ_M quorums. Note that, since t is fixed, for large $\mathcal{L}$ this value is $O(\mathcal{L})$.

We now bound the expected number of corruptions. Let X be a random variable giving the number of quorums which initially have bad leaders. For $1 \leq i \leq n_q$, let p_i be the probability that the i-th quorum has a bad leader. By linearity of expectation, $E(X) = \sum_{i=1}^{n_q} p_i$. Note that for $1 \leq i \leq n_q$, p_i equals the number of bad nodes in the i-th quorum divided by the size of the i-th quorum. Thus, the denominator for each p_i is $\theta(\log n)$ and the sum of all the numerators is $O(t \log n)$, since each node is in $O(\log n)$ quorums. This implies that $E(X) = O(t)$

Now by Lemma 4, the expected number of times *CHECK* must catch a corruption before the leaders of all quorums are good, is no more than $2X$. Hence, by linearity of expectation, the expected number of corruptions that must be caught is $2E(X) = O(t)$. Finally, if a bad node caused a corruption during a call to *SEND-LEADER*, then, by Lemmas 2 and 3, with probability at least $1/2$, *CHECK* will catch it. As a consequence, it will call *UPDATE*, which will elect two new leaders. *UPDATE* is thus called with probability $1/(\log^* n)^2$, so the expected total number of corruptions is $O(t(\log^* n)^2)$. □

4 Empirical Results

4.1 Setup

In this section, we empirically compare the message costs and the corrupted message counts of two algorithms via simulation. The first algorithm we simulate is the *Butterfly* algorithm from [4]. This algorithm has no self-healing properties, and simply uses all-to-all communication between quorums that are connected in a butterfly network. The second algorithm is *Loglog*, wherein we apply a modified version of our self-healing algorithm to the butterfly network.

For the *Loglog* algorithm, we modify *CHECK* so that it requires fewer messages for practical values of n. Instead of requiring $O(\ell(\log^* n)^2)$ messages per check, we modify it to require $O(\ell(\log\log n)^2)$ messages. When the nodes are picked to participate in subquorums, we replace incrementally adding one node to the S_i sets over each of $4\log^* n$ rounds, with directly adding $\log\log n$ nodes in only one round. Effectively, each $\log\log n$-sized subquorum S_i engages in all-to-all communication with its neighboring subquorum S_{i+1}.

We can show that our modified check fails with $o(1)$ probabilty. This new *CHECK* succeeds if every subquorum has at least one good node. The probability of any subquorum having only bad nodes is at most $(1/4)^{\log\log n} = 1/\log^2 n$. Union-bounding over all ℓ subquorums, the probability of our modified *CHECK* failing is at most

$\ell / \log^2 n$. For the Butterfly topology, we have $\ell = O(\log n)$, so the probability of our modified *CHECK* failing is $o(1)$.

Our simulations consist of a sequence of *queries* over the network, consisting of a pair of nodes $\mathbf{s}, \mathbf{r}$, chosen uniformly at random, such that $\mathbf{s}$ sends a message to $\mathbf{r}$. We simulate an adversary who chooses at the beginning of each simulation a fixed number of nodes to control uniformly at random without replacement. Our adversary attempts to corrupt messages between nodes whenever possible. Aside from attempting to corrupt messages, the adversary performs no other attacks to attempt to deny service.

4.2 Results

The results of our experiments are shown in Figures 2, 3 and 4. These results highlight two strengths of our self-healing algorithms (*Loglog*) when compared to algorithms without self-healing (*Butterfly*). First, the cost of a query decreases as the total number of queries increases, as illustrated in Figure 2. Second, for a fixed number of queries, the cost of a query decreases as the total number of bad nodes decreases, as illustrated in Figure 3. In particular, when there are no bad nodes, *Loglog* has dramatically less cost than *Butterfly*.

We now describe our results in more detail. Figure 2 shows the number of messages per query versus the number of queries for *Butterfly* and *Loglog* when the fraction of bad nodes is $\frac{1}{8}$. The left plot is for a network of size $n = 1{,}329$, and the right plot is for a network of size $n = 14{,}116$. The two curves intersect when the total number of queries is 5,909 and 98,168 respectively. These intersection points represent an average of 4.4 queries per node for the left plot, and 7.0 queries per node for the right plot.

Figure 3 shows the number of messages per query versus the number of bad nodes for both *Butterfly* and *Loglog*. In the left plot, the network size $n = 1{,}329$ and the number of queries is *fixed* at 10,000. In the right plot the network size is $n = 14{,}116$ and the number of queries is fixed at 100,000. The two curves intersect when the fraction of bad nodes is .182 in the left plot, and .126 in the right plot.

Figure 4 shows the number of corruptions versus the number of queries for *Loglog*, when the fraction of bad nodes is 1/8. The left plot is for a network of size $n = 1{,}329$, and the right plot is for a network of size $n = 14{,}116$. The number of corruptions flattens out at about 7.4 total corruptions for the left plot, and 408 total corruptions for the right plot.

5 Conclusion and Future Work

We have presented algorithms that can significantly reduce communication cost in attack-resistant peer-to-peer networks. The price we pay for this improvement is the possibility of message corruption. In particular, if there are $t \leq n/4$ bad nodes in the network, our algorithm allows $O((\log^* n)^2 t)$ message transmissions to be corrupted in expectation, before the bad nodes are quarantined so they cause no more corruptions. We have simulated variants of our algorithms and demonstrated that they perform well in practice, particularly as the number of queries grows large.

Many problems remain open. First, it seems unlikely that the smallest number of corruptions allowable by an attack-resistant algorithm with optimal message complexity is $O((\log^* n)^2 t)$. Can we improve this upper bound to $O(t)$ or else prove a non-trivial lower bound? Second, can we apply techniques in this paper to problems more general that enabling secure communication? For example, can we create self-healing algorithms for distributed *computation* with Byzantine faults? Finally, can we optimize constants and make use of heuristic techniques in order to significantly improve our algorithms' empirical performance?

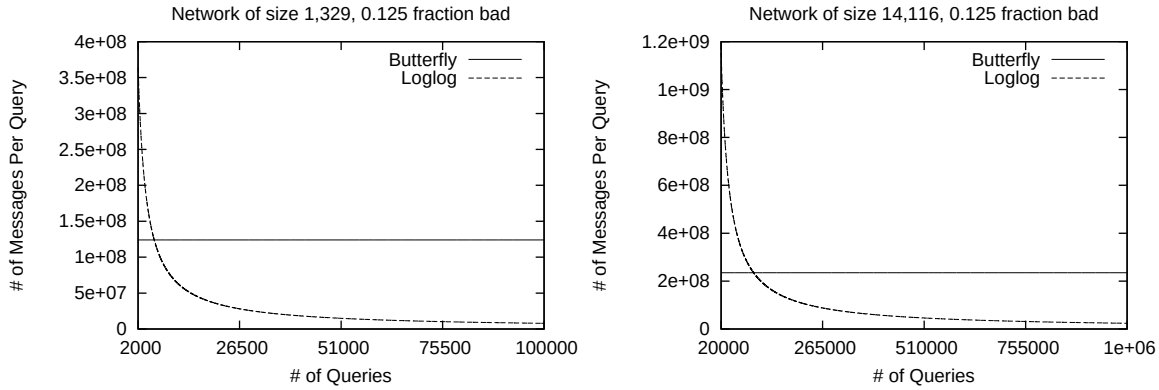

Figure 2: Number of messages per query versus total number of queries

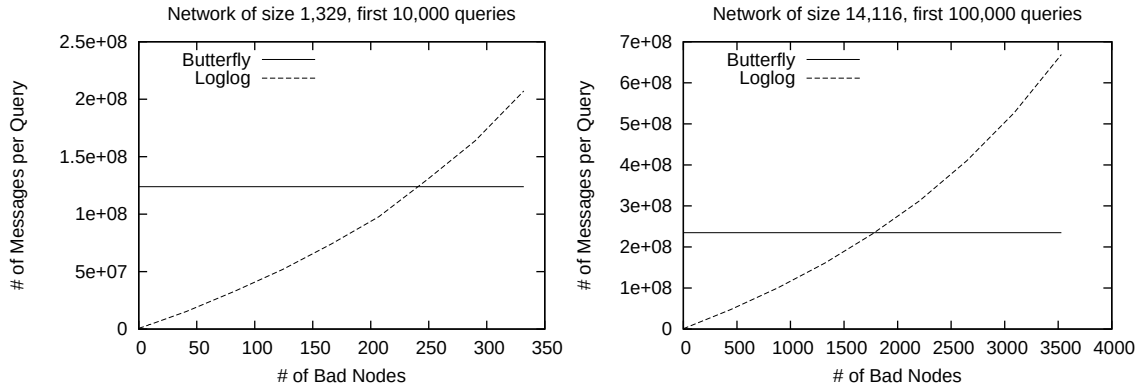

Figure 3: Number of messages per query versus the number of bad nodes

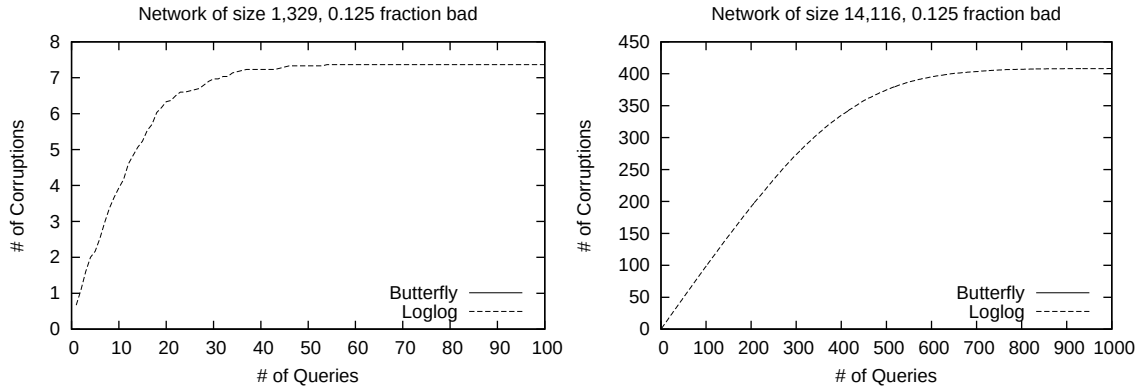

Figure 4: Number of corruptions versus number of queries

Algorithm 3 *CHECK*$(m, \mathbf{r})$

Initialization: Node **s** generates public/private key pair k_p, k_s to be used throughout this procedure. Throughout the procedure, if a node has previously received k_p, then it verifies each subsequent message with it. If a node receives inconsistent messages or fails to receive and verify an expected message, then it initiates a call to *UPDATE*. We again let $Q_1, Q_2, \ldots Q_\ell$ be the quorum path from **s** to **r** in the quorum graph. Finally, each S_j is empty initially, for $1 \leq j \leq \ell$.

for $i \leftarrow 1, \ldots, 4 \log^* n$ **do**

1. **s** sets R to be a ℓ by $C' \log n$ array of random numbers, where $C' \log n$ is the maximum size of any quorum. For any k between 1 and $C' \log n$, $R_{j,k}$ is a uniformly random number between 1 and k that will be used by quorum Q_j.
2. **s** chooses a node, x_1 uniformly at random from all neighbors in Q_1, and adds this node to S_1
3. **s** sets m' to be the messages signed by k_s consisting of m, k_p, **r**, and R
4. **s** sends m' to all i nodes in S_1 and also the leader of Q_1
5. For $j \leftarrow 1, \ldots, \ell - 1$
   (a) $S_j \leftarrow S_j$ plus the leader of Q_j
   (b) The nodes in S_j set x_{j+1} to be a node selected uniformly at random from the nodes in Q_{j+1}. To do this, they let x be the number of nodes in Q_{j+1} and select the $R_{j,x}$ node in Q_{j+1} (using a canonical ordering of the nodes based on their IDs).
   (c) The nodes in S_j set their new value of S_{j+1} to be equal to their old value union the node x_{j+1}
   (d) The nodes in S_j send to x_{j+1} both m' and the IDs of all nodes in S_{j+1}.
   (e) The node x_{j+1} sends m' to all the nodes in S_{j+1}.
6. The nodes in S_ℓ send m' to the node **r**

end for

Algorithm 4 *UPDATE*

Assumptions: All broadcasts are done via Byzantine agreement

1. The node, x, making the call to *UPDATE* broadcasts this fact to its quorum, Q', along with all the messages that x has received during this call to *SEND*. The nodes in Q' check that x received inconsistent messages before proceeding.
2. The quorum Q' propagates the fact that a call to *UPDATE* is occurring, via all-to-all communication, to all quorums $Q_1, Q_2, \ldots, Q_\ell$.
3. **s** broadcasts all messages it sent in this call to *SEND* to the nodes in Q_1 and these messages are sent via all-to-all communication to all remaining quorums $Q_2, Q_3, \ldots Q_\ell$.
4. Each node involved in this call to *SEND* compiles all messages they have received (and from whom) in this call to *SEND*, and broadcasts these messages to the nodes in its own quorum, and all neighboring quorums in the quorum path. The node **s** broadcasts the messages to the nodes in Q_1; and the node **r** broadcasts the messages to the nodes in Q_ℓ.
5. A pair of nodes x and y is declared to *be in conflict* if: 1) x was scheduled to send a message to y at some point in this call to *SEND*; and 2) the message that x reported that it received is different than the message that y reported that it received. For every pair of nodes x, y that are in conflict the edge between x and y is removed. Specifically, the edge (x, y) is removed from the edge list of all nodes in the quorums of x and y.
6. A pair of leaders, (q_j, q_{j+1}), is deposed if they are in conflict and q_j is not in conflict with q_{j-1}. The quorums Q_j and Q_{j+1} then both hold elections for two new leaders. If the new elected leaders have no edge between them, then we repeatedly elect two new leaders until the two leaders are connected.

References

[1] Baruch Awerbuch and Christian Scheideler. Towards a scalable and robust dht. *Theory Comput. Syst.*, 45(2):234–260, 2009.

[2] I. Boman, J. Saia, C. Abdallah, and E. Schamiloglu. Brief announcement: Self-healing algorithms for reconfigurable networks. *Stabilization, Safety, and Security of Distributed Systems*, pages 563–565, 2006.

[3] M. Datar. Butterflies and peer-to-peer networks. *AlgorithmsESA 2002*, pages 201–222, 2002.

[4] Amos Fiat and Jared Saia. Censorship resistant peer-to-peer content addressable networks. In *Proceedings of the Thirteenth ACM Symposium on Discrete Algorithms (SODA)*, 2002.

[5] Amos Fiat and Jared Saia. Censorship resistant peer-to-peer networks. *Theory of Computing*, 3(1):1–23, 2007.

[6] Amos Fiat, Jared Saia, and Maxwell Young. Making Chord Robust to Byzantine Attacks. In *Proceedings of the European Symposium on Algorithms(ESA)*, 2005.

[7] T. Hayes, N. Rustagi, J. Saia, and A. Trehan. The forgiving tree: a self-healing distributed data structure. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 203–212. ACM, 2008.

[8] T.P. Hayes, J. Saia, and A. Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 121–130. ACM, 2009.

[9] K. Hildrum and J. Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. *Distributed Computing*, pages 321–336, 2003.

[10] V. King, S. Lonargan, J. Saia, and A. Trehan. Load balanced scalable byzantine agreement through quorum building, with full information. *Distributed Computing and Networking*, pages 203–214, 2011.

[11] Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[12] G. Pandurangan and A. Trehan. Xheal: localized self-healing using expanders. In *Proceedings of the twenty-ninth ACM symposium on Principles of distributed computing*. ACM, 2011.

[13] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85. ACM, 1989.

[14] J. Saia and A. Trehan. Picking up the pieces: Self-healing in reconfigurable networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[15] J. Saia and M. Young. Reducing communication costs in robust peer-to-peer networks. *Information Processing Letters*, 106(4):152–158, 2008.

[16] A.D. Sarma and A. Trehan. Edge-preserving self-healing: keeping network backbones densely connected. *Arxiv preprint arXiv:1108.5893*, 2011.

[17] C. Scheideler. How to Spread Adversarial Nodes? Rotate! In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing(STOC)*, 2005.

[18] TOR Website. https://www.torproject.org/.

[19] M Young, A Kate, I Goldberg, and M Karsten. Practical robust communication in dhts tolerating a byzantine adversary. In *ICDCS*, pages 263–272, 2010.

A Appendix - Deferred Proofs

In this appendix, we provide proofs that were deferred from the main paper.

Lemma 1. Consider a sequence of x nodes, where each node in the sequence is bad independently with probability 1/4. Then the probability that there is any substring of length $\max(1, \log x)$ bad nodes in this sequence is no more than 1/2.

Proof. The probability of a specific substring of $\log x$ nodes being all bad is

$$\left(\frac{1}{4}\right)^{\log x} = \frac{1}{x^2}.$$

Union bounding over all possible substrings of length $\log x$, the probability of any all-bad substring existing is at most

$$x\frac{1}{x^2} = \frac{1}{x} \leq \frac{1}{2},$$

for $x \geq 2$. For $x = 1$, $\max(1, \log x) = 1$, and the probability of having a subsequence of 1 bad node in a sequence of 1 bad node is 1/4. □

Lemma 3. If some bad leader has corrupted a message in the last call to *SEND-LEADER*, then the algorithm *UPDATE* will 1) identify a pair of neighboring quorums Q_j and Q_{j+1}, for some $1 \leq j < \ell$ such that at least one of the two quorums currently has a bad leader; and 2) elect new leaders for these two quorums. Moreover, *UPDATE* will always remove at least one edge from the network, and at least one endpoint of each edge removed will be a bad node.

Proof. First, we show that if a pair of nodes x and y is in conflict, then at least one of them is bad. Assume not. Then both x and y are good. Then node x would have truthfully reported what it received; any message that x received would have been sent directly to y; and y would have truthfully reported what it received from x. But this is a contradiction, since for x and y to be in conflict, y must have reported that it received from x something different than what x reported receiving.

Now consider the case where a bad leader corrupted a message in the last call to *SEND-LEADER*. By the definition of corruption, there must be two good leaders q_j and q_k such that $j < k$ and q_j received the message m' sent by node $\mathbf{s}$, and q_k did not. We now show that some pair of leaders between q_j and q_k will be in conflict. Assume this is not the case. Then for all x, where $j \leq x \leq k-1$, leaders q_x and q_{x+1} are not in conflict. But then, since leader q_j received the message m', and there are no pairs of leaders in conflict, it must be the case that the leader q_k received the message m'. This is a contradiction. Thus, *UPDATE* will find two leaders that are in conflict, and at least one of them will be bad.

Now we prove that at least one pair of nodes is found to be in conflict as a result of triggering *UPDATE*. Assume that no pair of nodes is in conflict. Then for every pair of nodes x and y, such that x was scheduled to send a message to y during any round i of *CHECK*, x and y must have reported that they received the same message in round i. In particular, this implies via induction, that for every round i, for all j, where $1 \leq j \leq \ell$, all nodes in the sets S_j must have broadcasted that they received the message m' that was initially sent by node $\mathbf{s}$ in round i. But if this is the case, the node x that initially called *UPDATE* would have received no inconsistent messages. This is a contradiction since in such a case, node x would have been unsuccessful in trying to initiate a call to *UPDATE*. Thus, some pair of nodes must be found to be in conflict, and at least one of them is bad. □

Note that the above proof shows that, even if the node $\mathbf{s}$ is bad, a call to *UPDATE* will remove an edge between two nodes, at least one of which is bad. Thus, a bad $\mathbf{s}$ can only force calls to *UPDATE* a fixed number of times.

Lemma 4. Assume there are j quorums in the quorum graph that have bad leaders, for any positive integer j. Then, in expectation, the number of corruptions that must be caught by *CHECK* before the leaders of all quorums are good is no more than $2j$.

Proof. By Lemma 3, if a corruption occurred in *SEND-LEADER*, it is caught by *CHECK*, and *UPDATE* is called, then two neighboring quorums, Q_j, Q_{j+1}, for some $1 \leq j < \ell$, will be identified such that at least one leader of these

quorums are bad. Then, the current leaders of these quorums will be deposed and new leaders will be elected for both quorums.

We will model the properties of our quorum graph with a walk on a Markov chain with states $0, 1, \ldots, n_q$.[6] The walk will be at state i if exactly i quorums in the quorum graph have bad leaders. Note that state 0 is an absorbing state: if no quorums have bad leaders, there will no longer be any corruptions, and so there is no possibility of any good leaders being deposed.

When two new leaders are elected, let P_{bb} be equal to the probability that two bad leaders are elected; P_{bg} be equal to the probability that one bad and one good leader are elected; and P_{gg} be equal to the probability that two good leaders are elected. Note that $P_{bb} \leq 1/16$, and $P_{gg} \geq 9/16$.

Transitions occur whenever a corruption is caught by *CHECK* and *UPDATE* is called. If this happens, two leaders are deposed (one of which is guaranteed to be bad); and two new leaders are elected. We want to bound the expected number of corruptions that are detected until we reach state 0, which is equivalent to bounding the expected number of steps in the walk until we reach state 0.

We now give transition probabilities on the Markov chain to upper bound the expected number of corruptions until all leaders are good. When in any state i, $0 < i < n_q$, the walk transitions to state $i-1$, with probability 9/16; the walk transitions to state $i+1$ with probability 1/16; and the walk stays in state i with probability 6/16. If the walk is in state 0, it will stay there with probability 1.

Let $f(i)$ be the expected number of steps on this Markov chain to reach state 0, given that we are currently in state i. Note that $f(i)$ is an upper bound on the expected number of pairs of leaders that must be deposed before all leaders are good, given that there are i bad leaders currently.

Solving for f is a simple variant of the "gamblers ruin" problem. We include it here for completeness. We have the following equations for f. $f(0) = 0$; and for all i, $1 \leq i < n_q$

$$f(i) = 1 + \frac{9}{16}f(i-1) + \frac{6}{16}f(i) + \frac{1}{16}f(i+1)$$

Rewriting this equation, we have for all i, $1 \leq i < n_q$

$$10f(i) = 16 + 9f(i-1) + f(i+1).$$

For any j, where $1 \leq j \leq n_q$, if we sum the above equations over all i, $1 \leq i \leq j-1$, we obtain:

$$\begin{aligned} f(j) &= 9f(j-1) + f(1) - 16(j-1) - 9f(0) \\ &= 9f(j-1) + f(1) - 16(j-1) \end{aligned}$$

We now prove that for all j, where $0 \leq j \leq n_q$, that $f(j) \leq 2j$. The base case, $f(0) = 0$, is trivially true. For the inductive step, we have

$$\begin{aligned} f(j) &= 9f(j-1) + f(1) - 16(j-1) \\ &\leq 18(j-1) + 2 - 16(j-1) \\ &= 2j \quad \square \end{aligned}$$

[6] recall $n_q = \theta(n)$ is the number of quorums in the quorum graph