Self-Healing of Byzantine Faults*

Jeffrey Knockel, George Saad, and Jared Saia

Department of Computer Science, University of New Mexico
{jeffk,george.saad,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 attacks. 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 $t \leq (1/8 - \epsilon)n$ nodes, for any non-negative ϵ. First, the network provides a point-to-point communication with bandwidth and latency costs that are asymptotically optimal. Second, the expected total number of message corruptions is $O(t(\log^* n)^2)$ before the adversarially controlled nodes are effectively quarantined so that they cause no more corruptions. Empirical results show that our algorithm can reduce bandwidth cost by up to a factor of 70.

Keywords: Byzantine Faults, Threshold Cryptography, Self-Healing Algorithms.

"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 [1–6]. 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 $1/8$ 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.

* This research is partially supported by NSF grants: CISE-1117985 and CNS-1017509.

T. Higashino et al. (Eds.): SSS 2013, LNCS 8255, pp. 98–112, 2013.

Moreover, our algorithms limit the expected total 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] Now 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. The nodes that are compromised by the adversary are *bad*, and the other nodes are *good*. The bad nodes may arbitrarily deviate from the protocol, by sending no messages, excessive numbers of messages, incorrect messages, or any combination 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 h time steps to be sent and received, and the value h is known to all nodes. Also, we allow for the adversary to be *rushing*, where 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 $1/8$-fraction are bad. Many results show how to create and maintain a network of quorums [7–13]. 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. Moreover, we assume that for any two nodes x and y in a quorum, node x knows all quorums that node y is in.

The 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*, $Q_1, Q_2, \ldots, Q_\ell$, through the quorum graph. 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 each node in Q_{i+1} to do majority filtering on the messages received in order to determine the true value of m. Then all nodes in Q_ℓ send m to node $\mathbf{r}$ that does a majority filter on the received messages. Unfortunately, this algorithm requires $O(\ell \log^2 n)$ messages and a latency of $O(\ell)$. This paper shows how to reduce the message cost to $O(\ell + \log n)$, in an amortized sense.

[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. $\log^* 10^{10} = 5$.

As we show in Section 4, this reduction can be large in practice. In particular, we reduce the bandwidth cost by a factor of 58 for $n = 14{,}116$, and by a factor of 70 for $n = 30{,}509$.

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. *Assume we have a network with n nodes and $t \leq (1/8 - \epsilon)n$ bad nodes, for any non-negative ϵ, and a quorum graph as described above. Then our algorithm ensures the following.*

- *For any call to SEND, the expected number of messages is $O(\ell + \log n)$ and the expected latency is $O(\ell)$, in an amortized sense.*[2]
- *The expected total number of times that SEND fails to deliver a message reliably is at most $3t(\log^* n)^2$.*

Due to the space constraints, the proof of this theorem is not given here [3].

1.3 Related Work

Several papers [14–18] have discussed different restoration mechanisms to preserve network performance by adding capacity and rerouting traffic streams in the presence of node or link failures. They present mathematical models to determine global optimal restoration paths, and provide methods for capacity optimization of path-restorable networks.

Our results are inspired by recent work on self-healing algorithms [1–6]. 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 [1–3]; the shortest path between any pair of nodes does not increase by much [4]; and expansion properties of the network are approximately preserved [5].

Our results are also similar in spirit to those of Saia and Young [19] and Young et al. [20], which both show how to reduce message complexity when transmitting a message across a quorum path of length ℓ. The first result, [19], achieves expected message complexity of $O(\ell \log n)$ by use of bipartite expanders. However, this result is impractical due to high hidden constants and high setup costs. The second result, [20],

[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 $O(\mathcal{L} + t \cdot (\ell_M \log^2 n + \log^5 n))$. Note that, since t is fixed, for large $\mathcal{L}$, the expected total number of messages is $O(\mathcal{L})$.

[3] A full version with all the proofs is available at the authors' homepages.

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.

As mentioned earlier, several peer-to-peer networks have been described that provably enable reliable communication, even in the face of adversarial attack [7–10, 12, 21]. To the best of our knowledge, our approach applies to each of these networks, with the exception of [21]. In particular, we can apply our algorithms to asymptotically improve the efficiency of the peer-to-peer networks from [7–10, 12].

Similarly to Young et al. [22], we use threshold cryptography as an alternative to Byzantine Agreement.

1.4 Organization of Paper

The rest of this paper is organized as follows. In Section 2, we describe our algorithms. The analysis of our algorithms is shown in Section 3. Section 4 gives empirical results showing how our algorithms can improve the efficiency of the butterfly networks of [7]. Finally, we conclude and describe problems for future work in Section 5.

2 Our Algorithms

In this section, we describe our algorithms: *BROADCAST*, *SEND*, *SEND-PATH*, *CHECK* and *UPDATE*.

2.1 Overview

Recall 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 $\mathbf{r}$. We assume that Q_1 is the leftmost quorum and Q_ℓ is the rightmost quorum; and we let $|Q_j|$ be the number of nodes in quorum Q_j, for $1 \leq j \leq \ell$.

The objective of our algorithms is marking all bad nodes in the network, where no more message corruption occurred. In order to do that, we mark nodes after they are *in conflict*, where a pair of nodes is *in conflict* if at least one of these nodes accuses the other node in this pair. If the half of nodes in any quorum are marked, we unmark these nodes. Note that when we mark (or unmark) a node, it is marked (or unmarked) in its quorums and in their neighboring quorums. Note that all nodes in the network are initially unmarked.

In our algorithms, we assume that when any node x in a quorum Q broadcasts a message m to a set of nodes S, it executes $BROADCAST(m, Q, S)$. Before discussing our main algorithm, *SEND*, we describe *BROADCAST* procedure (Algorithm 1).

In *BROADCAST*, we make use of the threshold cryptography as an alternative to Byzantine Agreement. We briefly describe the threshold cryptography, (η, d)-threshold scheme.

Threshold Cryptography. In (η, d)-threshold scheme, the secret key is distributed among η parties, and any subset of more than d parties can jointly reassemble the key. The secret key can be distributed by a completely distributed approach, *Distributed Key*

Algorithm 1. $BROADCAST(m, Q, S)$

Declarations: *BROADCAST* is being called by a node x in a quorum Q in order to send a message m to a set of nodes S.

1. Node x sends message m to all nodes in Q.
2. Each node in Q signs m by its private key share to obtain a signed-message share, and sends this signed-message share back to node x.
3. Node x interpolates at least $\frac{7|Q|}{8}$ of the received signed-message shares to obtain the signed-message of Q.
4. Node x sends the signed-message of Q to all nodes in S.

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

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

1. Node $\mathbf{s}$ calls *SEND-PATH* $(m, \mathbf{r})$.
2. With probability p_{call}, node $\mathbf{s}$ calls *CHECK* $(m, \mathbf{r})$.

Generation (DKG) [23]. The *Distributed Key Generation* (DKG) generates the public/private key shares of all nodes in every quorum and the public/private key pair of each quorum. The public key share of any node is known only to the nodes that are in the same quorum. Moreover, the public key of each quorum is known to all nodes of this quorum and its neighboring quorums, but the private key of any quorum is unknown to all nodes.

BROADCAST Algorithm. In $BROADCAST(m, Q, S)$, we use in particular a $(|Q|, \frac{7}{8}|Q| - 1)$-threshold scheme, where $|Q|$ is the quorum size. Throughout the paper, when any node x in a quorum Q broadcasts a message m to a set of nodes S, it calls $BROADCAST(m, Q, S)$[4]. In $BROADCAST(m, Q, S)$, node x sends m to all nodes in Q. Then each node in the quorum signs m by its private key share, and sends the signed message share to node x. Node x interpolates at least $\frac{7|Q|}{8}$ of the received signed-message shares to construct a signed-message of the quorum. We know that at least $7/8$-fraction of the nodes in any quorum are good. So if this signed-message is constructed, it ensures that at least $7/8$-fraction of the nodes in this quorum has received the same message m, agrees upon the content of the message and gives the permission to node x of broadcasting this message. Then node x sends this constructed signed-message to all nodes in S.

Now we describe our main algorithm, *SEND*, that is stated formally in Algorithm 2. *SEND* calls *SEND-PATH*, which is formally described in Algorithm 3. In *SEND-PATH*, node $\mathbf{s}$ sends message m to node $\mathbf{r}$ through a path of unmarked nodes selected uniformly at random. However, *SEND-PATH* algorithm is vulnerable to corruption. Thus, with probability p_{call}, *SEND* calls *CHECK*, which detects if a message was corrupted in the previous call to *SEND-PATH*, with probability p_{detect}.

[4] When node $\mathbf{s}$ broadcasts m to a set of nodes S, it calls $BROADCAST(m, Q_1, S)$.

Algorithm 3. SEND-PATH$(m, \mathbf{r})$

Declarations: m is the message to be sent, and $\mathbf{r}$ is the destination.

1. Node $\mathbf{s}$ selects an unmarked node, q_1, in Q_1 uniformly at random.
2. Node $\mathbf{s}$ broadcasts to all nodes in Q_1 the message m and q_1's ID.
3. All nodes in Q_1 forward the message to node q_1.
4. For $i = 1, \ldots, \ell - 1$ do
   (a) Node q_i selects an unmarked node, q_{i+1}, in Q_{i+1} uniformly at random.
   (b) Node q_i sends m to node q_{i+1}.
5. Node q_ℓ in Q_ℓ broadcasts m to all nodes in Q_ℓ.
6. All nodes in Q_ℓ send m to node $\mathbf{r}$.

In *CHECK*, the message is propagated from the leftmost quorum to the rightmost quorum in the quorum path through a path of subquorums, where a subquorum is a subset of unmarked nodes selected uniformly at random in a quorum.

CHECK is implemented as either *CHECK1* (Algorithm 4) or *CHECK2* (Algorithm 5). For *CHECK1*, $p_{call} = 1/(\log\log n)^2$ and $p_{detect} = 1 - o(1)$, and it requires $O(\ell(\log\log n)^2 + \log n \cdot \log\log n)$ messages and has latency $O(\ell)$. For *CHECK2*, $p_{call} = 1/(\log^* n)^2$ and $p_{detect} \geq 1/2$, and it has message cost $O((\ell + \log n)(\log^* n)^2)$ and latency $O(\ell \log^* n)$.

Unfortunately, while *CHECK* can determine if a corruption occurred, it does not specify the location where the corruption occurred. Thus, if *CHECK* detects a corruption, *UPDATE* (Algorithm 6) is called. When *UPDATE* is called, it identifies two neighboring quorums Q_i and Q_{i+1} in the path, for some $1 \leq i < \ell$, such that at least one pair of nodes in these quorums is in conflict and at least one node in such pair is bad. Then quorums Q_i and Q_{i+1} mark these nodes and notify all other quorums that these nodes are in. All quorums in which these nodes are notify their neighboring quorums. Recall that if the half of nodes in any quorum have been marked, these nodes are set unmarked in all quorums they are in, and their neighboring quorums are notified.

Moreover, we use *BROADCAST* in *SEND-PATH* and *CHECK* to handle any accusation in *UPDATE* against node $\mathbf{s}$ or node $\mathbf{r}$. In *SEND-PATH* (or *CHECK*), we make node $\mathbf{s}$ broadcast the message to all nodes in Q_1 that forward the message to the selected unmarked node (or subquorum) in Q_1; and when the message is propagated to the selected unmarked node (or subquorum) in Q_ℓ, the message is broadcasted to all nodes in Q_ℓ.

Our model does not directly consider concurrency. In a real system, concurrent calls to UPDATE that overlap at a single quorum may allow the adversary to achieve multiple corruptions at the cost of a single marked bad node. However, this does not effect correctness, and, in practice, this issue can be avoided by serializing concurrent calls to SEND. For simplicity of presentation, we leave the concurrency aspect out of this paper.

Throughout this paper, we let U_i be the set of unmarked nodes in Q_i, and we let $|U_i|$ be the number of nodes in U_i, for $1 \leq i \leq \ell$.

2.2 CHECK1

Algorithm 4. CHECK1$(m, \mathbf{r})$

Initialization: let the subquorums $S_1, S_2, ..., S_\ell$ be initially empty.

1. Node $\mathbf{s}$ constructs R to be an ℓ by $|Q_{max}|$ by $\log\log n$ array of random numbers, where $|Q_{max}|$ is the maximum size of any quorum and $\log\log n$ is the size of any subquorum. Note that $R[j, k]$ is a multiset of $\log\log n$ numbers selected uniformly at random with replacement between 1 and k.
2. Node $\mathbf{s}$ sets m' to be a message consisting of m, $\mathbf{r}$, and R.
3. Node $\mathbf{s}$ broadcasts m' to all nodes in Q_1.
4. The nodes in Q_1 calculate the nodes of S_1 using the numbers in $R[1, |U_1|]$ to index U_1's nodes sorted by their IDs.
5. The nodes in Q_1 send m' to the nodes of S_1.
6. For $j \leftarrow 1, \ldots, \ell - 1$ do
   (a) The nodes of S_j calculate the nodes of S_{j+1} using the numbers in $R[j + 1, |U_{j+1}|]$ to index U_{j+1}'s nodes sorted by their IDs.
   (b) The nodes of S_j send m' to all nodes of S_{j+1}.
7. The nodes of S_ℓ broadcast m' to all nodes in Q_ℓ.

Note that: throughout *CHECK1*, if a node receives inconsistent messages or fails to receive an expected message, then it initiates a call to *UPDATE*.

Now we describe *CHECK1* that is stated formally as Algorithm 4. *CHECK1* is a simpler *CHECK* procedure compared to *CHECK2*, and, although it has a worse asymptotic message cost, it performs well in practice.

In *CHECK1*, each subquorum S_i has $\log\log n$ nodes that are chosen uniformly at random with replacement from the nodes in U_i in the quorum path, for $1 \leq i \leq \ell$.

First, node $\mathbf{s}$ broadcasts the message to all nodes in Q_1, which send the message to the nodes of S_1. Then the nodes of S_1 forward this message to the nodes of S_ℓ through a path of subquorums in the quorum path via all-to-all communication. Once the nodes in S_ℓ receive the message, they broadcast this message to all nodes in Q_ℓ. Further, if any node receives inconsistent messages or fails to receive an expected message, it initiates a call to *UPDATE*.

Now we show that if the message was corrupted during the last call to *SEND-PATH*, the probability that *CHECK1* fails to detect a corruption is $o(1)$ when $\ell = O(\log n)$.

Lemma 1. *If $\ell = O(\log n)$, then CHECK1 fails to detect any message corruption with probability $o(1)$.*

Proof. *CHECK1* succeeds in detecting the message corruption if every subquorum has at least one good node. We know that at least $1/2$-fraction of the nodes in any quorum are unmarked, then the probability that an unmarked bad node is selected uniformly at random is at most $1/4$. Thus 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 *CHECK1* failing is at most $\ell/\log^2 n$. For $\ell = O(\log n)$, the probability that *CHECK1* fails is $o(1)$. □

2.3 CHECK2

Algorithm 5. CHECK2$(m, \mathbf{r})$

Initializations: node $\mathbf{s}$ generates public/private key pair k_p, k_s to be used in this procedure; also let the subquorums $S_1, S_2, ..., S_\ell$ be initially empty.

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

1. Node $\mathbf{s}$ constructs R to be an ℓ by $|Q_{max}|$ array of random numbers, where $|Q_{max}|$ is the maximum size of any quorum. Note that $R[j,k]$ is a uniformly random number between 1 and k.
2. Node $\mathbf{s}$ sets m' to be m, k_p, $\mathbf{r}$, and R signed by k_s.
3. Node $\mathbf{s}$ broadcasts m' to all nodes in Q_1.
4. The nodes in Q_1 calculate the node, $x_1 \in U_1$, to be added to S_1 using the number $R[1, |U_1|]$ to index U_1's nodes sorted by their IDs.
5. The nodes in Q_1 send m' to the nodes of S_1.
6. For $j \leftarrow 1, \ldots, \ell - 1$ do
   (a) All i nodes in S_j calculate the node, $x_{j+1} \in U_{j+1}$, to be added to S_{j+1} using the number $R[j+1, |U_{j+1}|]$ to index U_{j+1}'s nodes sorted by their IDs.
   (b) The nodes in S_j send m' to node x_{j+1}.
   (c) Node x_{j+1} sends m' to all the nodes in S_{j+1}.
7. The nodes in S_ℓ broadcast m' to all nodes in Q_ℓ.

end for

Note that: throughout this procedure, if a node has previously received k_p, then it verifies each subsequent message with it; also if a node receives inconsistent messages or fails to receive and verify an expected message, then it initiates a call to *UPDATE*.

In this section, we describe *CHECK2* algorithm, which is stated formally as Algorithm 5. Firstly, node $\mathbf{s}$ generates a public/private key pair k_p, k_s to let the nodes verify any message received. Then *CHECK2* runs for $4\log^* n$ rounds, and has a subset $S_j \subset U_j$ for each quorum Q_j in the quorum path, for $1 \leq j \leq \ell$. Every S_j is an incremental subquorum, where each S_j is initially empty; and in each round, an unmarked node, x_j, is selected uniformly at random from all nodes in U_j and is added to S_j, for all $1 \leq j \leq \ell$.

In each round, node $\mathbf{s}$ broadcasts the message to all nodes in Q_1, which forward such message to the nodes of S_1. Then for all $1 \leq j < \ell$, all nodes in S_j send the message to node x_{j+1}, which forwards the message to all nodes in S_{j+1}. Finally, all nodes in S_ℓ broadcast the message to all nodes in Q_ℓ.

Note that if any node receives inconsistent messages or fails to receive and verify any expected message in any round, it initiates a call to *UPDATE*.

An example run of *CHECK2* 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 *CHECK2*. 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).

Recall that the adversary knows *CHECK2* algorithm, but in any round, the adversary does not know which nodes will be selected to participate for any subsequent round. The adversary's strategy is to maintain an interval of bad nodes in each row to corrupt (or drop) the message so that *CHECK2* will not be able to detect the corruption.

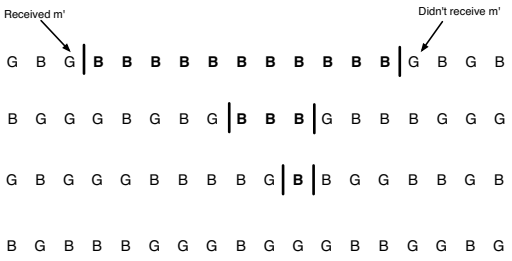

Fig. 1. Example run of *CHECK2*

In Figure 1, the intervals maintained by the adversary are outlined by a left and a right bar in each row; where the best interest of the adversary is to maintain the longest interval of bad nodes in the first row. The left bar in each row specifies the rightmost subquorum in which there is some good node that receives m'. The right bar in each row specifies the leftmost subquorum in which there is some good node that does not receive m'.

Note that, as rounds progress, the left bar can never move leftwards, because a node that has already received k_p will call *UPDATE* unless it receives messages signed with k_p for all subsequent rounds. Note further that the right bar can never move rightwards, since the nodes of each subset S_j send m' to the node x_{j+1}, which forwards such message to all nodes that are currently in S_{j+1}, for $1 \leq j < \ell$. Finally, when these two bars meet, a corruption is detected.

Intuitively, the reason *CHECK2* 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 at most $1/4$, and value "G" with probability at least $3/4$, 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 2 and 3 of Section 3.

In contrast to *CHECK1*, even though all nodes in column 9 are bad in Figure 1, *CHECK2* algorithm handles this case since the good node in row 3 and column 10 receives the message m'.

2.4 UPDATE

When a message corruption occurs and *CHECK* detects this corruption, *UPDATE* is called. Now the task of *UPDATE* is to 1) determine the location in which the corruption occurred; 2) mark the nodes that are in conflict. The *UPDATE* algorithm is described formally as Algorithm 6. When *UPDATE* starts, all nodes in each quorum in the quorum path are notified.

To determine the location in which the corruption occurred, each node previously involved in *SEND-PATH* or *CHECK* broadcasts to all nodes in its quorum and the

Algorithm 6. *UPDATE*

1. The node x in a quorum Q' making the call to *UPDATE* broadcasts this fact to all nodes in Q' along with all the messages that x has received during this call to *SEND*. The nodes in Q' verify 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. Each node involved in the last call to *SEND-PATH* or *CHECK*, except node $\mathbf{s}$ and node $\mathbf{r}$, compiles all messages they have received (and from whom) in that last call, and broadcasts all these messages to all nodes in its quorum and the neighboring quorums.
4. Each node involved in the last call to *SEND-PATH*, except node $\mathbf{s}$ and node $\mathbf{r}$, broadcasts all messages they have sent (and to whom) in that last call, to all nodes in its quorum and the neighboring quorums.
5. A node v is *in conflict* with a node u if:
   (a) node u was scheduled to send a message to node v at some point in the last call to *SEND-PATH* or *CHECK*; and
   (b) node v does not receive an expected message from node u in step 3 or step 4, or node v receives a message in step 3 or step 4 that is different than the message that it has received from node u in the last call to *SEND-PATH* or *CHECK*.
6. For each pair of nodes, (u, v), that are in conflict in (Q_k, Q_{k+1}), for $1 \leq k < \ell$:
   (a) node v sends a *conflict* message "(u,v)" to all nodes in Q_v,
   (b) each node in Q_v forwards this conflict message to all nodes in Q_v and all nodes in Q_u,
   (c) quorum Q_u (or Q_v) sends the conflict message to all other quorums that node u (or v) is in, and
   (d) each quorum that node u or node v is in sends such conflict message to its neighboring quorums.
7. Each node that receives this conflict message mark the nodes u and v.
8. If the half of nodes in any quorum have been marked, they are set unmarked in all quorums these nodes are in, and their neighboring quorums are notified.

neighboring quorums all messages they have received in the previous call to *SEND-PATH* or *CHECK*. Moreover, to announce the unmarked nodes that are selected uniformly at random in the last call to *SEND-PATH*, every node involved in the last call to *SEND-PATH*, broadcasts all messages they have sent (and to whom) in such call, to all nodes in its quorum and the neighboring quorums.

We say that a node v is *in conflict* with a node u if 1) node u was scheduled to send a message to node v at some point in the previous call to *SEND-PATH* or *CHECK*; and 2) node v does not receive an expected message from node u in this call to *UPDATE*, or node v receives a message in this *UPDATE* that is different than the message that it has received from node u in the previous call to *SEND-PATH* or *CHECK*.

For each pair of nodes, (u, v), that are in conflict in quorums (Q_k, Q_{k+1}) for $1 \leq k < \ell$, these two quorums send a *conflict* message "(u,v)" to all quorums in which node u or node v is and to all neighboring quorums to mark these nodes. Recall that if the half of nodes in any quorum are marked, this quorum notifies all other quorums in which these nodes are via all-to-all communication to unmark such nodes, and their neighboring quorums are notified as well.

3 Analysis

In this section, we prove the lemmas required for Theorem 1. Throughout this section, *SEND-PATH* calls *CHECK2*. Also we let all logarithms be base 2.

Lemma 2. *When a coin is tossed x times independently given that each coin has a tail with probability at most $1/4$, then the probability of having any substring of length $\max(1, \log x)$ being all tails is at most $1/2$.*

Proof. The probability of a specific substring of length $\log x$ being all tails is

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

Union bounding over all possible substrings of length $\log x$, then the probability of any all-tailed substring existing is at most $x\frac{1}{x^2}$; or equivalently, for $x \geq 2$, $\frac{1}{x} \leq \frac{1}{2}$; and for $x = 1$, the probability of having a substring of $\max(1, \log x)$ tail is trivially $1/4$. □

The next lemma shows that the algorithm *CHECK2* catches corruptions with probability at least $1/2$.

Lemma 3. *Assume some node selected uniformly at random in the last call to SEND-PATH has corrupted a message. Then when the algorithm CHECK2 is called, with probability at least $1/2$, some node will call UPDATE.*

Lemma 4. *If some node selected uniformly at random in the last call to SEND-PATH has corrupted a message, then the algorithm UPDATE will identify a pair of neighboring quorums Q_j and Q_{j+1}, for some $1 \leq j < \ell$, such that at least one pair of nodes in these quorums is in conflict and at least one node in such pair is bad.*

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

CHECK2, 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. □

The next lemma bounds the number of times that *UPDATE* must be called before all bad nodes are marked.

Lemma 5. *UPDATE is called at most* $3t/2$ *times before all bad nodes are marked.*

Proof. By Lemma 4, if a message corruption occurred in the last call to *SEND-PATH*, and is caught by *CHECK2*, then *UPDATE* is called. *UPDATE* identifies at least one pair of nodes that are in conflict.

Now let g be the number of good nodes that are marked, and let b be the number of bad nodes that are marked. Also let $f(g, b) = b - g/3$.

For each corruption caught, at least one bad node is marked, and so $f(g, b)$ increases by at least $2/3$ since b increases by at least 1 and g increases by at most 1. When a $1/2$-fraction of nodes in any quorum Q of size $|Q|$ are unmarked, $f(g, b)$ further increases by at least 0 since g decreases by at least $\frac{3|Q|}{8}$ and b decreases by at most $\frac{|Q|}{8}$.

Hence, $f(g, b)$ is monotonically increasing by at least $2/3$ for each corruption caught. When all bad nodes are marked, $f(g, b) \leq t$. Therefore, after at most $3t/2$ calls to *UPDATE*, all bad nodes are marked. □

4 Empirical Results

4.1 Setup

In this section, we empirically compare the message cost and the fraction of messages corrupted of two algorithms via simulation. The first algorithm we simulate is *no-self-healing* algorithm from [24]. 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 *self-healing*, wherein we apply our self-healing algorithm in the butterfly networks using *CHECK1*.

In our experiments, we consider two butterfly network sizes: $n = 14{,}116$ and $n = 30{,}509$, where $\ell = \lfloor \log n \rfloor - 2$, the quorum size is $\lfloor 4 \log n \rfloor$ and the subquorum size is $\lfloor \log \log n \rfloor$. Moreover, we do our experiments for several fractions of bad nodes such as f equal to $1/8, 1/16, 1/32$ and $1/64$, where $f = t/n$.

Our simulations consist of a sequence of calls to *SEND* over the network, given a pair of nodes $\mathbf{s}, \mathbf{r}$, chosen uniformly at random, such that node $\mathbf{s}$ sends a message to node $\mathbf{r}$. 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.

4.2 Results

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

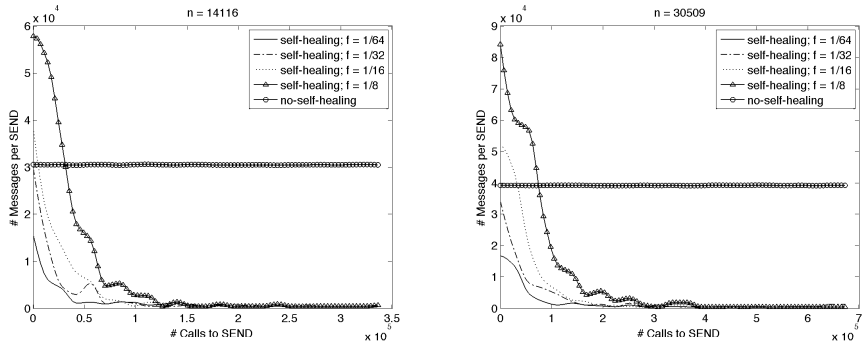

Fig. 2. # Messages per *SEND* versus # calls to *SEND*, for $n = 14{,}116$ and $n = 30{,}509$

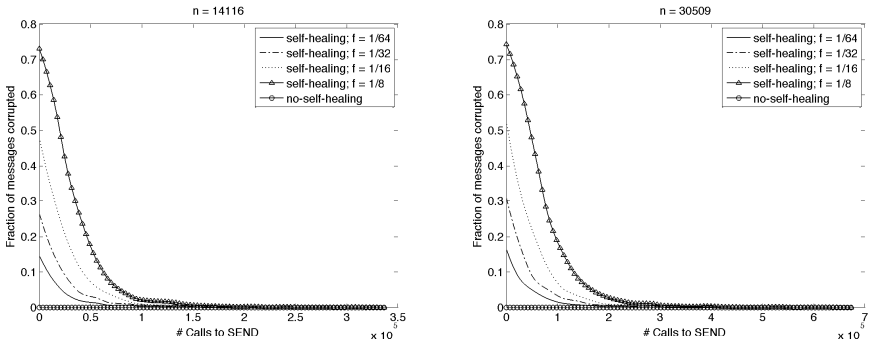

Fig. 3. Fraction of messages corrupted versus # calls to *SEND*, for $n = 14{,}116$ and $n = 30{,}509$

Figure 2 shows that for $n = 14{,}116$, the number of messages per SEND for *no-self-healing* is $30{,}516$; and for *self-healing*, it is 525. Hence, the message cost is reduced by a factor of 58. Also for $n = 30{,}509$, the number of messages per SEND for *no-self-healing* is $39{,}170$; and for *self-healing*, it is 562; which implies that the message cost is reduced by a factor of 70.

In Figure 3, *no-self-healing* has 0 corruptions; however, for *self-healing*, the fraction of messages corrupted per *SEND* decreases as the total number of calls to *SEND* increases. Also for a fixed number of calls to *SEND*, the fraction of messages corrupted per *SEND* decreases as the total number of bad nodes decreases.

Furthermore, in Figure 3, for each network, given the size and the fraction of bad nodes, if we integrate the corresponding curve, we get the total number of times that a message can be corrupted in calls to *SEND* in this network. These experiments show that the total number of message corruptions is at most $3t(\log\log n)^2$.

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 $t \leq n/8$ bad nodes in the network, our algorithm allows $O(t(\log^* n)^2)$ message transmissions to be corrupted in expectation.

Many problems remain. First, it seems unlikely that the smallest number of corruptions allowable by an attack-resistant algorithm with optimal message complexity is $O(t(\log^* n)^2)$. Can we improve this 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?

References

1. Boman, I., Saia, J., Abdallah, C.T., Schamiloglu, E.: Brief announcement: Self-healing algorithms for reconfigurable networks. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 563–565. Springer, Heidelberg (2006)
2. Saia, J., Trehan, A.: Picking up the pieces: Self-healing in reconfigurable networks. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–12 (2008)
3. Hayes, T., Rustagi, N., Saia, J., Trehan, A.: The forgiving tree: a self-healing distributed data structure. In: PODC 2008, pp. 203–212 (2008)
4. Hayes, T.P., Saia, J., Trehan, A.: The forgiving graph: a distributed data structure for low stretch under adversarial attack. In: PODC 2009, pp. 121–130 (2009)
5. Pandurangan, G., Trehan, A.: Xheal: localized self-healing using expanders. In: PODC 2011, pp. 301–310 (2011)
6. Das Sarma, A., Trehan, A.: Edge-preserving self-healing: keeping network backbones densely connected. In: 2012 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 226–231 (2012)
7. Fiat, A., Saia, J.: Censorship resistant peer-to-peer networks. Theory of Computing 3(1), 1–23 (2007)
8. Hildrum, K., Kubiatowicz, J.D.: Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 321–336. Springer, Heidelberg (2003)

9. Naor, M., Wieder, U.: A simple fault tolerant distributed hash table. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 88–97. Springer, Heidelberg (2003)
10. Scheideler, C.: How to spread adversarial nodes? rotate! In: STOC 2005 (2005) 704–713
11. Fiat, A., Saia, J., Young, M.: Making chord robust to byzantine attacks. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 803–814. Springer, Heidelberg (2005)
12. Awerbuch, B., Scheideler, C.: Towards a scalable and robust dht. Theory of Computing Systems 45(2), 234–260 (2009)
13. King, V., Lonargan, S., Saia, J., Trehan, A.: Load balanced scalable byzantine agreement through quorum building, with full information. In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 203–214. Springer, Heidelberg (2011)
14. Frisanco, T.: Optimal spare capacity design for various protection switching methods in atm networks. In: ICC 1997 Montreal, vol. 1, pp. 293–298 (1997)
15. Iraschko, R., MacGregor, M., Grover, W.: Optimal capacity placement for path restoration in stm or atm mesh-survivable networks. IEEE/ACM Transactions on Networking 6(3), 325–336 (1998)
16. Murakami, K., Kim, H.: Comparative study on restoration schemes of survivable atm networks. In: INFOCOM 1997, vol. 1, pp. 345–352 (1997)
17. Van Caenegem, B., Wauters, N., Demeester, P.: Spare capacity assignment for different restoration strategies in mesh survivable networks. In: Communications, ICC 1997 Montreal, vol. 1, pp. 288–292 (1997)
18. Xiong, Y., Mason, L.: Restoration strategies and spare capacity requirements in self-healing atm networks. IEEE/ACM Transactions on Networking 7(1), 98–110 (1999)
19. Saia, J., Young, M.: Reducing communication costs in robust peer-to-peer networks. Information Processing Letters 106(4), 152–158 (2008)
20. Young, M., Kate, A., Goldberg, I., Karsten, M.: Practical robust communication in dhts tolerating a byzantine adversary. In: ICDCS 2010, pp. 263–272 (2010)
21. Datar, M.: Butterflies and peer-to-peer networks. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 310–322. Springer, Heidelberg (2002)
22. Young, M., Kate, A., Goldberg, I., Karsten, M.: Towards practical communication in byzantine-resistant dhts. IEEE/ACM Transactions on Networking 21(1), 190–203 (2013)
23. Kate, A., Goldberg, I.: Distributed key generation for the internet. In: ICDCS 2009, pp. 119–128 (2009)
24. Fiat, A., Saia, J.: Censorship resistant peer-to-peer content addressable networks. In: SODA 2002, pp. 94–103 (2002)