Noname manuscript No.
(will be inserted by the editor)

Self-Healing Communication

Jeffrey Knockel · George Saad · Jared Saia

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 $t \leq (1/4 - \epsilon)n$ nodes, for any constant $\epsilon > 0$. First, the network provides point-to-point communication with message and latency costs that are asymptotically optimal. Second, the expected total number of message corruptions is $O(t(\log^* n)^2)$, after which the adversarially controlled nodes are effectively quarantined so that they cause no more corruptions. Empirical results show that our algorithm can reduce message cost by up to a factor of 60 when compared with algorithms that are not self-healing.

Keywords Byzantine Faults, Threshold Cryptography, Quorum Graph, 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 [1,2,3,4,5,6]. 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 $1/4$ fraction of Byzantine faults. Our algorithms enable any node to send a message to any other node in the network with message and latency costs that are asymptotically optimal.

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] [2]

This paper is organized as follows. In Section 2, we describe our model. Our main theorem is given in Section 3, and we provide a technical overview in Section 4. The related work is discussed in Section 5. Section 6 describes our algorithms. The analysis of our algorithms is shown in Section 7. Section 8 gives empirical results showing how our algorithms improve the efficiency of the butterfly networks of [7]. Finally, we conclude and describe problems for future work in Section 9.

J. Knockel · G. Saad · J. Saia
Department of Computer Science, University of New Mexico
E-mail: {jeffk, saad, saia}@cs.unm.edu

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

[2] We thus amend our initial proverb to: *"Fool me once, shame on you. Fool me $\omega((\log^* n)^2)$ times, shame on me."*

Many of the results in this paper were first presented as an extended abstract in [8]. This paper is the full version of that extended abstract.

2 Our Model

We assume a *static* Byzantine adversary in the sense that it takes over nodes before the algorithm begins. The nodes that are controlled 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 in the sense that 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 tolerate if the adversary is *rushing*, where the bad nodes receive all messages from good nodes in a round before sending out their own messages.

3 Our Contributions

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

Theorem 1 *Assume we have a network with n nodes and $t \leq (1/4-\epsilon)n$ bad nodes, for any constant $\epsilon > 0$. Then our algorithm has the following properties:*

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

Our experimental results (Section 8) show that our algorithms reduce the message cost, compared to the naive algorithm, by a factor of 50 for $n = 14{,}116$, and by a factor of 60 for $n = 30{,}509$.

[3] In particular, if we call *SEND* $\mathcal{L}$ times through quorum paths, where ℓ_M is the longest such path, then the expected total number of messages sent will be $O(\mathcal{L}(\ell_M + \log n) + t \cdot (\ell_M \log^2 n + \log^5 n))$ with latency $O(\ell(\mathcal{L} + t))$. Since t is fixed for large $\mathcal{L}$, the expected number of messages per *SEND* is $O(\ell_M + \log n)$ with expected latency $O(\ell)$.

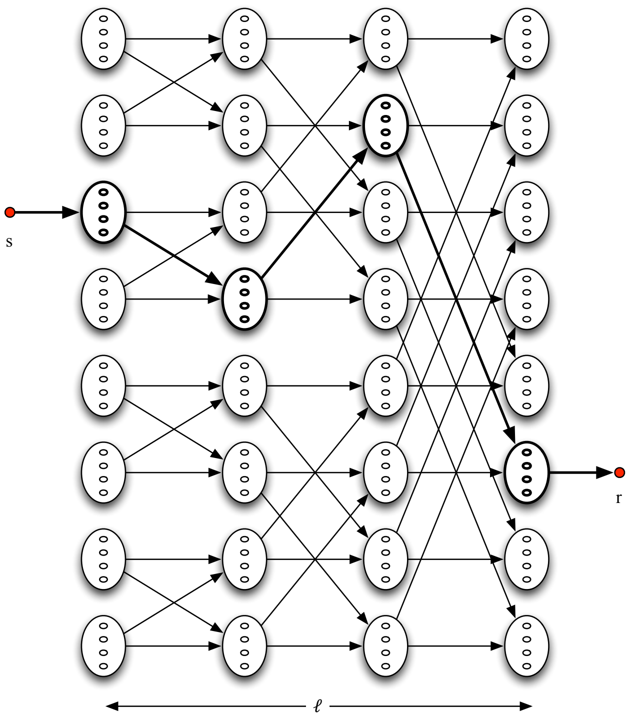


Fig. 1 Quorum Graph

4 Technical Overview

Our algorithms make critical use of quorums and a quorum graph.

4.1 Quorums and the Quorum Graph

We define a *quorum* to be a set of $\Theta(\log n)$ nodes, of which at most $1/4$-fraction are bad. Many results show how to create and maintain a network of quorums [7,9,10,11,12,13,14]. 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.

Note that the quorum graph is created after the adversary chooses the bad nodes.

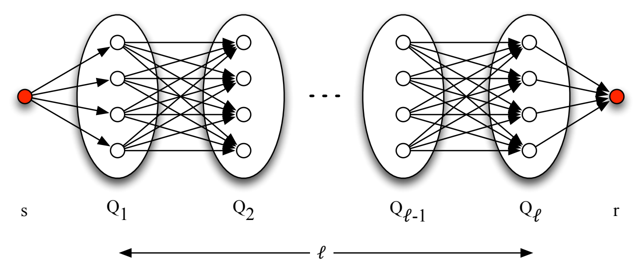


Fig. 2 Naive Algorithm

4.2 Communicating with Quorums

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 $\mathbf{r}$. Note that we assume that node $\mathbf{s}$ is a good node.

Figure 1 shows the communication between node $\mathbf{s}$ and node $\mathbf{r}$ through a quorum path in the quorum graph, where the message is propagated from the left to the right.

4.3 Naive Algorithm

A correct but inefficient algorithm to route a message m reliably from a source node to a target node is shown in Figure 2. In this algorithm, node $\mathbf{s}$ sends a message m to node $\mathbf{r}$ through a path of quorums via all-to-all communication. Each node participating in the quorum path takes the majority of the messages it receives in order to determine the true value of m.

Unfortunately, this algorithm requires $O(\ell \log^2 n)$ messages and latency $O(\ell)$. A main result of this paper is to reduce the message cost to $O(\ell + \log n)$ in an amortized sense.

4.4 Our Approach

An efficient approach to communication is to have a path of nodes selected uniformly at random through the quorum path from node $\mathbf{s}$ to node $\mathbf{r}$. In this path of nodes, each node forwards the message it has received to eventually be received by node $\mathbf{r}$. Unfortunately, a single bad node in this path can corrupt the entire communication.

We provide an algorithm, *CHECK*, which detects if there has been a corruption. *CHECK* is a light-weight algorithm in terms of message cost. Node $\mathbf{s}$ triggers *CHECK* with some probability. If *CHECK* detects a corruption, it calls *HEAL*, which is a heavy-weight algorithm but it is bounded in an amortized sense. In particular, the expected total number of calls to *HEAL* before all bad nodes are marked is $O(t)$.

CHECK is implemented as either *CHECK1* or *CHECK2*.

CHECK1 runs in one round. Node $\mathbf{s}$ resends the message through a path of subsets of $2 \log \log n$ nodes in the quorum path via all-to-all communication. *CHECK1* fails to detect corruptions if all nodes in any subset are bad. A key lemma (Lemma 1) shows that this algorithm fails to detect a corruption with probability less than $1/2$ for $\ell \leq \frac{\log^2 n}{2}$.

CHECK2 is a more sophisticated algorithm, which runs in $O(\log^* n)$ rounds. In each round, node $\mathbf{s}$ sends the message through a path of subsets of nodes in the quorum path. These subsets are initially empty; and in each round, a new node selected uniformly at random is added to each subset.

In order to show how *CHECK2* detects a corruption, we first define a *deception interval* in a round as a path of bad nodes that are selected in this round to be added to the subsets, and by which the adversary corrupts the message. Any corruption in the first round must occur in a deception interval in this round. Note that if the adversary corrupts a message in any round, it has to keep corrupting this message in all subsequent rounds. Thus, for i rounds of *CHECK2* to fail to detect a corruption, there must be nesting levels of deception intervals in each of those i rounds. A key lemma (Lemma 3) shows that any deception interval shrinks logarithmically from round to round with probability at least $1/2$. We use this lemma to show that *CHECK2* requires only $O(\log^* n)$ rounds to detect a corruption with constant probability.

In *HEAL*, each node has participated in the communication is investigated in order to determine which node(s) corrupted the message. In particular, each node announces the messages it has received. In this way, each call to *HEAL* identifies at least one pair of nodes that are in conflict, where informally, we say that a pair of nodes are in conflict if they each accuse the other of malicious behavior. In such a situation, we know that at least one node in the pair is bad. In *HEAL*, both nodes in each conflicting pair are marked, and these marked nodes are prohibited from participating in future communication.

A naive approach would be to never unmark marked nodes. Unfortunately, this approach fails because all nodes in a quorum may get marked and so inhabit communication.

To avoid this, we unmark the nodes of any quorum that has $1/2$-fraction of nodes marked. A subtle potential function argument (Lemma 7) shows that this marking scheme will mark all bad nodes after $O(t)$ calls to *HEAL*, after which the network is completely healed, i.e., no more corruption will occur.

5 Related Work

Several papers [15, 16, 17, 18, 19] 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 [1,2,3,4,5,6]. 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 [1,2,3]; 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 [20] and Young et al. [21], which both show how to reduce message complexity when transmitting a message across a quorum path of length ℓ. The first result, [20], 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, [21], 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 [7,22,9,10,11,13]. To the best of our knowledge, our approach applies to each of these networks, with the exception of [22]. In particular, we can apply our algorithms to asymptotically improve the efficiency of the peer-to-peer networks from [7,9,10,11,13].

Similar to Young et al. [23], we use threshold cryptography as an alternative to Byzantine Agreement.

The results of this paper were first presented in an extended abstract in [8].

6 Our Algorithms

In this section, we describe our algorithms: *SEND*, *SEND-PATH*, *CHECK* and *HEAL*. The main technical challenge of our paper is in the design of the algorithm *CHECK*, which is described in Section 6.4.

6.1 Overview

The objective of our algorithms is to detect the corruption and to mark all bad nodes in the network, after which no message corruption occurs. If a node is marked, it is not allowed to participate for communication. Before our algorithms start, all nodes in the network are initially unmarked.

Before discussing our main *SEND* algorithm, we describe that when a node x broadcasts a message m, signed by the private key of a quorum Q, to a set of nodes S, it calls $BROADCAST\ (m, Q, S)$.

6.2 *BROADCAST*

In *BROADCAST* (Algorithm 1), we use threshold cryptography to avoid the overhead of Byzantine Agreement.

In a (η, η')-threshold cryptographic scheme, a private key is distributed among η nodes in such a way that 1) any subset of more than η' nodes can jointly reassemble the key; and 2) no subset of at most η' nodes can recover the key. The private key can be distributed using a *Distributed Key Generation* (DKG) protocol [24].

DKG generates the public/private key shares of all nodes in every quorum. We assume that the public key share of each node and the public key of each quorum are known to all nodes in the quorum and the neighboring quorums in the network.

Algorithm 1 BROADCAST(m, Q, S) ▷ A node x sends a message m, signed by quorum Q, 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 its message share.
3: Each node in $Q$ sends its message share back to node $x$.
4: Node $x$ interpolates at least $\frac{3|Q|}{4}$ message shares to obtain a signed-message of $Q$.
5: Node $x$ sends this signed-message to all nodes in $S$.
```

In particular, we use a ($|Q|$, $\frac{3|Q|}{4} - 1$)-threshold scheme, where $|Q|$ is the quorum size. A node x calls $BROADCAST$ in order to send a message m to all nodes in S so that: 1) at least $3/4$-fraction of the nodes in quorum Q have received the same message m; 2) they agree upon the content of m; and 3) they give a permission to x to broadcast this message.

Any call to *BROADCAST* requires $O(\log n + |S|)$ messages for signing the message m by $O(\log n)$ nodes in quorum Q, with latency $O(1)$.

6.3 *SEND*

Now we describe our main algorithm, *SEND*, that is stated formally in Algorithm 2. *SEND* calls *SEND-PATH*, which is described in Algorithm 3.

In *SEND-PATH*, as shown in Figure 3, node **s** sends message m to node **r** through a path of unmarked nodes selected uniformly at random.

Algorithm 2 SEND$(m, \mathbf{r})$ ▷ node s sends a message m reliably to node $\mathbf{r}$.

```
1: Node s calls SEND-PATH $(m, \mathbf{r})$.
2: With probability $p_c$, node s calls CHECK $(m, \mathbf{r})$.
```

Algorithm 3 SEND-PATH$(m, \mathbf{r})$ ▷ node s sends a message m through a path of unmarked nodes to node $\mathbf{r}$.

Declaration: for $1 < i < \ell$, let U_i be the set of all unmarked nodes in Q_i. Also let w be the maximum number of nodes in any quorum.

```
1: Node s sets $R$ to be an array of $w$ integers selected uniformly at random between 1 and $w$.
2: Node s broadcasts $m$ and $R$ to all nodes in $Q_1$.
3: The nodes in $Q_1$ calculate the node $q_2$ using $R[|U_2|]$ to index $U_2$'s nodes sorted by their IDs.
4: All nodes in $Q_1$ send $m$ to node $q_2$.
5: for $i = 2, \dots, \ell - 2$ do
6:     Node $q_i$ selects $q_{i+1} \in U_{i+1}$ uniformly at random.
7:     Node $q_i$ sends $m$ to node $q_{i+1}$.
8: end for
9: Node $q_{\ell-1} \in U_{\ell-1}$ broadcasts $m$ to all nodes in $Q_\ell$.
10: All nodes in $Q_\ell$ send $m$ to node $\mathbf{r}$.
```

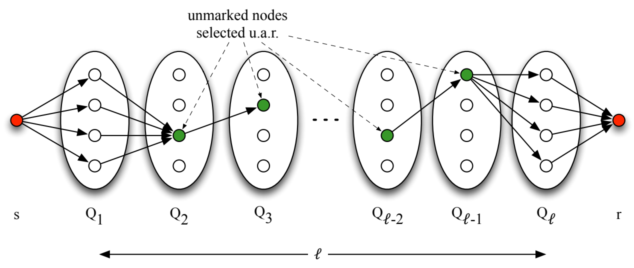


Fig. 3 SEND-PATH Algorithm

SEND-PATH is efficient in terms of message cost and latency; but in the presence of bad nodes, it is vulnerable to corruption. Thus, we make *SEND* call *CHECK* algorithm with probability p_c, where *CHECK* detects with probability p_d if a message has been corrupted in the last call to *SEND-PATH*.

In *CHECK*, the message is sent from node s to node $\mathbf{r}$ through a path of subquorums, where a subquorum is a subset of unmarked nodes selected uniformly at random in a quorum.

Unfortunately, while *CHECK* can determine if a corruption occurred, it does not specify the location where the corruption occurred. Hence, if *CHECK* detects a corruption, *HEAL* algorithm is called.

When *HEAL* 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. These nodes are marked in all quorums they are in and in their neighboring quorums.

In each call to *HEAL*, we mark at most one good node. In order to always provide unmarked nodes to participate in communication, we set the following condition. If $(1/2-\gamma)$-fraction of nodes in any quorum have been marked, for a constant $\gamma > 0$, these nodes are set unmarked.

In *SEND-PATH* and *CHECK*, the message is broadcasted to Q_1 and Q_ℓ to handle any accusation against node s or node $\mathbf{r}$ in *HEAL*.

Our model does not directly consider concurrency. In a real system, concurrent calls to *HEAL* 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.

6.4 *CHECK*

CHECK is implemented as either *CHECK1* or *CHECK2*. In this section, we describe *CHECK1* and *CHECK2*. Then, we compare between them in terms of message cost and latency.

Throughout this section, we let U_j be the set of all unmarked nodes in Q_j for $1 < j < \ell$.

6.4.1 CHECK1

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

Algorithm 4 CHECK1$(m, \mathbf{r})$ ▷ checks in one round if there has been a corruption.

Declaration: for $1 < j < \ell$, let U_j be the set of all unmarked nodes in Q_j and let each subquorum S_j be initially empty. Note that each subquorum will have at most $2 \log\log n$ nodes. Also, let w be the maximum number of nodes in any quorum.

```
1: Node s generates $R$ as an $\ell$ by $w$ by $2 \log\log n$ array of random integers.*
2: Node s sets $m'$ to be a message consisting of $m$, $\mathbf{r}$, and $R$.
3: Node s broadcasts $m'$ to all nodes in $Q_1$.
4: The nodes in $Q_1$ use $R_2$ to calculate the nodes of $S_2$.**
5: The nodes in $Q_1$ send $m'$ to the nodes of $S_2$.
6: for $j \leftarrow 2, \dots, \ell - 2$ do
7:     The nodes of $S_j$ use $R_{j+1}$ to calculate the nodes of $S_{j+1}$.
8:     The nodes of $S_j$ send $m'$ to all nodes of $S_{j+1}$.
9: end for
10: The nodes of $S_{\ell-1}$ broadcast $m'$ to all nodes in $Q_\ell$.
11: The nodes of $Q_\ell$ send $m'$ to node $\mathbf{r}$.
```

* $R[j, k]$ is a multiset of $2 \log\log n$ integers selected uniformly at random with replacement between 1 and k, for $1 < j < \ell$ and $1 \leq k \leq w$.
** $R[j, |U_j|]$, shortly R_j, has the indices of the nodes of S_j selected u.a.r. from the nodes of U_j; note that the nodes of U_j are sorted by their IDs.
Note that: if a node receives inconsistent messages or fails to receive an expected message, then it initiates a call to *HEAL*.

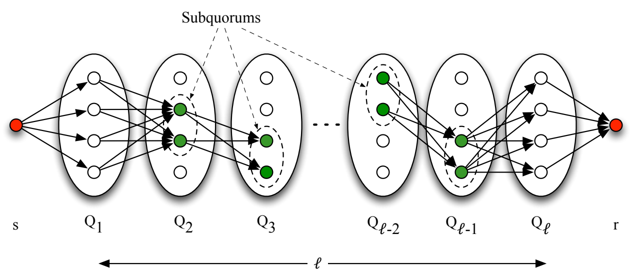


Fig. 4 *CHECK1* Algorithm

CHECK1 is triggered by *SEND* with probability $p_c = 1/(\log\log n)^2$. *CHECK1* runs in one round, in which node **s** sends a message m to node **r** through a path of subquorums via all-to-all communication. (See Figure 4).

Each subquorum, S_j, has $2\log\log n$ nodes that are chosen uniformly at random with replacement from the nodes of U_j in the quorum path, for $1 < j < \ell$.

Note that if any node receives inconsistent messages or fails to receive an expected message during *CHECK1*, it initiates a call to *HEAL*.

CHECK1 fails to detect message corruptions if all nodes of any subquorum in the quorum path are bad. In Section 7, 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 less than $1/2$ when $\ell \leq \frac{\log^2 n}{2}$.

Thus, *CHECK1* detects message corruptions with probability $p_d > 1/2$. This requires $O(\ell(\log\log n)^2 + \log n \cdot \log\log n)$ messages and latency $O(\ell)$. But since *CHECK1* is triggered with probability $1/(\log\log n)^2$, it has an expected message cost of $O(\ell + \log n/\log\log n)$ with latency $O(\ell/(\log\log n)^2)$.

6.4.2 CHECK2

In this section, we describe *CHECK2*, which is stated formally as Algorithm 5.

SEND calls *CHECK2* with probability $p_c = 1/(\log^* n)^2$. When *CHECK2* is triggered, *CHECK2* runs for $4(\log^* n+3)$ rounds.

First, node s generates a public/private key pair (k_p, k_s) to let the nodes that receive k_p verify any subsequent message signed by k_s. 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 *HEAL*.

In each round, node s sends a message m through a path of subquorums of incremental size, in the sense that each subquorum in the quorum path is initially empty; and in each round, a new node $x_j \in U_j$ selected uniformly at random is added to S_j, for all $1 < j < \ell$. (See Figure 5).

CHECK2 detects message corruptions with probability $p_d \geq 1/2$. It requires $O((\ell + \log n)(\log^* n)^2)$ message cost and $O(\ell \log^* n)$ latency. But since *CHECK2* is triggered with

Algorithm 5 CHECK2$(m, \mathbf{r})$ ▷ checks in multiple rounds if there has been a corruption.

Declaration: for $1 < j < \ell$, let U_j be the set of all unmarked nodes in Q_j and let each subquorum S_j be initially empty. Also let w be the maximum number of nodes in any quorum.

```
 1: Node s generates public/private key pair $(k_p, k_s)$.
 2: for $i \leftarrow 1, \ldots, 4(\log^* n + 3)$ do
 3:     Node s generates $R_i$ as an $\ell$ by $w$ array of random integers.*
 4:     Node s sets $m_i = ([m, \mathbf{r}, i, R_i]_{k_s}, k_p)$.
 5:     Node s broadcasts $m_i$ to all nodes in $Q_1$.
 6:     All nodes in $Q_1$ use $R_{i2}$ to calculate node $x_{i2}$ to be added to $S_2$.**
 7:     The nodes in $Q_1$ send $m_i$ to all nodes in $S_2$.
 8:     The nodes in $Q_1$ send $R_1, \ldots, R_{i-1}$ to node $x_{i2}$.
 9:     for $j \leftarrow 2, \ldots, \ell - 2$ do
10:         All $i$ nodes in $S_j$ use $R_{i(j+1)}$ to calculate node $x_{i(j+1)}$ to be added to $S_{j+1}$.
11:         for $k \leftarrow 1, \ldots, i-1$ do
12:             Node $x_{kj}$ sends $m_k$ to node $x_{i(j+1)}$.
13:             Node $x_{ij}$ uses $R_{k(j+1)}$ to calculate node $x_{k(j+1)}$.
14:             Node $x_{ij}$ sends $m_i$ to node $x_{k(j+1)}$.
15:         end for
16:         Node $x_{ij}$ sends $m_i$ to node $x_{i(j+1)}$.
17:     end for
18:     The nodes in $S_{\ell-1}$ broadcast $m_i$ to all nodes in $Q_\ell$.
19:     All nodes in $Q_\ell$ send $m_i$ to node $\mathbf{r}$.
20: end for
```

* $R_i[j, k]$ is a uniformly random integer between 1 and k, for $1 < j < \ell$ and $1 \leq k \leq w$.
** $R_i[j, |U_j|]$, shortly R_{ij}, is the index of node, x_{ij}, to be selected u.a.r. from the nodes of U_j in round i; note that the nodes of U_j are sorted by their IDs.
Note that: 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 *HEAL*.

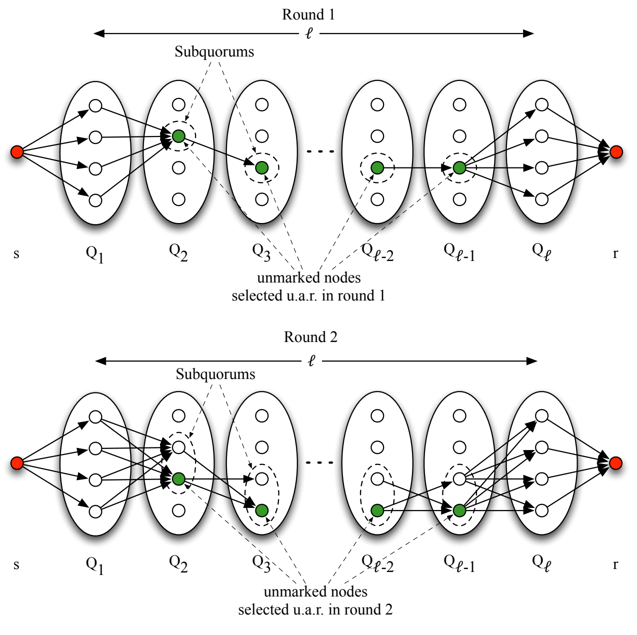


Fig. 5 *CHECK2* Algorithm

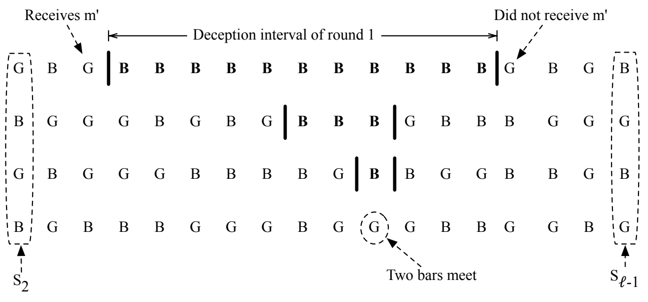
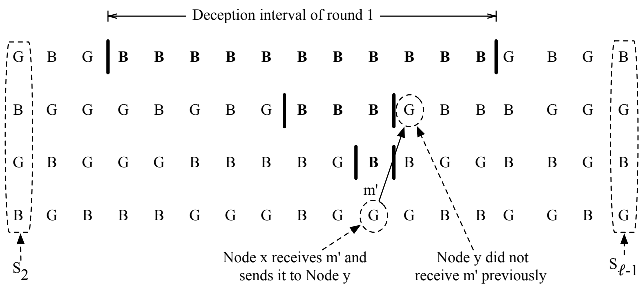


Fig. 6 Example run of *CHECK2*



Deception interval of round 1
m'
Node x receives m' and sends it to Node y
Node y did not receive m' previously
S2
Sℓ-1

Fig. 7 A corruption is detected after the two bars meet.

probability $1/(\log^* n)^2$, it has expected message cost $O(\ell + \log n))$ with expected latency $O(\ell/\log^* n)$.

An example run of *CHECK2* is illustrated in Figure 6. In this figure, there is a column for each subquorum from S_2 to $S_{\ell-1}$ 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).

We define a *deception interval* in a round as a path of bad nodes that are selected in this round to be added to the subquorums in the quorum path.

In Figure 6, we show the deception intervals maintained by the adversary. These intervals are outlined by left and right bar in each row. The adversary's strategy is to maintain the longest deception interval in the first row, and it keeps corrupting the message in all subsequent deception intervals.

The left bar in each row specifies the rightmost subquorum in which there is some good node that receives the correct message m'. The right bar in each row specifies the leftmost subquorum in which there is some good node that does not receive m'.

There are two key points by which *CHECK2* detects message corruptions: 1) any deception interval in any round never expands in any subsequent round; and 2) any deception interval shrinks to length zero after $O(\log^* n)$ rounds.

Deception intervals never expand. In order to show that any deception interval never expends over rounds. We show that the left bar never moves leftwards, and the right bar never moves rightwards.

The left bar never moves leftwards because each good node receives the message m' in round i has to receive the same message in all subsequent rounds; otherwise, it will call *HEAL*. Moreover, we know that for each round i, all nodes in each subquorum S_j send the message to the new node that is selected in this round to be added to S_{j+1} for $1 < j < \ell$. Thus, those good nodes that receive and provide m' to the deception interval in round i will provide the same message to all subsequent deception intervals.

Now we show that the right bar never moves rightwards. Recall that in each round, each node that is added to each subquorum S_j sends the message it has received to all nodes in S_{j+1} for $1 < j < \ell$. Thus, all good nodes that receive a message through a deception interval in any round expect to receive the same message in all subsequent rounds. Also, each good node that did not receive m' in any round must not receive this message in all subsequent rounds; otherwise, it will initiate a call to *HEAL*. This shows that the right bar never moves rightwards.

Deception intervals shrink logarithmically. The reason that *CHECK2* requires $O(\log^* n)$ 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 $1/2$, and value "G" with probability at least $1/2$, and we toss it x times, then the expected length of the longest run of B's is $O(\log x)$. 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 $O(\log x)$. 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 Lemma 5 (Section 7).

When the two bars meet, the corruption is detected. Figure 7 shows that when the two bars meet, a corruption is detected. In this figure, as the deception intervals shrink over rounds, node x in the last round receives message m'. Then node x forwards this message to node y which has not previously received m' in this call to *CHECK2*. As a result, node y calls *HEAL* declaring that it has received inconsistent messages.

If all nodes in some subquorum are bad, does *CHECK2* successfully detect message corruptions? We know that *CHECK1* fails if all nodes in any subquorum in the quorum path are bad. However, *CHECK2* can detect corruptions even if all nodes in some subquorums are bad.

Recall that *CHECK2* runs in $O(\log^* n)$ rounds. In each round, new nodes are selected uniformly at random to be added to the subquorums. This makes the adversary not be able to know, before all rounds finish, if all the nodes in any particular subquorum are bad. Thus, the adversary would rather maintain the longest deception interval in the first

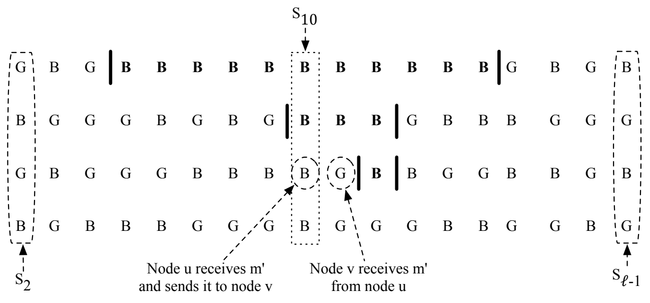

S_{10}
S_2
$S_{\ell-1}$
Node u receives m' and sends it to node v
Node v receives m' from node u

Fig. 8 A corruption is detected even if all nodes in subquorum S_{10} are bad.

round and keeps corrupting the message over all subsequent deception intervals. Note that if the adversary corrupts the message in more than one interval in the same round, it will increase the chance of detecting message corruption. Figure 8 shows that even though all nodes in S_{10} are bad, the good node v receives message m' from the bad node u, where node u is not in the deception interval chosen by the adversary in row 3.

6.4.3 A comparison between CHECK1 and CHECK2

CHECK1 has the following advantages over *CHECK2*. 1) *CHECK1* has less latency compared to *CHECK2*, where each call to *CHECK1* runs in one round, and each call to *CHECK2* runs in $O(\log^* n)$ rounds; and 2) *CHECK1* has fewer calls to broadcast, in particular, any call to *CHECK1* calls *BROADCAST* twice, and any call to *CHECK2* calls *BROADCAST* $O(\log^* n)$ times.

However, *CHECK2* has the following advantages over *CHECK1*. 1) *CHECK2* has less message cost for large n; and 2) *CHECK2* can handle a quorum path of length $\ell \leq n$ but *CHECK1* handles a quorum path of length $\ell \leq \frac{\log^2 n}{2}$.

6.5 HEAL

When a message is corrupted and *CHECK* detects such corruption, *HEAL* is called. *HEAL* is described formally as Algorithm 6.

The purpose of calling *HEAL* is 1) to determine the location at which the corruption has occurred; and 2) to mark the nodes that are in conflict.

When *HEAL* starts, all nodes in each quorum in the quorum path are notified. To determine the location at which the corruption is occurred, *HEAL* investigates the corruption situation, where each node, previously involved in *SEND-PATH* or *CHECK*, broadcasts to all nodes in its quorum and the neighboring quorums, the messages they have received (and from whom) and the messages they have sent (and to whom) in the previous call to *SEND-PATH* or *CHECK*.

Algorithm 6 *HEAL* ▷ Node $q' \in Q'$ calls *HEAL* after it detects a corruption.

```
1: q' broadcasts, the fact that it calls HEAL along with all the messages that it has received in this call to SEND, to all nodes in Q'.
2: The nodes in Q' verify that q' received inconsistent messages before proceeding.
3: Q' notifies that a call to HEAL is occurring, via all-to-all communication, to all quorums in the quorum path.
4: INVESTIGATE
5: MARK-IN-CONFLICTS
```

Algorithm 7 *INVESTIGATE* ▷ investigates the corruption situation

```
1: for each node, $q \notin \{\mathbf{s}, \mathbf{r}\}$, involved in the last call to SEND-PATH or CHECK do
2:     q compiles all messages they have received (and from whom) and they have sent (and to whom) in the last call to SEND-PATH or CHECK.
3:     q broadcasts these messages to all nodes in its quorum and the neighboring quorums.
4: end for
```

As a result of the investigation, *HEAL* identifies at least one pair of nodes that are in conflict. We say that a pair of nodes are in conflict if they each have broadcasted messages that are in conflict with the messages broadcasted by the other.

Algorithm 8 *MARK-IN-CONFLICTS* ▷ marks the nodes that are in conflict

```
1: for each pair of nodes, $(q_k, q_{k+1}) \in (Q_k, Q_{k+1})$, that is in conflict*, for $1 \leq k < \ell$ do
2:     node $q_{k+1}$ broadcasts a conflict message "$\{q_k, q_{k+1}\}$" to all nodes in $Q_{k+1}$,
3:     each node in $Q_v$ forwards "$\{q_k, q_{k+1}\}$" to all nodes in $Q_{k+1}$ and all nodes in $Q_k$,
4:     all nodes in $Q_k$ (or $Q_{k+1}$) send "$\{q_k, q_{k+1}\}$" to all other quorums that has node $q_k$ (or $q_{k+1}$).
5:     all nodes in each quorum that has $q_x$ or $q_{k+1}$ send "$\{q_k, q_{k+1}\}$" to the neighboring quorums.
6: end for
7: for each node, q, that receives a conflict message "$\{q_k, q_{k+1}\}$" do
8:     q marks the nodes $q_k$ and $q_{k+1}$ in its marking table.
9: end for
10: if $(1/2 - \gamma)$-fraction of nodes in any quorum have been marked, for $\gamma = 0.01$ then
11:     each of these nodes is set unmarked in all quorums.
12:     each of these nodes is set unmarked in all its neighboring quorums.
13: end if
```

* A pair of nodes, (q_k, q_{k+1}) is *in conflict* if: 1) q_k was scheduled to send a message to q_{k+1} at some point in the last call to *SEND-PATH* or *CHECK*; and 2) q_{k+1} does not receive an expected message from q_k in *INVESTIGATE*, or q_{k+1} receives a message in *INVESTIGATE* that is different than the message that it has received from q_k in the last call to *SEND-PATH* or *CHECK*.

Each pair of nodes that *HEAL* has identified to be in conflict are marked in their quorums and the neighboring quorums.

Moreover, each pair of nodes that are in conflict has at least one bad node. Thus, at most one good node is marked in each call to *HEAL*. In order to keep providing unmarked nodes to participate in *SEND-PATH* and *CHECK*, we set the constraint that if a $(1/2-\gamma)$-fraction of nodes in any quorum has been marked, they are set unmarked in all their quorums and the neighboring quorums.

Even though we unmark nodes in some situation, we provide a potential function argument (Lemma 7), which shows that all bad nodes are marked after $O(t)$ calls to *HEAL*. After all bad nodes are marked, no more corruptions occur.

7 Analysis

In this section, we provide the analysis of our algorithms. We first prove that *CHECK1* succeeds to detect corruptions with probability more than $1/2$. Then, we prove the lemmas required for Theorem 1, in which *SEND* calls *CHECK2*. Note that throughout this section, we let all logarithms be base 2.

7.1 *CHECK1*

First, we show that if a message is corrupted in *SEND-PATH*, *CHECK1* succeeds to detect such corruption with probability more than $1/2$.

Lemma 1 *If* $\ell \leq \frac{\log^2 n}{2}$*, then CHECK1 fails to detect any message corruption with probability less than* $1/2$.

Proof *CHECK1* succeeds in detecting the message corruption if every subquorum has at least one good node.

Note that the fraction of bad nodes in any quorum is at most $1/4$. Note further that at least $(1/2+\gamma)$-fraction of the nodes in any quorum are unmarked, for $\gamma > 0$. Thus, the probability that an unmarked bad node is selected uniformly at random is at most $\frac{1/2}{1+2\gamma}$. Therefore, the probability that any subquorum of size $2\log\log n$ has only bad nodes is less than

$$(1/2)^{2\log\log n} = 1/\log^2 n.$$

Union-bounding over all ℓ subquorums, the probability that *CHECK1* fails to detect corruptions is less than $\ell/\log^2 n$. For $\ell \leq \frac{\log^2 n}{2}$, the probability that *CHECK1* fails is less than $1/2$. □

7.2 *CHECK2*

In order to show that *CHECK2* succeeds to detect corruptions with probability at least $1/2$, we first define the deception interval.

Definition 1 A deception interval, $d_i(j,k)$, is a path of unmarked bad nodes, x_{iw}'s, that are added to the subquorums, S_w's, in round i, for $1 < j \leq w \leq k < \ell$, such that: 1) Q_{j-1} is the rightmost quorum that has at least one good node which provides the correct message to node x_{ij}; and 2) Q_{k+1} is the leftmost quorum that has at least one good node to which node x_{ik} is scheduled to send and does not provide the correct message.

Note that we say a deception interval, $d_i(j,k)$, in round i expands in any subsequent round if there exists a deception interval $d_{i'}(j',k')$ in round $i' > i$ such that $j' < j \leq k'$ or $j' \leq k < k'$.

Note further that we say a deception interval, $d_i(j,k)$, in round i shrinks to length x in round $i' > i$ if there exists a deception interval, $d_{i'}(j',k')$, in round i' such that $j \leq j' \leq k' \leq k$ and $x = k'-j'+1 < k-j+1$.

Then, we prove that 1) any deception interval never expands; and 2) any deception interval shrinks logarithmically from round to round. This will imply that in $O(\log^* n)$ rounds, any deception interval shrinks to length zero at which the corruption is detected.

Lemma 2 *Any deception interval in any round never expands in any subsequent round; otherwise, HEAL will be called.*

Proof For each deception interval $d_i(j,k)$, we have the following.

All good nodes in Q_{j-1} that are selected and have received k_p in rounds i or less, they must receive uncorrupted messages signed by k_s, in all rounds subsequent to i; otherwise, *HEAL* will be called. Those good nodes that receive the message signed by k_s will provide this message to node x_{ij} in $d_i(j,k)$. They will also provide the nodes in each subsequent deception interval, $d_{i'}(j',k')$, with the same message, for all $i' > i$ and $j' \geq j$.

Also, all good nodes in Q_{k+1} that have not received the correct message from node x_{ik} in $d_i(j,k)$ must not receive this message from the nodes of each subsequent deception interval, $d_{i'}(j',k')$, for all $i' > i$ and $k' \leq k$; otherwise, they will call *HEAL*. □

Now we prove that any deception interval shrinks logarithmically from round to round.

Lemma 3 *When a coin is flipped x times independently given that each coin gives tail with probability at most* $(1/2-\epsilon)$*, for any constant* $\epsilon > 0$*, then the probability of having any substring of tails of length at least* $\max(1, 2\log x)$ *is at most* $1/2$.

Proof The probability of having a specific substring of tails of length at least $2\log x$ is

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

Union bounding over all possible substrings of length $2\log x$, then the probability of having a substring of tails of length at least $2\log x$ 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 length at least $\max(1, 2\log x)$ is trivially at most $(1/2-\epsilon)$ for $\epsilon > 0$. □

Corollary 1 *When a coin is flipped x times independently given that each coin gives tail with probability at most $(1/2-\epsilon)$, for any constant $\epsilon > 0$, then the probability of having any substring of tails of length at least $\max(1, \lfloor x/2 \rfloor)$ is at most $1/2$.*

Now let $f(n) = 2\log n$, and let $f^{(i)}(n)$ be the function of applying function f, i times, over n. Also, we let $\log^{(i)}(n)$ be the function of applying logarithm i times over n.

Fact 1 $\forall n > 4$ *and* $\forall i \geq 1$ *such that* $\log^{(i)}(n) \geq 2$,

$$f^{(i)}(n) \leq 4\log^{(i)}(n).$$

Proof We prove by induction over $i \geq 1$ that for $n > 4$ and $\log^{(i)}(n) \geq 2$,

$$f^{(i)}(n) \leq 4\log^{(i)}(n).$$

Base case: for $i = 1$, by definition,

$$f(n) = 2\log n \leq 4\log n.$$

Induction hypothesis: for $\log^{(j)}(n) \geq 2$,

$$\forall j < i, f^{(j)}(n) \leq 4\log^{(j)}(n).$$

Induction step: by definition,

$$f^{(i)}(n) = f(f^{(i-1)}(n)).$$

By induction hypothesis, for $\log^{(i-1)}(n) \geq 2$,

$$f^{(i-1)}(n) \leq 4\log^{(i-1)}(n).$$

Then, we have

$$f^{(i)}(n) \leq f(4\log^{(i-1)}(n)) = 2\log(4\log^{(i-1)}(n)),$$

or equivalently,

$$f^{(i)}(n) \leq 2(2 + \log^{(i)}(n)) \leq 4\log^{(i)}(n),$$

for $\log^{(i)}(n) \geq 2$. □

Now let $f^*(n)$ be the smallest value i such that $f^{(i)}(n) \leq 16$.

Fact 2 $\forall n > 4, f^*(n) \leq \log^* n - 2$.

Proof Let $j = \log^* n - 2$. We know that $2 < \log^{(j)}(n) \leq 4$. And so by Fact 1, we have

$$f^{(j)}(n) \leq 4\log^{(j)}(n) \leq 16.$$

Thus, by definition, $f^*(n) \leq j = \log^* n - 2$. □

Lemma 4 *Assume that any deception interval of length x shrinks to length $2\log x$ in a successful step. Then, for any deception interval of length $x' > 16$, after $\log^* x' - 2$ successful steps, it shrinks to a length of at most 16.*

Proof Fact 2 proves this lemma. □

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

Lemma 5 *Assume some node selected uniformly at random in the last call to SEND-PATH has corrupted a message in a quorum path of length $\ell \leq n$. Then when CHECK2 is called, with probability at least $1/2$, some node will call HEAL.*

Proof By Lemma 2, any deception interval never expands over rounds. For shrinking deception intervals over rounds, we make use of Lemma 3 to shrink logarithmically any deception interval of length more than 16; otherwise, deception intervals shrink geometrically using Corollary 1.

Let X_i be an indicator random variable that is equal 1 if the deception interval in round i shrinks logarithmically in round $i+1$; and 0 otherwise.

Recall that the longest deception interval has length of at most $\ell \leq n$. By Lemma 4, after having at most $\log^* n - 2$ of X_i's equal 1, the longest deception interval of length more than 16 shrinks to a deception interval of length at most 16.

Also let Y_j be an indicator random variable that is equal 1 if the deception interval of length $x \leq 16$ in round j shrinks geometrically to a deception interval of length at most $\lfloor x/2 \rfloor$ in round $j+1$; and 0 otherwise.

Thus, we require at most $\log^* n - 2$ of X_i's equal 1 to shrink the longest deception interval, d, of length at most n to a deception interval, d', of length at most 16. Further, we require at most 5 Y_j's equal 1, to shrink d' to length 0.

By Lemma 3, each X_i is 1 with probability at least $1/2$; and by Corollary 1, each Y_j is 1 with probability at least $1/2$.

Let

$$X = \sum_{i=1}^{4(\log^* n-2)} X_i$$

and

$$Y = \sum_{j=4(\log^* n-2)+1}^{4(\log^* n+3)} Y_j.$$

Now let Z_k be an indicator random variable that is 1 with probability $1/2$; and 0 otherwise, for $1 \leq k \leq 4(\log^* n+3)$; and let

$$Z = \sum_{k=1}^{4(\log^* n+3)} Z_k.$$

We know that for all i, j and k, X_i and Y_j are stochastically larger than Z_k. Thus, $X + Y$ is stochastically larger than Z. Therefore,

$$\Pr\left(X + Y \geq \log^* n + 3\right) \geq \Pr\left(Z \geq \log^* n + 3\right),$$

or equivalently,

$$1 - \Pr\left(X + Y < \log^* n + 3\right) \geq 1 - \Pr\left(Z < \log^* n + 3\right).$$

Thus, we obtain

$$\Pr\left(X + Y < \log^* n + 3\right) \leq \Pr\left(Z < \log^* n + 3\right).$$

Note that $\mathbf{E}(Z) = 2(\log^* n + 3)$. Since the Z_k's are independent random variables, by Chernoff bounds,

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

For $n > 1$ and $\delta = \frac{1}{2}$,

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

It is trivial the case that $n = 1$, in which the network has only one node, which is node $\mathbf{r}$.

Thus, the probability that *CHECK2* succeeds in finding a corruption and calling *HEAL* is at least $1/2$. □

7.3 *HEAL*

Lemma 6 *If some node selected uniformly at random in the last call to SEND-PATH has corrupted a message, then the algorithm HEAL 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 $\mathbf{s}$, 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, *HEAL* 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 *HEAL*. 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 *HEAL* 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 *HEAL*. 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 *HEAL* must be called before all bad nodes are marked.

Lemma 7 *HEAL is called $O(t)$ times before all bad nodes are marked.*

Proof By Lemma 6, if a message is corrupted in the last call to *SEND-PATH* and is caught by *CHECK*, then *HEAL* is called. *HEAL* identifies at least one pair of nodes that are in conflict.

Let p be the probability of selecting an unmarked bad node uniformly at random. Recall that the fraction of bad nodes in any quorum is at most $1/4$ and at any moment the fraction of unmarked nodes in any quorum is at least $(1/2 + \gamma)$ for $\gamma > 0$. Thus, we have

$$p \leq \frac{1}{2}\left(\frac{1}{1+2\gamma}\right).$$

Now let b be the number of bad nodes that are marked, and let g be the number of good nodes that are marked. Further, we let

$$f(b,g) = b - \left(\frac{p}{1-p}\right) g.$$

For each corruption caught, at least one bad node is marked. This implies that b increases by at least 1 and g increases by at most 1. Note that $\frac{p}{1-p} < 1$. Thus, $f(b,g)$ increases by at least $(1 - \frac{p}{1-p}) > 0$.

Moreover, when a $(1/2 - \gamma)$-fraction of nodes in any quorum Q of size $|Q|$ get unmarked for a constant $\gamma > 0$, b decreases by at most $p(1/2 - \gamma)|Q|$ and g decreases by at least $(1-p)(1/2-\gamma)|Q|$. This implies that $f(b,g)$ further increases by at least 0.

Thus, $f(b,g)$ is monotonically increasing by at least $(1-\frac{p}{1-p}) > 0$ for each corruption caught. Note that when all bad nodes are marked,

$$f(b,g) \leq t.$$

Therefore, all bad nodes are marked after at most $\left(\frac{1-p}{1-2p}\right)t$, or equivalently at most $\left(1+\frac{1}{2\gamma}\right)t/2$, calls to *HEAL*. $\square$

7.4 Our Theorem

Now we prove Theorem 1. Note that we consider that *SEND* calls *CHECK2*.

Theorem 1 Assume we have a network with n nodes and $t \leq (1/4-\epsilon)n$ bad nodes, for any constant $\epsilon > 0$. Then our algorithm has the following properties. 1) In an amortized sense, any call to *SEND* has $O(\ell + \log n)$ expected number of messages with $O(\ell)$ expected latency; and 2) the expected total number of times that *SEND* fails to deliver a message reliably is $O(t(\log^* n)^2)$.

Proof We first show the message complexity and the latency of our algorithms. By Lemma 7, the number of calls to *HEAL* is $O(t)$. Thus, the resource cost of all calls to *HEAL* are bounded as the number of calls to *SEND* grows large. Therefore, for the amortized cost, we consider only the cost of the calls to *SEND-PATH* and *CHECK2*.

When sending a message through ℓ quorums, *SEND-PATH* has message cost $O(\ell + \log n)$ and latency $O(\ell)$. Recall that *CHECK2* has a message cost of $O((\ell + \log n)(\log^* n)^2)$ and a latency of $O(\ell \log^* n)$, but *CHECK2* is called only with probability $1/(\log^* n)^2$. Hence, the call to *SEND* has amortized expected message cost $O(\ell + \log n)$ and amortized expected latency $O(\ell)$.

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 $O(\mathcal{L} + t \cdot (\ell_M \log^2 n + \log^5 n))$, and the latency is $O(t \cdot \ell_M)$.

This is true since each call to *HEAL* has message cost $O(\ell_M \log^2 n + \log^5 n)$ and latency $O(\ell_M)$, where:

1. the node, x, making the call to *HEAL* broadcasts its reason of calling *HEAL* to all nodes in its quorum, this has message cost $O(\log n)$ and latency $O(1)$;
2. all nodes in every quorum in the quorum path are notified via all-to-all communication when *HEAL* is called, these notifications have a message cost of $O(\ell_M \log^2 n)$ and a latency of $O(\ell_M)$;
3. *HEAL* has $O(\log^* n)$ broadcasts over at most ℓ_M quorums, that has message cost $O(\ell_M \log^* n \cdot \log n)$ and latency $O(1)$;
4. the message cost when all nodes in Q_1 and Q_ℓ broadcast is $O(\log^2 n)$ with latency $O(1)$;
5. marking a pair of nodes that are in conflict has message cost $O(\log^3 n)$ and latency $O(1)$; and
6. note that marking a pair of nodes that are in conflict could cause $O(\log n)$ quorums to be unmarked. Note further that unmarking $O(\log n)$ nodes in any quorum has a message cost of $O(\log^4 n)$. Thus, unmarking $O(\log n)$ quorums has message cost $O(\log^5 n)$ and latency $O(1)$.

Now we show the expected total number of corruptions. Recall that by Lemma 7, the number of calls to *HEAL* before all bad nodes are marked is $O(t)$. Thus, *CHECK2* must detect corruptions and calls *HEAL* $O(t)$ times. Moreover, if a bad node caused a corruption during a call to *SEND-PATH*, then by Lemmas 5 and 6, with probability at least $1/2$, *CHECK2* will catch it. Note that $CHECK2$ is called with probability $\frac{1}{(\log^* n)^2}$. Therefore, the expected total number of corruptions is $O(t(\log^* n)^2)$. $\square$

8 Empirical Results

8.1 Setup

In this section, we empirically compare between two algorithms in terms of the message cost, the latency, the fraction of messages corrupted and the expected total number of corruptions via simulation.

The first algorithm we simulate is *no-self-healing* algorithm from [25]. 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 triggering *CHECK1* and *CHECK2* separately.

In our experiments, we consider a butterfly network with two sizes: $n = 14{,}116$ and $n = 30{,}509$, where $\ell = \lfloor \log n \rfloor - 2$ and the quorum size is $\lfloor 4 \log n \rfloor$.

In one experiment, *SEND* calls *CHECK1* with probability $1/(\log\log n)^2$ and with subquorum size $\lfloor 2 \log\log n \rfloor$. Another experiment has *SEND* trigger *CHECK2* with probability $1/(\log^* n)^2$ and with subquorum size $\lfloor 2 \log^* n \rfloor$.

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 $f = t/n$. Note that for larger f, marking and unmarking processes are performed more frequently. This makes the simulation take longer to eventually mark all bad nodes.

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, where node $\mathbf{s}$ sends a message to node $\mathbf{r}$. We simulate an adversary who at the beginning of each simulation chooses uniformly at random without replacement a fixed number of nodes to control. Our adversary attempts to corrupt messages between nodes whenever possible. Aside from attempting to corrupt messages, the adversary performs no other attacks.

8.2 Results

The results of our experiments are shown in Figures 9, 10, 11, 12, 13 and 14.

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. Second, for a fixed number of calls to *SEND*, the message cost per *SEND* decreases as the total number of bad nodes decreases. In particular, when there are no bad nodes, *self-healing* has dramatically less message cost than *no-self-healing*.

In our experiments, we show the expected number of messages per call to *SEND*, the expected latency per call to *SEND*, the fraction of messages corrupted for each call to *SEND* and the expected total number of corruptions, for $n = 14{,}116$ and $n = 30{,}509$.

8.2.1 Expected number of messages

Figures 9 and 10 show that before all bad nodes are marked: 1) the expected number of messages per call to *SEND* decreases as the total number of calls to *SEND* increases; and 2) for a fixed number of calls to *SEND*, the expected number of messages per call to *SEND* decreases as f decreases.

Moreover, Table 1 shows that when all bad nodes are marked, *self-healing* has dramatically less expected number of messages per call to *SEND* than *no-self-healing*.

n	*self-healing*		*no-self-healing*
	CHECK1	*CHECK2*	
14,116	598	1,078	30,390
30,509	649	1,177	39,100

Table 1 Expected # messages per call to *SEND* in *self-healing* and in *no-self-healing* for $n = 14{,}116$ and $n = 30{,}509$.

8.2.2 Expected latency

Figures 11 and 12 show that the latency of *no-self-healing* is always less than the latency of *self-healing* due to the latency of *CHECK1* and *CHECK2*.

Moreover, Table 2 shows that for $n = 14{,}116$ and $n = 30{,}506$, after all bad nodes are marked, we have that: 1) *self-healing* calling *CHECK2* has more latency than *self-healing* calling *CHECK1*; and 2) the latency of *self-healing* is at most twofold the latency of *no-self-healing*.

n	*self-healing*		*no-self-healing*
	CHECK1	*CHECK2*	
14,116	17	23	12
30,509	18	25	13

Table 2 Expected latency per call to *SEND* in *self-healing* and in *no-self-healing* for $n = 14{,}116$ and $n = 30{,}509$.

8.2.3 Fraction of messages corrupted

The fraction of messages corrupted per a call to *SEND* presents the probability that the message is corrupted in this call.

In Figures 13 and 14, *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.

8.2.4 Expected total number of corruptions

In Figures 13 and 14, for each network given the number of nodes and the fraction of bad nodes, if we integrate the corresponding curve, then we get the total number of times that the message is corrupted in all calls to *SEND* in this network.

f	total # corruptions	Σ_1
1/64	3,457	4,102
1/32	6,930	8,507
1/16	13,831	18,526
1/8	27,721	47,641

Table 3 Total # corruptions when *SEND* calls *CHECK1* for $n = 14{,}116$.

f	total # corruptions	Σ_2
1/64	3,454	7,293
1/32	6,918	15,124
1/16	13,845	32,859
1/8	27,685	84,696

Table 4 Total # corruptions when *SEND* calls *CHECK2* for $n = 14{,}116$.

Tables 3, 4, 5 and 6 show the fact that when *SEND* calls *CHECK1*, the expected total number of corruptions is at most Σ_1, where

$$\Sigma_1 = 2\left(\frac{1-2f}{1-4f}\right) t(\log^* n)^2,$$

and when *SEND* calls *CHECK2*, the expected total number of corruptions is at most Σ_2, where

$$\Sigma_2 = 2\left(\frac{1-2f}{1-4f}\right) t(\log\log n)^2.$$

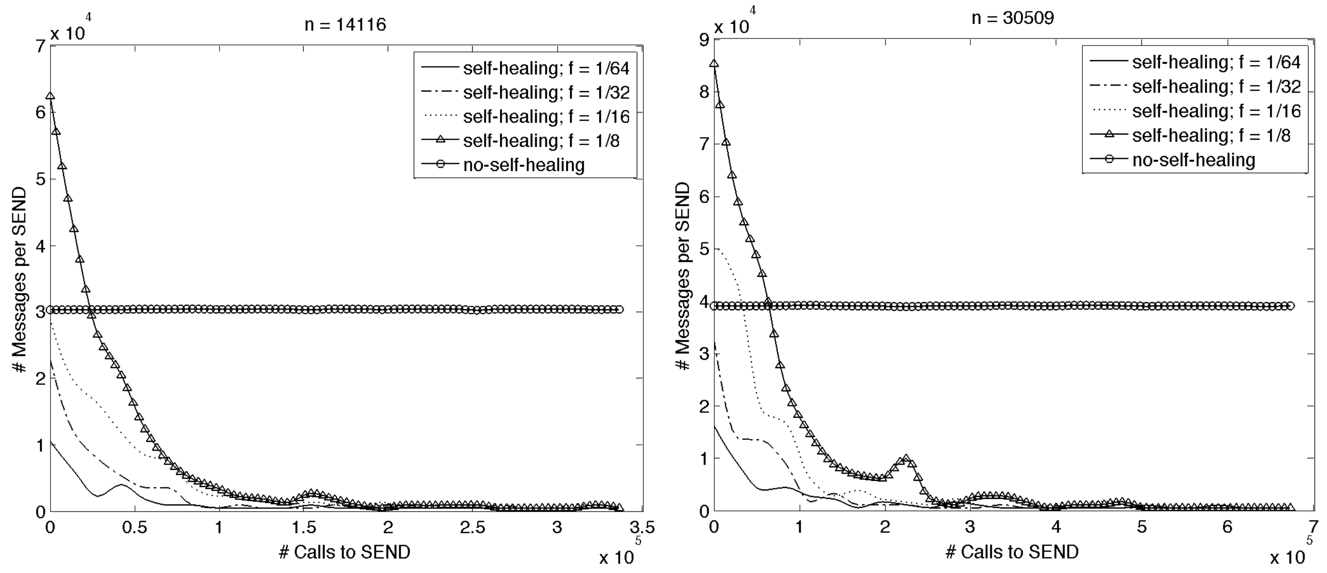


Fig. 9 # messages per call to *SEND* versus # calls to *SEND*, for $n = 14{,}116$ and $n = 30{,}509$, when *SEND* calls *CHECK1*.

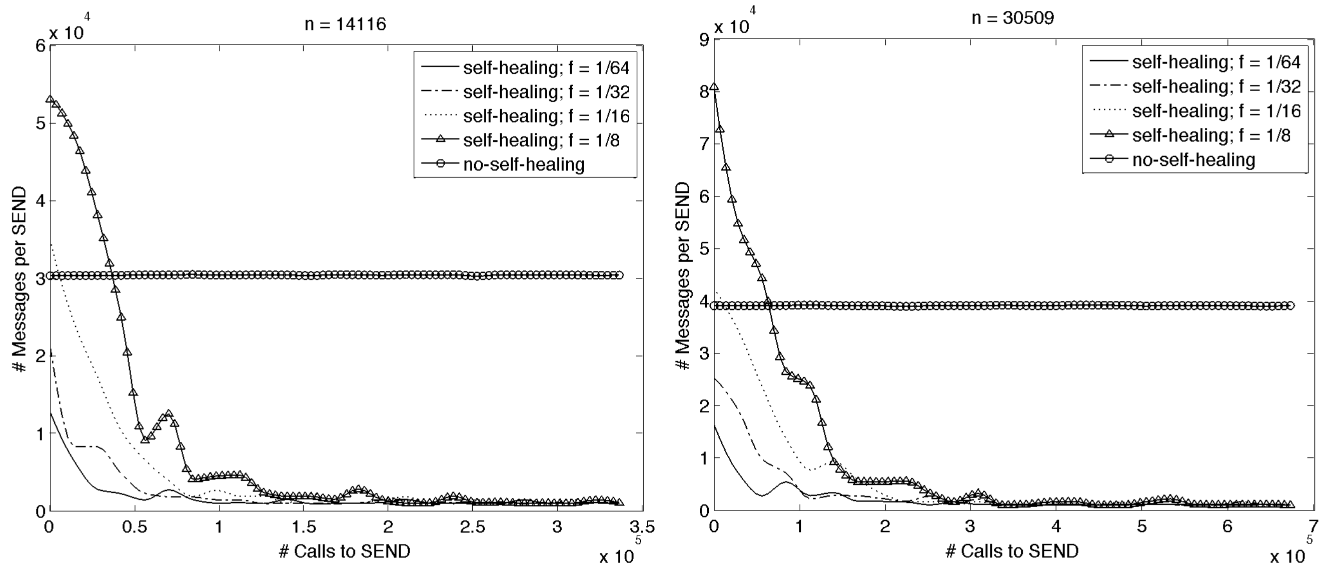


Fig. 10 # messages per call to *SEND* versus # calls to *SEND*, for $n = 14{,}116$ and $n = 30{,}509$, when *SEND* calls *CHECK2*.

f	total # corruptions	Σ_1
1/64	7,490	8,866
1/32	14,996	18,386
1/16	29,949	40,042
1/8	59,932	102,967

Table 5 Total # corruptions when *SEND* calls *CHECK1* for $n = 30{,}509$.

f	total # corruptions	Σ_2
1/64	7,498	15,762
1/32	14,989	32,687
1/16	29,970	71,187
1/8	59,969	183,054

Table 6 Total # corruptions when *SEND* calls *CHECK2* for $n = 30{,}509$.

9 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 $t < n/4$ bad nodes in the network, our algorithm allows $O(t(\log^* n)^2)$ message transmissions to be corrupted in expectation.

There are many issues remain.

– We assume that the sender is a good node, can we extend our algorithms to tolerate if the sender is bad in such a

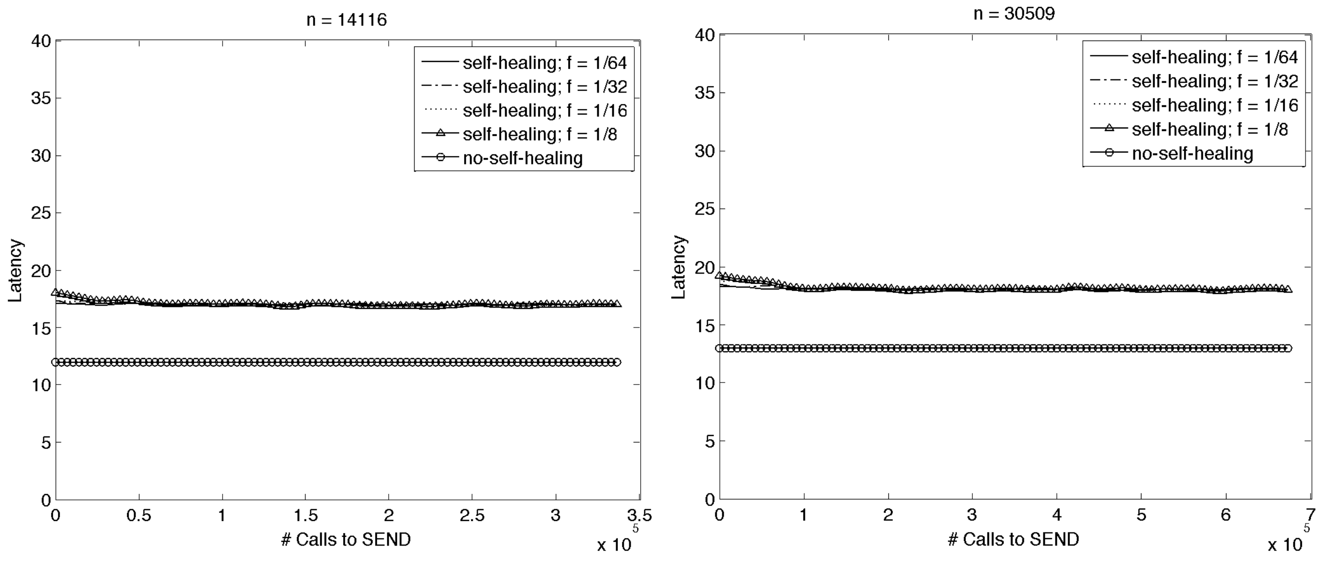


Fig. 11 Latency per call to *SEND* versus # calls to *SEND*, for $n = 14{,}116$ and $n = 30{,}509$, when *SEND* calls *CHECK1*.

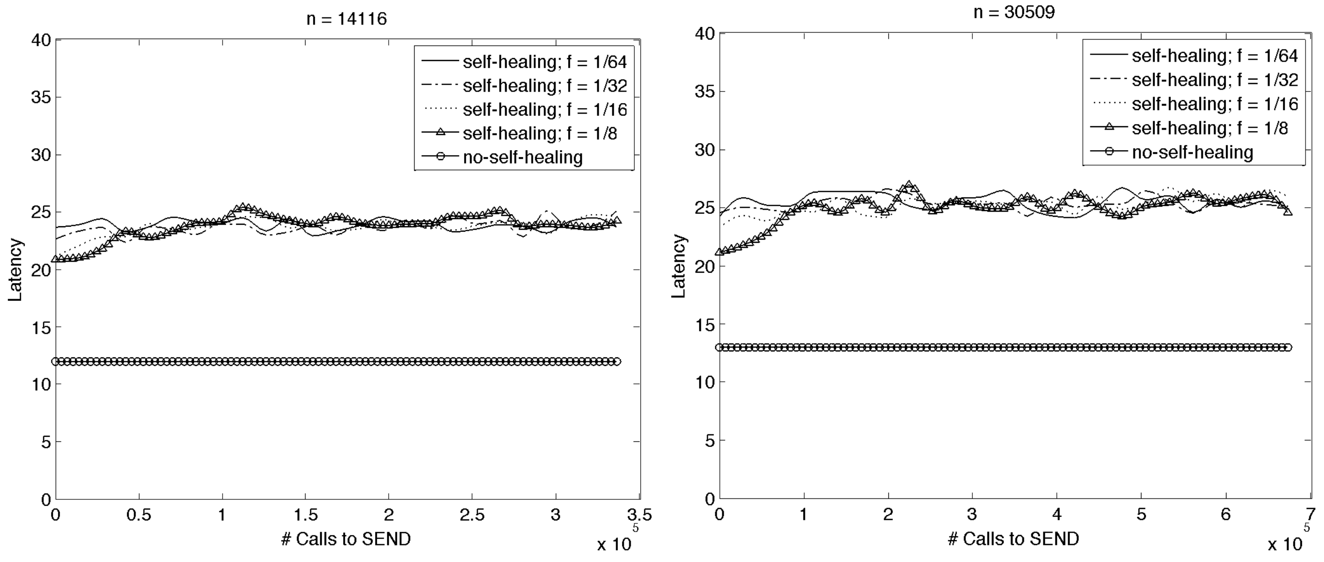


Fig. 12 Latency per call to *SEND* versus # calls to *SEND*, for $n = 14{,}116$ and $n = 30{,}509$, when *SEND* calls *CHECK2*.

way that 1) our algorithms will remain efficient in terms of message cost; and 2) they will be resistant to denial of service attacks?

– 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?
– 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?
– Can we optimize constants and make use of heuristic techniques in order to significantly improve our algorithms' empirical performance?
– In our algorithm *CHECK2*, we provide an array of random integers in each round to select nodes uniformly at random in order to participate for detecting message corruptions. Each array has $O(\ell \log n \log\log n)$ bits. If the message m has b bits, then the communication complexity per call to *SEND* is $O((\ell + \log n)(\ell \log n \log\log n + b))$ and the communication complexity of the naive algorithm is $O(\ell \log^2 n \cdot b)$. In order to improve the communication complexity of our algorithms, can we reduce the number of bits that is required to represent the array of random integers to $O(\ell \log\log n)$?
– We assume a partially synchronous communication model, which is crucial for our *CHECK2* algorithm to detect

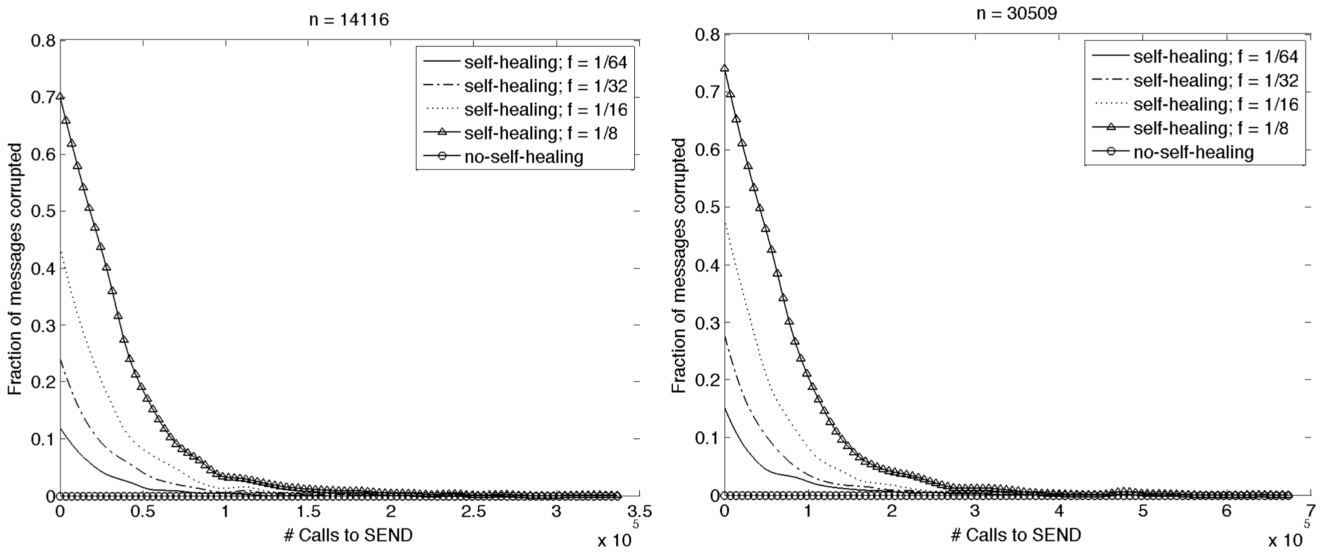


Fig. 13 Fraction of messages corrupted versus # calls to *SEND*, $n = 14{,}116$ and $n = 30{,}509$, when *SEND* calls *CHECK1*.

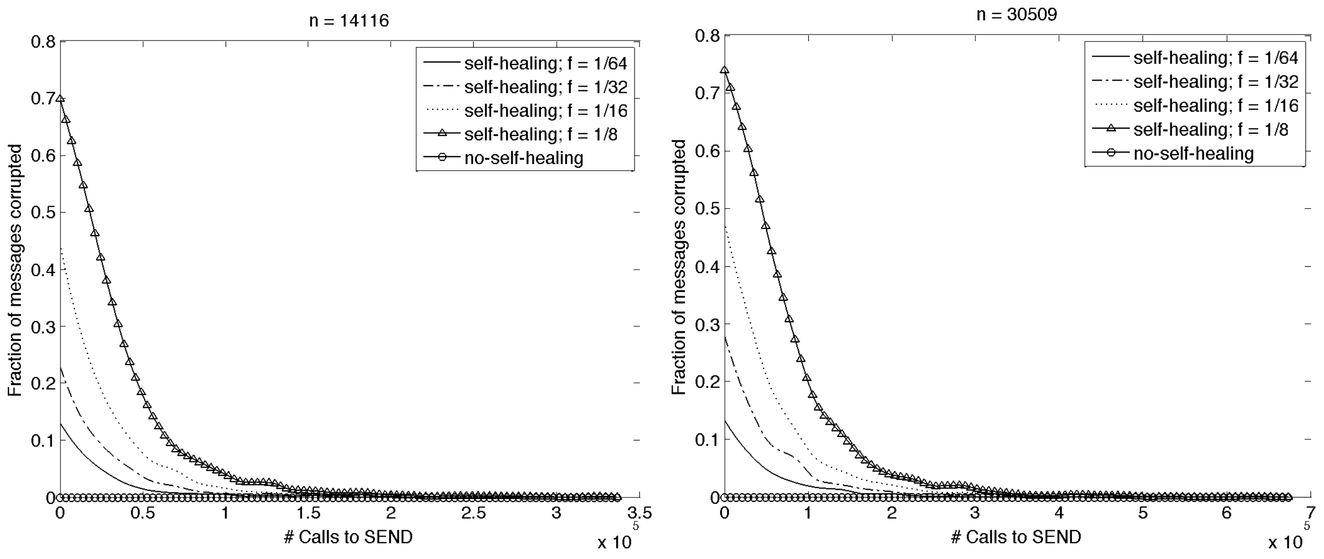


Fig. 14 Fraction of messages corrupted versus # calls to *SEND*, for $n = 14{,}116$ and $n = 30{,}509$, when *SEND* calls *CHECK2*.

message corruptions over rounds. Can we extend this algorithm to fit for asynchronous communication?

References

1. Boman, I., Saia, J., Abdallah, C., Schamiloglu, E.: Brief announcement: Self-healing algorithms for reconfigurable networks. Volume 4280 of SSS'06. (2006) 563–565
2. Saia, J., Trehan, A.: Picking up the pieces: Self-healing in reconfigurable networks. IPDPS'08 (2008) 1–12
3. Hayes, T., Rustagi, N., Saia, J., Trehan, A.: The forgiving tree: a self-healing distributed data structure. PODC'08 (2008) 203–212
4. Hayes, T.P., Saia, J., Trehan, A.: The forgiving graph: a distributed data structure for low stretch under adversarial attack. PODC'09 (2009) 121–130
5. Pandurangan, G., Trehan, A.: Xheal: localized self-healing using expanders. PODC'11 (2011) 301–310
6. Sarma, A.D., Trehan, A.: Edge-preserving self-healing: keeping network backbones densely connected. In: IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). (2012) 226–231
7. Fiat, A., Saia, J.: Censorship resistant peer-to-peer networks. Theory of Computing **3**(1) (2007) 1–23
8. Knockel, J., Saad, G., Saia, J.: Self-healing of Byzantine faults. SSS'13 (2013) 98–112
9. Hildrum, K., Kubiatowicz, J.: Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In: Distributed Computing. Volume 2848. (2003) 321–336
10. Naor, M., Wieder, U.: A simple fault tolerant distributed hash table. IPTPS'03 (2003) 88–97
11. Scheideler, C.: How to spread adversarial nodes? rotate! STOC'05 (2005) 704–713
12. Fiat, A., Saia, J., Young, M.: Making chord robust to Byzantine attacks. ESA'05 (2005) 803–814

13. Awerbuch, B., Scheideler, C.: Towards a scalable and robust DHT. Theory of Computing Systems **45**(2) (2009) 234–260
14. King, V., Lonargan, S., Saia, J., Trehan, A.: Load balanced scalable Byzantine agreement through quorum building, with full information. ICDCN'11 (2011) 203–214
15. Frisanco, T.: Optimal spare capacity design for various protection switching methods in ATM networks. Volume 1 of ICC'97. (1997) 293–298
16. Iraschko, R.R., MacGregor, M.H., Grover, W.D.: Optimal capacity placement for path restoration in STM or ATM mesh-survivable networks. IEEE/ACM Transactions on Networking **6**(3) (1998) 325–336
17. Murakami, K., Kim, H.S.: Comparative study on restoration schemes of survivable ATM networks. Volume 1 of INFOCOM'97. (1997) 345–352
18. Van Caenegem, B., Wauters, N., Demeester, P.: Spare capacity assignment for different restoration strategies in mesh survivable networks. Volume 1 of ICC'97. (1997) 288–292
19. Xiong, Y., Mason, L.G.: Restoration strategies and spare capacity requirements in self-healing ATM networks. IEEE/ACM Transactions on Networking **7**(1) (1999) 98–110
20. Saia, J., Young, M.: Reducing communication costs in robust peer-to-peer networks. Information Processing Letters **106**(4) (2008) 152–158
21. Young, M., Kate, A., Goldberg, I., Karsten, M.: Practical robust communication in DHTs tolerating a Byzantine adversary. ICDCS'10 (2010) 263–272
22. Datar, M.: Butterflies and peer-to-peer networks. Volume 2461 of ESA'02. (2002) 310–322
23. Young, M., Kate, A., Goldberg, I., Karsten, M.: Towards practical communication in Byzantine-resistant DHTs. IEEE/ACM Transactions on Networking **21**(1) (2013) 190 –203
24. Kate, A., Goldberg, I.: Distributed key generation for the internet. ICDCS'09 (2009) 119–128
25. Fiat, A., Saia, J.: Censorship resistant peer-to-peer content addressable networks. SODA'02 (2002) 94–103