Byzantine Reliable Broadcast in Sparse Networks

Sisi Duan
Oak Ridge National Laboratory
Email: duans@ornl.gov

Lucas Nicely
University of Tennessee, Knoxville
Email: lnicely@vols.utk.edu

Haibin Zhang
University of Connecticut
Email: haibin.zhang@uconn.edu

***Abstract*—Modern large-scale networks require the ability to withstand arbitrary failures (*i.e.,* Byzantine failures). Byzantine reliable broadcast algorithms can be used to reliably disseminate information in the presence of Byzantine failures.**

We design a novel Byzantine reliable broadcast protocol for loosely connected and synchronous networks. While previous such protocols all assume correct senders, our protocol is the first to handle Byzantine senders. To achieve this goal, we have developed new techniques for fault detection and fault tolerance. Our protocol is efficient, and under normal circumstances, no expensive public-key cryptographic operations are used.

We implement and evaluate our protocol, demonstrating that our protocol has high throughput and is superior to the existing protocols in uncivil executions.

***Index Terms*—Byzantine broadcast, reliable broadcast, fault detection, fault tolerance, sparse networks.**

I. Introduction

Byzantine failures occur when a participant in a distributed system deviates arbitrarily from the protocol specification, *e.g.,* due to a software bug or a cyber attack. Protocols detecting or tolerating such failures are particularly appealing for modern distributed systems and network applications that increasingly grow larger (*e.g.,* clouds, cryptocurrency systems).

We study how to efficiently achieve reliable broadcast in sparse networks that are subject to Byzantine failures. A sparse network is a network with a low number of links. Examples of sparse networks include sensor, robotic, and *most* real-world networks [19], [27].

Previous Byzantine broadcast protocols for sparse networks [23]–[25] assume that senders (*i.e.,* source nodes) who broadcast messages are *correct*, while a fraction of non-source nodes can be Byzantine. These protocols are essentially Byzantine variants of *best-effort broadcast* protocols, which guarantee that all correct processes deliver the same set of messages only if the senders are correct. If the sender is faulty or behaves maliciously, some nodes may deliver the message while others may not. Our goal is to build stronger broadcast protocols in the customary sense of *reliable broadcast*, one that handles Byzantine senders.

Before proceeding to our protocol properties, we will review and clarify models of distributed systems in terms of channel and graph connectivity.

Channel. When considering Byzantine failures, we need methods to provide correct sender identification. One may generally assume *authenticated channels*: if a correct process delivers a message with a correct sender, the message was previously sent by the sender. This model is also known as the *full Byzantine model*. Basing protocols on this model is desirable because it maximizes the fault models. One may choose either cryptographic techniques (such as message authentication codes (MACs) and digital signatures), or non-cryptographic techniques [33].

A number of works (*e.g.,* [23]–[25]) regard protocols designed in the full Byzantine model as "cryptography-free protocols." This view is slightly problematic, because authenticated channels using digital signatures, message authentication codes, or a combination of both (*e.g.,* SSL, TLS) dominate Internet communication. (But the view is correct in the sense that it does not rely on any *specific* cryptographic primitive such as digital signatures.)

It is more efficient to implement the protocols using authenticated channels with MACs because MACs are much less expensive than digital signatures. Therefore, while maximizing the number of fault models to which the protocols can be applied, it is desirable to design Byzantine resilient protocols assuming authenticated channels due to efficiency concerns. It is less convincing to argue that protocols in full Byzantine model "do not require a trusted infrastructure," because authentication requires an extensive setup—there must be an agreed upon setup for the authentication model.

Lastly, we must clarify four points. First, most Byzantine resilient protocols that explicitly use MACs, such as PBFT [9] "[c]an be modified easily to rely only on point-to-point authenticated channels," as commented in [9, pp. 402]. Second, it is unnecessarily true that all Byzantine resilient protocols using MACs also work in the full Byzantine model, because it is difficult, if not impossible, to transform a handful of protocols which use MACs in a more complex manner [2], [16], [35] into ones assuming only authenticated channels. Third, many protocols that claim to be "cryptography-free" are the most efficient cryptographic solutions in practice because they can be implemented simply using MACs. For instance, the broadcast protocol that tolerates Byzantine non-source nodes in sparse networks [25] leads to the most efficient MAC based protocol. Four, if MACs are inadequate for designing a cryptographic solution, another design choice (see, *e.g.,* [2], [21]) is to optimize the gracious execution (*i.e.,* the case without failures) and to use signatures only for uncivil executions (*i.e.,* the case with failures).

Graph connectivity. We briefly review the graph model for the Byzantine broadcast case. Most of these protocols are designed for completely connected graphs [6], [9], [22], which

978-1-5090-3216-7/16/31.00

attempt to tolerate a *maximum* number of Byzantine failures. Dolev [13] was the first to show that in order to tolerate k Byzantine failures, it is necessary and sufficient that the network is $(2k+1)$-connected, given that there is at least $3k+1$ nodes. Later, Nesterenko and Tixeuil [26] generalized the result in a manner that the topology is unknown and the environment is asynchronous.

Other groups have considered the *density* of Byzantine failures. These cases can be divided into two categories—those for dense networks [5], [20], [28] and those for sparse networks [23]–[25]. The protocols for dense networks are not adaptable for use in sparse networks, because if they were, the number failures that they can tolerate would be very small. In sparse networks, the failures are best measured as the distance between any two Byzantine nodes. A significant drawback of the protocols in sparse networks is that they *all* consider only non-faulty senders, and thereby are not secure in the *customary* sense of reliable broadcast. Assuming that the sender is always correct apparently limits the scope of deployment of these broadcast protocols in practice. Our primary goal is to build a reliable broadcast protocol that also tolerates Byzantine senders in an efficient manner.

Our contributions can be summarized as follows:

1) We present the first Byzantine reliable broadcast protocol in sparse networks that also handles Byzantine senders in synchronous environments. Our protocol is based on Maurer and Tixeuil (MT) [25], and has the following features:

- Our protocol provides multi-tiered security. Namely, when the networks are synchronous, it tolerates Byzantine senders; in settings where the networks may be asynchronous and senders are correct, our protocol still provides meaningful consistency guarantees.
- Our protocol is optimal in its gracious execution where there are no failures. Even in its uncivil execution, our protocol remains more efficient than the existing protocols with similar goals.

2) We develop novel techniques in both fault tolerance and fault detection, which are of independent interests and may be applicable to some other scenarios such as secure routing.

3) We implement and evaluate our protocol. Our experimental evaluation shows that our protocol has high throughput and high failure resilience.

II. Related Work

Byzantine broadcast: Additional related work. In the context of Byzantine failures, two classic broadcast notions are *consistent broadcast* and *reliable broadcast*. The notion of consistent broadcast was implicit in earlier papers on the topic [6], [7], [34]. Byzantine consistent broadcast ensures only that the *delivered requests* are the same for all receivers. Byzantine reliable broadcast, also known as the "Byzantine generals problem" [22], additionally guarantees that either all correct parties deliver some request or none delivers any. For instance, Bracha's broadcast [6], one that assumes only authenticated channels, is a well-known implementation of Byzantine reliable broadcast for complete graphs.

For sparse graphs, previous works [23]–[25] all assume correct senders. They fail to provide any meaningful reliability properties if senders become faulty: these protocols are not consistent broadcast, let alone reliable broadcast. One should be aware, however, that the problem of broadcast in the presence of Byzantine non-source nodes is still highly non-trivial, as Byzantine non-source nodes can disseminate fake messages and lie to the network.

Byzantine fault detection. Our protocol developed new techniques of Byzantine fault detection. In contrast to crash failures, Byzantine failures are *not* context-free, and therefore it is impossible to define a general failure detector in Byzantine environments, independently of the algorithm itself [15].

Almost all protocols for Byzantine fault detection (and fault diagnosis) [1], [18], [29]–[32], [37] use the idea of collecting a *proof of misbehavior* by executing modified Byzantine resilient protocols. However, the approach requires a (large) number of rounds and a huge volume of exchanged messages to collect the necessary information to provide such a proof. An adversary can easily render the system even less practical by intermittently following and violating the protocol specification. Similarly, PeerReview [17] can detect and deter failures by exploiting accountability. It also uses a "sufficient" number of witnesses to discover faulty replicas. An exception is BChain [16] which does not need to regularly collect evidence. However, BChain is an atomic broadcast protocol for a complete graph. The technique developed in our protocol deviates significantly from that of BChain.

Self-stabilizing Byzantine broadcast. Self-stabilization [14] is a powerful approach to obtaining correct behavior regardless of the consistency of initial states. Specifically in sparse networks, Maurer and Tixeuil [25] combine Byzantine tolerance and self-stabilization to deal with both a fraction of permanent Byzantine failures and an arbitrary number of transient Byzantine failures. This generalizes the traditional Byzantine broadcast. As our experimental evaluation shows, if the senders are Byzantine, the time needed to recover from transient failures may be prohibitively long. Our protocol, instead, can be well applied to this scenario, yielding a more robust and efficient protocol.

III. Preliminaries

We represent the network using an undirected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Let D be the network diameter (*i.e.,* the maximal distance between any two nodes). Let Δ be the maximum degree of G. We begin by describing several definitions.

Definition 1. (Path). A sequence of nodes $X = (p_1, p_2, \cdots, p_n)$ is a path if $\forall i \in \{1, 2, \cdots, n-1\}$, p_i and p_{i+1} are neighbors.

Definition 2. (Same source disjoint paths). Two paths $X = (p_1, p_2, \cdots, p_{n_1})$ and $X' = (q_1, q_2, \cdots, q_{n_2})$ are same source disjoint if $p_1 = q_1$ and $(X \backslash p_1) \cap (X' \backslash q_1) = \emptyset$; we write $\mathrm{DP}(p_1, X, X') = 1$ to denote two paths X and X' with the same source p_1. Generally, for n paths $X_1, X_2, \cdots, X_n$, if

they share the same source node p and every two paths are same source disjoint, we write DP$(p, X_1, \cdots, X_n) = 1$.

Definition 3. (Distance). The distance between two nodes p_i and p_j, denoted $distance(p_i, p_j)$, is the smallest number of edges between them.

The following definition is novel and vital to this work.

Definition 4. (Neighbor zone) Given an integer Z, the neighbor zone of a node p, denoted as NZ(p), is a set of connected nodes $\{p_1, \cdots, p_n\}$, where $p \in$ NZ(p) and $\forall i, j \in \{1, \cdots, n\}$ and $i \neq j$, $distance(p_i, p_j) \leq 2Z$.

We consider the problem of reliable broadcast in a sparse network that can be decomposed into cycles. We borrow the following definitions from MT [25].

Definition 5. (Cycle). A set of nodes is a cycle if there exists a path $X = (p_1, p_2, \cdots, p_n)$ that contains all the nodes in the set and p_1 and p_n are neighbors. The *diameter* of a cycle is $n/2$ if n is even, and $(n-1)/2$ if n is odd.

Definition 6. (Connected set of cycles). A set of cycles S is connected if, $\forall\{C, C'\} \subseteq S$, there exists a sequence of cycles $(C_1, \cdots, C_n)$ such that $C = C_1, C' = C_n$ and, $\forall i \in \{1, \cdots, n-1\}$, C_i and C_{i+1} have at least two nodes in common.

Definition 7. (Resilient decomposition). An arbitrary set of cycles S of the network is a resilient decomposition if, for each pair of nodes p and q, there exists a connected set $S(p, q) \subseteq S$ of at most Δ cycles such that 1) Each cycle contains p and not q; 2) Each neighbor of p (distinct from q) belongs to a cycle of $S(p, q)$.

Definition 8. (Z-resilient network). A network is Z-resilient if there exists an arbitrary set of cycles S of the network such that S is a resilient decomposition, and the diameter of the cycles of S is at most Z.

Definition 9. ((p, q)-valid set of sequences). Let Ω be a set of sequences, where a sequence $(p_1, \cdots, p_n)$ is a valid path and $n \leq 4D\Delta^2 Z$. Let $G(\Omega)$ represent a subgraph (V, E) of G such that: 1) V is the set of node identifiers in Ω; 2) For any sequence $(p_1, \cdots, p_n)$ and $i \in \{1, \cdots n\}$, there exists an edge in G such that $p = p_i$ and $q = p_{i+1}$. We say that Ω is (p, q)-valid if: 1) $\forall(p_1, \cdots, p_n) \in \Omega, p_1 = p$ and $p_n = q$; 2) The graph $G(\Omega)$ can be decomposed into a connected set of cycles $(C_1, \cdots, C_m)$ such that $\forall i \in \{1, \cdots, m\}$, C_i and C_{i+1} has at least two nodes in common.

IV. Our Protocol

System model. We assume a Z-resilient network where the minimum distance between any two Byzantine nodes is strictly greater than $2Z$. A key property is that in *any* neighbor zone of any sender, there is at most one faulty node. If, for instance, the sender is faulty, all the other nodes in its neighbor zone must be correct. We use both MACs and signatures, but signatures are only needed in case of failures. Let $\langle M \rangle_i$ denote an authenticated message for M using signatures, signed by a node p_i. Let $[M]$ denote an authenticated message for M using MACs.

Property 1. *Validity:* If a correct node broadcasts a message m, then every correct node eventually delivers m.
Property 2. *No Duplication:* No message is delivered more than once.
Property 3. *No Creation:* If a correct node delivers a message m with sender p_s, then m was previously broadcast by p_s.
Property 4. *Agreement:* If a message m is delivered by some correct nodes, then m is eventually delivered by every correct node.

Fig. 1: Byzantine reliable broadcast specification.

Definition of Byzantine reliable broadcast. A Byzantine reliable broadcast algorithm ensures that the correct nodes agree on the set of messages, even when the senders of these messages behave arbitrarily. It is characterized by the four properties in Fig. 1 (see also [8]).

```
00    on receiving m_0 = [s, m, (p_1, p_2, ..., p_n)]
01      if n ≥ Z then discard m_0                                  ⇐ [MT1]
02      if p_n is a neighbor of p_i then add m_0 to p_i.Rec, multicast m_0
03-1    pred ← ∃X, X' s.t. DP(p_s, X, X') = 1 for m                ⇐ [MT1]
03-2    pred ← ∃(p_s, p_i)-valid set for m                         ⇐ [MT2]
04      if pred then accept m from p_s, remove (s, m', X') from p_i.Rec
```

Fig. 2: The MT algorithms.

The underlying MT algorithms. MT [25] presented two simple and elegant Byzantine broadcast algorithms for sparse networks—MT1 (a broadcast protocol) and MT2 (a self-stabilizing broadcast protocol), as depicted in Fig. 2 in pseudocode. Neither of them can tolerate faulty senders.

For both algorithms, a sender p_s multicasts a message m to all its neighbors which then multicast the message to their neighbors, and so forth. When multicasting a message, each node appends its identity to the message and the nodes form a *travel path* X.

For MT1, a node delivers a message, if it receives a matching message from two disjoint paths and the number of nodes in each path does not exceed Z. MT2 removes the limit of Z nodes identifiers. A node q waits until it receives a same message from p_s with several different paths. If the fusion of the paths form a (p_s, q)-valid set of sequences, q accepts the message from p. As a self-stabilizing broadcast protocol, MT2 does not terminate: A node q that have already accepted any message from a node p_s, can accept another message from p_s, and so forth.

Overview. Our protocol trades network environments for stronger reliability, while preserving high throughput for both fault-free and failure scenarios. To handle sender equivocation, we combine fault detection and fault tolerance techniques. If a sender fails to send any message to some nodes in time,

it is convenient to regard that the sender sends an "empty" message ϵ that is different from any messages in the usual message space $\mathcal{M}$, i.e., we consider an extended message space $\mathcal{M} \cup \{\epsilon\}$. We divide sender equivocation into three types such that no matter what the graph connectivity is, any equivocation behavior falls into one of them:

Type I: The source node sends at least three different messages to three neighbors.
Type II: The source node sends only two different messages and each such message reaches at least two neighbors.
Type III: The source node sends only two different messages and one of the messages reaches only one neighbor.

10 *cond*:{$\exists(p_i, p_j, p_k) \in \{X_i, X_j, X_k\}$ & DP$(p_s, X_i, X_j, X_k) = 1$ & NZ(p_s, p_i, p_j, p_k)&$M.p_i \neq M.p_j \neq M.p_k$} **or** {$\exists(p_i, p_j, p_k, p_l) \in \{X_i, X_j, X_k, X_l\}$ & DP$(p_s, X_i, X_j, X_k, X_l) = 1$ & NZ$(p_s, p_i, p_j, p_k, p_l)$ &$M.p_i = M.p_j = m$ & $M.p_k \neq m$ & $M.p_l \neq m$}
20 **on receiving** $[\text{MSG}, p_s, m, X]$
21 **add** $[\text{MSG}, p_s, m, X]$ to $p_i.Rec$
22 **if** $p_i = \tilde{p}_s$&m is new **or** $p_i \neq \tilde{p}_s$ **then**
23 forward $[\text{MSG}, p_s, m, X]$ to neighbors
24 **if** m is new **then** start timer T_2
25 **if** m is new & $p_i = \tilde{p}_s$ **then** start timer T_1
26 **if** $X = p_s$ & $run(T_1)$ **then** cancel timer T_1
27 **if** $\exists(X_i, X_j) \in p_i.Rec$ & DP(p_s, X_i, X_j) & $M.X_i \neq M.X_j$ **then**
28 **send** $\langle \text{ALERT}, p_s, p_i, p_i.Rec\rangle_i$
30 **on receiving** $\langle \text{ACCUSE}, p_s, p_j, p_j.Rec\rangle_j$
31 **add** $\langle \text{ACCUSE}, p_s, p_j, p_j.Rec\rangle_j$ to $p_i.Acc(p_s)$
32 **if** $\exists(p_j, p_k) \in p_i.Acc(p_s)$ *s.t.* NZ(p_s, p_i, p_j) **then**
33 cancel timers, block p_s
34 **forward** $\langle \text{ACCUSE}, p_s, p_j, p_j.Rec\rangle_j$ and $\langle \text{ACCUSE}, p_s, p_k, p_k.Rec\rangle_k$
35 **if** *cond* **then send** $\langle \text{ACCUSE}, p_s, p_i, p_i.Rec\rangle_i$, cancel timers
40 **on receiving** $\langle \text{ALERT}, p_s, p_i, p_j.Rec\rangle_j$
41 **add** $\langle \text{ALERT}, p_s, p_i, p_j.Rec\rangle_j$ to $p_i.Ale$
42 **if** *cond* **then**
43 **send** $\langle \text{ACCUSE}, p_s, p_i, p_i.Rec\rangle_i$, cancel timers
44 remove $\langle \text{ALERT}\rangle$ from $p_i.Ale$ and $\langle \text{ACCUSE}\rangle$ from $p_i.Acc$
50 **on** timeout T_1 & $p_i = \tilde{p}_s$
51 add $[\text{MSG}, p_s, \epsilon, p_s]$ to $p_i.Rec$, **send** $\langle \text{ALERT}, p_s, p_i, p_i.Rec\rangle_i$
60 **on** timeout T_2
61 **if** $pred$ **then** deliver m, remove $[\text{MSG}, p_s, m, X]$ from $p_i.Rec$

Fig. 3: Our protocol.

Our protocol can be based on either MT1 or MT2, leading to a Byzantine reliable broadcast protocol or self-stabilizing broadcast protocol. Under normal circumstances, nodes run MT1 or MT2 with MACs based authenticated messages [MSG]. We use two types of signed messages—$\langle$ALERT$\rangle$ and $\langle$ACCUSE$\rangle$ to cope with failures. An $\langle$ALERT$\rangle$ message is triggered if a node receives mismatching messages from disjoint paths. However, a faulty node might issue an $\langle$ALERT$\rangle$ to frame the sender. A correct node thus needs to rely on $\langle$ALERT$\rangle$ messages *in the same neighbor zone* to rule out this possibility. After a node is certain that the source node is faulty, it generates an $\langle$ACCUSE$\rangle$ message. A node discards all the messages related to p_s if it already generates an $\langle$ACCUSE$\rangle$ message or receives two $\langle$ACCUSE$\rangle$ messages from nodes in the same neighbor zone.

Our protocol. The protocol is depicted in Fig. 3. Security of our protocol is *multi-tiered*: when the networks are synchronous, it satisfies the definitions in Fig. 1; when the networks are asynchronous but senders are correct, our protocol meets all the definitions except validity, in which case correct nodes still always agree on the same set of messages.

Let X be the travel path of the message, $M.p_i$ denote the message that p_i receives from a source node p_s, $M.X$ be the message p_i receives from a path X, $M.X_i = M.X_j$ be the case where the messages from path X_i and X_j are matching, $run(T_i)$ represent if the timer T_i has been started, and $\tilde{p}_s$ be a neighbor of p_s. Each node stores three sets of messages, including $p_i.Rec$ for the [MSG], $p_i.Ale$ for the $\langle$ALERT$\rangle$ messages, and $p_i.Acc$ for the $\langle$ACCUSE$\rangle$ messages.

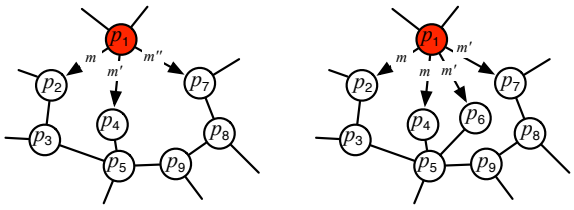
The algorithm proceeds as follows. A sender p_s multicasts a message $[\text{MSG}, p_s, m, X]$ to its neighbors. When a node p_i receives the message, it appends its id p_i to X and forwards the message to its neighbors, as shown in lines 22-23. Note that if a neighbor p_i receives a message from a source node p_s, it forwards the message only if the message is new. After receiving a [MSG], p_i may start two fixed timers T_1 and T_2. T_1 is used for the neighbors of p_s to monitor if they have received the [MSG] from p_s. The timer will be canceled if the neighbor of p_s receives the [MSG] from p_s. Instead, T_2 is a timer used for any nodes to see if a desired condition $pred$ can be met before the timer expires (see lines 60-61). Recall that we used the same notation $pred$ when we describe MT1 and MT2 in Fig. 2. If our protocol is instantiated using, say, MT1, then $pred$ is defined as in Fig. 2, *i.e.,* $pred$ returns 1 if a node receives matching messages from two disjoint paths.

Line 10 specifies a condition, which is used by nodes to verify if the source node is faulty based on their own $\langle$ALERT$\rangle$ and $\langle$ACCUSE$\rangle$ sets. The first and second OR clauses match Type I and Type II equivocation, respectively. The first clause aims to check if there exists at least three nodes in some NZ(p_s) such that their $\langle$ALERT$\rangle$ and $\langle$ACCUSE$\rangle$ sets contain three inconsistent messages. Note that there exists at most one faulty node in any neighbor zone of p_s. If p_s sent consistent messages to all its neighbors, it would be impossible that three of its neighbors (two of which must be correct) claim they received inconsistent messages. The second clause checks if there exists at least four nodes in some NZ(p_s) such that two of them received m while another two of them received a different message m'. Similarly, if the source node is correct, the condition will not be satisfied.

We do not need to worry about the Type III sender failures, because this type of failures are effectively masked by our protocol (an example coming shortly.)

When $cond$ is satisfied, a node sends a message $\langle \text{ACCUSE}, p_s, p_i, p_i.Rec\rangle_i$ to all the neighbors. When a node receives two $\langle$ACCUSE$\rangle$ messages from two nodes in the same neighbor zone, it can also confirm that the source node is faulty. From then on, it ignores any messages which are from

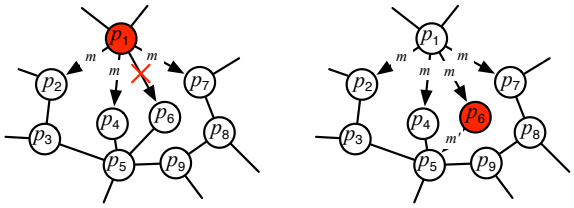
p_s and ⟨ALERT⟩ and ⟨ACCUSE⟩ messages related to p_s, and discards the corresponding ⟨ALERT⟩ and ⟨ACCUSE⟩ sets.



(a) An example for Type I equivocation. (b) An example for Type II equivocation.

Fig. 4: Examples for Type I and Type II failures.

Examples. Consider an example in Fig. 4 and suppose that all the nodes are in NZ(p_1). Fig. 4a illustrates Type I equivocation, where p_1 sends m to p_2, m' to p_4, and m'' to p_7. All the nodes will send an ⟨ALERT⟩ message, because they will all receive inconsistent messages from disjoint paths, and all the nodes will receive ⟨ALERT⟩ messages satisfying the first clause in *cond*. Thus, all the correct nodes can confirm that p_1 is faulty and generate an ⟨ACCUSE⟩ message. Fig. 4b illustrates Type II equivocation, where p_1 sends m to two of its neighbors and m' to two other neighbors. Likewise, all the nodes will receive conflicting messages and generate an ⟨ALERT⟩ message. For instance, p_2 receives m from $X = \{p_1\}$ and m' from the path $X = \{p_1, p_6, p_5, p_3\}$. After the nodes receive the ⟨ALERT⟩ messages, they can all verify *cond* and generate ⟨ACCUSE⟩ messages.



(a) The source p_1 is faulty. (b) Node p_6 is faulty.

Fig. 5: Example for a Type III failure.

We point out that it is "impossible" to detect a Type III faulty sender, if we consider the case where source nodes multicast messages without using signatures. The reason is that Type III is indistinguishable from the case with a faulty non-source node. Let's consider an example in Fig. 5. In Fig. 5a, we assume that the source node p_1 is faulty. It sends m to nodes p_2, p_4, and p_7, and sends m' to p_6. In Fig. 5b, we assume that p_1 is correct and but one of its neighbors p_6 is faulty. If p_6 changes m to m' and sends m' to other nodes, then no one can discover if conflicting messages are due to p_1 or p_6. Fortunately, this type of failures can be effectively tolerated by MT protocols and also our protocol.

V. CORRECTNESS PROOF

We begin by recalling several lemmas in MT. Lemma V.1 and Lemma V.2 are the properties of the Z-resilient networks, while Lemma V.3 are the properties of MT1 and MT2.

Lemma V.1. *Let p_s be a Byzantine node. Let p_i and p_j be two neighbors of p_s. There exists a correct path of at most $\alpha = \Delta Z$ hops connecting p_i and p_j.*

Lemma V.2. *Let p_i and p_j be two correct nodes. There exists a correct path of at most $\beta = 2D\Delta$ hops connecting p_i and p_j.*

Lemma V.3. *Let p_i and p_j be two non-neighbors correct nodes. Node p_j accepts message m from p_i within at most γ time and never accepts another message from p_i. For MT1, $\gamma = 8D\Delta^2 ZT$ and for MT2, $\gamma = 12D\Delta^2 ZT$, where T is the upper bound on the channel transmission time.*

Our protocol can be based on either MT1 or MT2. We focus on the case of MT1 and the other case is similar. The validity, no duplication, and no creation properties essentially follow from MT. The crux is to prove the correctness of the agreement property.

Theorem V.4. *(Agreement) If a message m is delivered by some correct nodes, then m is eventually delivered by every correct node.*

Proof. MT shows that agreement is satisfied if the sender is correct. We demonstrate that any sender equivocation behavior can be either *eventually* and *accurately* detected by all the correct nodes or tolerated by our protocol.

To this end, we prove the following claims in the rest of the section: 1) Type I and Type II failures can be eventually identified by all the nodes; 2) Correct nodes never accept any messages from senders who exhibit Type I or Type II failures; 3) If the sender is correct, it will never be accused by correct nodes; and 4) Type III failures do not introduce any inconsistency. The agreement property of our protocol will then easily follow from the above claims and the agreement property of MT. ■

Below we prove the four claims for Theorem V.4.

Theorem V.5. *Type I and Type II Byzantine senders can be always detected by at least two correct nodes in some neighbor zone. After at most $(3\alpha + \beta)T + T_1$ time, all the correct nodes learn that the sender p_s is faulty.*

Proof. We first prove in Lemma V.6 that Type I and Type II failures can be effectively detected by nodes in some neighbor zone. Then we show that eventually all the correct nodes learn the fact and we upper bound the time in Lemma V.7.

Lemma V.6. *Type I and Type II Byzantine senders can be always detected by at least two correct nodes in some neighbor zone NZ(p_s). Nodes in the network will all receive the ⟨ACCUSE⟩ messages.*

Proof. We first show that in the presence of Type I and Type II Byzantine senders, all the correct nodes in NZ(p_s) will generate ⟨ALERT⟩ messages and we distinguish two cases:

1) Assume that none of messages which the faulty source node sends is an empty message. We claim that correct nodes will receive conflicting messages from disjoint paths in NZ(p_s). Indeed, according to Lemma V.1, there exists a correct path

between any two correct nodes. Therefore, correct nodes will then generate ⟨ALERT⟩ messages.

2) Assume otherwise there exists at least one correct node (say, p_i) which the sender sent an empty message. Recall that if p_i learns a non-empty message from any other node in NZ(p_s), it starts a timer T_1. If p_1 does not receive any message from p_s before the timer expires, it sends an ⟨ALERT⟩ message. This additional step allows p_i to know if p_s sent some message to other neighbors but did not send any message to it. (The rest of the scenario is now the same as the above one.)

We now show that correct nodes in NZ(p_s) will generate ⟨ACCUSE⟩ messages. Note that ⟨ALERT⟩ messages contain the [MSG] received from p_s and their travel paths. For Type I senders, there exist three correct nodes whose ⟨ALERT⟩ messages contain three inconsistent messages from three disjoint paths in NZ(p_s). This will satisfy the first clause of $cond$ and these nodes will generate ⟨ACCUSE⟩ messages. Likewise, for Type II senders, there exist at least two correct nodes which generate ⟨ACCUSE⟩ messages. Since there exists a correct path between any two correct nodes, it is easy to see that all correct nodes in the network will receive ⟨ACCUSE⟩ messages. □

Lemma V.7. *All the correct nodes learn that p_s is faulty after at most $(3\alpha + \beta)T + T_1$ time.*

Proof. We consider the worst case where some nodes in NZ(p_s) need to wait for the timer T_1 to verify if the source node sends an empty message. In this case, a node p_i in NZ(p_s) needs at most ①αT time to learn that another node p_j in NZ(p_s) receive at least a message, say m (according to Lemma V.1), and then waits for ②T_1 time before p_i send an ⟨ALERT⟩ message. For those nodes in NZ(p_s) that have already received m, it takes another ③αT time for them to receive the ⟨ALERT⟩ message and generate their ⟨ALERT⟩ messages. After the nodes in NZ(p_s) generate ⟨ALERT⟩ messages, it takes ④αT time for the nodes to receive the ⟨ALERT⟩ messages and generate ⟨ACCUSE⟩ messages. For the rest of nodes in the graph, according to Lemma V.2, there exists a correct path of at most β hops connecting any two correct nodes, so it takes at most ⑤βT time for the rest of nodes to learn that p_s is faulty after receiving two ⟨ACCUSE⟩ messages. Notice that since T represents the transmission time in the network, T_1 can be set to T for a neighbor node to detect the failure of the source. Summing up ① to ⑤, the maximum time is $(3\alpha + \beta)T + T_1 = (3\alpha + \beta + 1)T$. As shown in Lemma V.3, nodes will not accept any other messages after γ time. Therefore, the timer T_2 can be set to be *max*(γ, $(3\alpha + \beta + 1)T$). This guarantees that before T_2 times out, all the correct nodes can receive ⟨ACCUSE⟩ messages. □

The theorem now follows. ■

Theorem V.8. *Correct nodes never accept any messages from senders who exhibit Type I or Type II failures.*

Proof. Setting the timer T_2 as the upper bound in the above lemma, the theorem trivially follows, as our protocol requires each node to wait for T_2 time to deliver the messages. ■

Theorem V.9. *If the source p_s is correct, it will never be accused by correct nodes.*

Proof. Suppose a correct node accused a source node p_s. According to our protocol, this node either received messages that satisfy $cond$ (type I) or received two ⟨ACCUSE⟩ messages from nodes in the same neighbor zone (type II). For type II, since there is one faulty in NZ(p_s), one of the two nodes that sent ⟨ACCUSE⟩ messages must be correct. This indicates that the correct node either received messages that satisfy $cond$, or received two ⟨ACCUSE⟩ messages from some other nodes in the zone. Inductively, we can prove that for type II, there exists some correct node that received messages that satisfy $cond$. Therefore, in both cases, there exists some correct node that received messages that satisfy $cond$ and for this reason it sent an ⟨ACCUSE⟩ message.

For this correct node, if the first clause of $cond$ was satisfied, there exist three nodes whose ⟨ALERT⟩ messages contain three inconsistent messages from three disjoint paths in NZ(p_s). In this case, either p_s sent three inconsistent messages or at least two of them are faulty. As there is at most one faulty node in each NZ(p_s), we know that p_s is faulty. On the other hand, if the second clause of $cond$ was satisfeid, either p_s sent two messages and each of them reached at least two nodes or at least two of them are faulty. Likewise, we can conclude that p_s is faulty. The theorem now follows. ■

Theorem V.10. *Type III failures do not introduce inconsistency.*

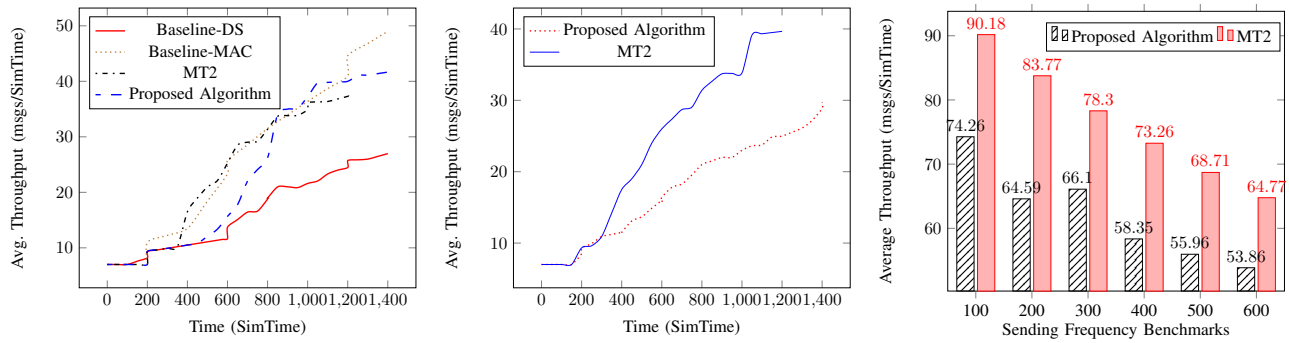
Proof. If p_s only sends a message m' to one neighbor and m to all other neighbors, a correct node will not receive a message from two disjoint paths. This is because there is at most one correct node in NZ(p_s) and the message is sent to at least two neighbors so that a node can accept m'. ■

VI. Implementation and Evaluation

We used the OMNeT++ Discrete Event Simulator [36] to model our algorithm. Each simulation begins with an initialization stage, during which the nodes send initialization messages in order to set up the network topology and to detect their neighbors.

We implemented the following algorithms and compared them with our algorithm. 1) Baseline-MAC: A MAC-based multicast protocol where nodes use MACs for message authentication in authenticated channels; 2) Baseline-DS: A digital signature-based multicast protocol, where nodes use digital signatures for message authentication; 3) MT algorithms.

For Baseline-MAC and Baseline-DS, we implemented the conventional gossip algorithm. Namely, both protocols are based on a multicast protocol, where a source node will send a message to each of its neighbors, which in turn will forward that message to each of their neighbors until all nodes have received the message. Baseline-MAC is efficient but does not provide meaningful fault tolerant guarantees. Instead, Baseline-DS is more robust and can detect sender equivocation, but still it fails to handle cases such as sender crashes.



(a) Faulty source node detection overhead. (b) Throughput for faulty non-source nodes. (c) Throughput of various benchmarks.

Fig. 6: Protocol evaluation.

We utilized HMAC [3] and RSA-FDH [4] to implement the underlying MAC and digital signature, respectively.

We implemented our protocol on top of MT1 and compare with MT2 in failure scenarios. Indeed, both our protocol and MT2 can deal with faulty senders, though via very different perspectives and with different properties. We implemented a failure injection mechanism where a number of random nodes are selected during each simulation. The faulty nodes can be further specified as either faulty source nodes or faulty non-source nodes or both. If a node is the source node, it simply equivocates to the neighbors by generating inconsistent messages with the same timestamp. If a node is a non-source node, it tampers with or falsifies the content of the messages and forwards to the neighbors.

We generate a random cyclic topology with a minimum of 3 nodes in each cycle. We use several benchmarks to evaluate a cyclic network with different traffic. For the x benchmark, each node multicasts a message to its neighbors every x ms. We evaluate throughput as the number of messages per simulation time (SimTime), and delivery rate as the percentage of messages received versus messages sent. Messages sent by the source node to the network are considered meaningful messages. All other invalid messages that correct nodes will not deliver are considered meaningless.

Faulty source node detection overhead. We compare failure-free cases for the Baseline-MAC and Baseline-DS algorithms with the faulty source node case for our algorithm. We compare Baseline-MAC and our algorithm to evaluate the overhead for our algorithm. Although Baseline-MAC does not handle any failures, we can compare Baseline-MAC in the failure-free case and our algorithm in the failure case to evaluate the overhead of our faulty source node detection algorithm. We then compare Baseline-DS and our algorithm to show our algorithm's efficiency in utilizing digital signatures, where in Baseline-DS, nodes are able to detect a faulty source node due to receiving mismatched messages.

We evaluate the three algorithms using a 200 benchmark and a single faulty source node. The benchmark determines the frequency that a faulty node sends false messages. Fig. 6a shows that our algorithm has a consistently higher throughput than Baseline-DS. This is caused by the fact that digital signatures are more expensive than MACs and our algorithm uses digital signatures when failures occur. The introduction of ⟨ALERT⟩ and ⟨ACCUSE⟩ messages in the presence of the faulty node also introduces more network traffic. Our algorithm achieves similar throughput, however, with Baseline-MAC. This shows that in the case where there are fewer failures, our algorithm generates low overhead for failure detection.

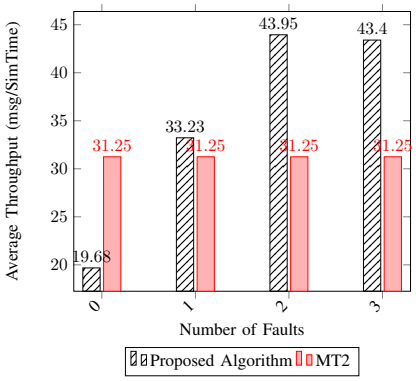


Fig. 7: Average throughput vs. number of faults.

Throughput. We compare MT and our algorithm in both the case where source nodes are faulty and non-source nodes are faulty. We use 200 benchmark and one failure in the network. Fig. 6a shows the case where the source node is faulty. Our algorithm achieves lower throughput than MT in the beginning and higher throughput later in the experiment. This is due to the ⟨ALERT⟩ and ⟨ACCUSE⟩ messages that are introduced into the network. As the faulty nodes begin to send false messages with an increased frequency, more ⟨ALERT⟩ and ⟨ACCUSE⟩ messages are generated. The lower the frequency, the faster we can confirm that there is a failure which results in a lower throughput initially. We also compare the performance where non-source nodes are faulty. Notice that the non-source nodes can "frame" correct source nodes by altering the content of once correct messages. It can be observed in Fig. 6b that our algorithm achieves lower throughput in this situation.

Additionally, we evaluate the case with various benchmarks that represent faulty node sending frequency. We randomly select 3 faulty nodes in a network of 12 nodes. As shown in Fig. 6c, the average throughput of our algorithm in each

benchmark is lower than MT. This is due to there being more false messages in the network as the frequency increases. Our algorithm detects the faulty node and alerts other nodes, which in turn do not forward any of the false messages. This results in a lower average throughput. We later test the case where nodes behave as faulty source nodes and multicast false messages. As shown in Fig. 7, our algorithm has a higher throughput in the presence of failures. The throughput is lower when there are no failures because our algorithm does not repetitively send the same message. We observe that our algorithm allows nodes perform more efficiently after a failure has been detected.

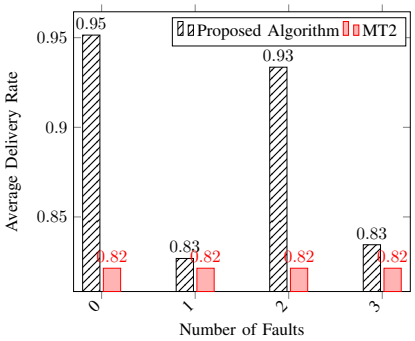


Fig. 8: Delivery rate vs. number of faults.

Delivery rate. As shown in Fig. 8, our algorithm consistently achieves a higher delivery rate than MT. This is because any decrease in delivery rate caused by a faulty node is balanced by the introduction of ⟨ALERT⟩ and ⟨ACCUSE⟩ messages.

VII. Conclusion

We presented the first Byzantine reliable broadcast protocol for sparse networks that can tolerate Byzantine senders in synchronous environments. We developed new techniques for fault detection. Our protocol is efficient for both fault-free and failure scenarios; in particular within gracious executions no public key cryptographic operations are needed. Finally, we implemented and evaluated our protocol. Our experimental evaluation shows that our protocol has high throughput and high failure resilience.

VIII. Acknowledgments

Sisi Duan was supported in part by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the Department of Energy. Lucas Nicely was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internship program. Haibin was supported in part by NSF grant CNS-1413996 for the MACS project.

References

[1] J. Adams and K. Ramarao. Distributed diagnosis of Byzantine processors and links. *ICDCS*, pp. 562–569, 1989.
[2] P-L. Aublin, R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. *TOCS*, vol. 32, issue 4, 2015.
[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *CRYPTO 1996.*
[4] M. Bellare and P. Rogaway. The exact security of digital signatures — How to sign with RSA and Rabin. *EUROCRYPT 1996*, pp. 399–416.
[5] V. Bhandari and N. Vaidya. On reliable broadcast in a radio network. *PODC*, pp. 138–147, 2005.
[6] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation 75*, pp. 130–143, 1987.
[7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM* 32(4), 824–840, 1985.
[8] C. Cachin, R. Guerraoui, and L. Rodrigues. Introduction to reliable and secure distributed programming. Springer, 2011.
[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. *ACM Trans. Comput. Syst*, 20(4): 398–461, 2002.
[10] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.
[11] S. Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. *PODC*, pp. 311–324, 1990.
[12] S. Chaudhuri, M. Herlihy, N. A. Lynch, and M. R. Tuttle. A tight lower bound for k-set agreement. *FOCS*, 1993.
[13] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.
[14] S. Dolev. Self-Stabilization. MIT Press, 2000.
[15] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. *EDCC*, 1999.
[16] S. Duan, H. Meling, S. Peisert, and H. Zhang. BChain: Byzantine replication with high throughput and embedded reconfiguration. *OPODIS 2014.*
[17] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. *SOSP*, pp. 175–188, ACM, 2007.
[18] H. Hsiao, Y. Chin, and W. Yang. Reaching fault diagnosis agreement under a hybrid fault model. *IEEE Trans. on Computers*, vol. 49, no. 9, 2000.
[19] M. Humphries and K. Gurney. Network 'small-world-ness': A quantitative method for determining canonical network equivalence. *PLoS One* 3(4): e2051, 2008.
[20] C.-Y. Koo. Broadcast in radio networks tolerating Byzantine adversarial behavior. *PODC*, pp. 275–282, ACM, 2004.
[21] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. *ICALP 2005*, pp. 204–215, 2005.
[22] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
[23] A. Maurer and S. Tixeuil. Limiting Byzantine influence in multihop asynchronous networks. *ICDCS*, pp. 183–192, 2012.
[24] A. Maurer and S. Tixeuil. On Byzantine broadcast in loosely connected networks. In *DISC*, pp. 253–266, 2012.
[25] A. Maurer and S. Tixeuil. Self-stabilizing Byzantine broadcast. *SRDS*, 2014.
[26] M. Nesterenko and S. Tixeuil. Discovering network topology in the presence of Byzantine nodes. *IEEE TPDS*, 20(12):1777–1789, 2009.
[27] M. Newman. The structure and function of complex networks. *SIAM Review* 45, pp. 67–256, 2003.
[28] A. Pelc and D. Peleg. Broadcasting with locally bounded Byzantine faults. *Inf. Process. Lett.*, 93(3):109–115, 2005.
[29] F. Preperata, G. Metze, and R. Chien. On the connection asssignment problem of diagnosable systems. *IEEE Trans. on Elec. Comp.*, 16(6): 848–854, 1967.
[30] K. Ramarao and J. Adams. On the diagnosis of Byzantine faults. *Proc. Symp. Reliable Distributed Systems*, pp. 144–153, 1988.
[31] M. Serafini, A. Bondavalli, and N. Suri. Online diagnosis and recovery: on the choice and impact of tuning parameters. *IEEE TDSC*, 4(4): 295–312, 2007.
[32] K. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. *Proc. Symp. Fault-Tolerant Computing*, pp. 55–60, July 1987.
[33] G. J. Simmons. A survey of information authentication. *Contemporary Cryptology, The Science of Information Integrity*, IEEE Press, 1999.
[34] S. Toueg. Randomized Byzantine agreements. *PODC 1984*, 1984.
[35] R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. *OPODIS 2012.*
[36] A. Varge. OMNeT++. In *Modeling and Tools for Network Simulation*, 91–104, 2010.
[37] C. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Trans. Software Eng*, 23(11): 684–721, 1997.