Reliable Broadcast in Practical Networks: Algorithm and Evaluation

Yingjian Wu

Computer Science Department

Boston College

Supervisor: Lewis Tseng

In partial fulfillment of the requirements of the degree of the Bachelor of Arts in Computer Science

May 15th 2020

Abstract

Reliable broadcast is an important primitive to ensure that a source can reliably broadcast a message to all the non-faulty nodes in either a synchronous or asynchronous network. This network system can also be failure prone, meaning packets can be dropped or the packets can be corrupted. The faulty server can perform different faulty behaviour such as sending wrong messages, and not sending any message. In 1987, Bracha first proposed reliable broadcast protocols, and since then different reliable broadcast protocols had been designed in order to achieve different goals, such as reducing round and bit complexity.

In a practical network, there are several constraints such as limited bandwidth or high latency. Thus We aim to design new reliable broadcast protocols that consider these practical network constraints. More specifically, we use cryptographic hash functions and erasure coding to reduce communication and computation complexity.

Finally, We also designed a general benchmark framework that can be used to test reliable broadcast algorithms. We evaluated new algorithms we have designed and implemented using this benchmark platform. and the algorithms showed superior performance in practical networks.

Contents:

1. Introduction:

We consider the reliable broadcast problem in an asynchronous network system with n number of servers (n-f number of non-faulty server, f faulty server) with each message size having L bits. Some properties need to be satisfied by a reliable broadcast algorithm. First, if the source is non-faulty, then all non-faulty nodes eventually deliver the same message, which is broadcasted by the non-faulty source. Second, if the source is faulty, all the non-faulty nodes should either not deliver or deliver the same message.

Since the proposal of Brach's reliable broadcast algorithm [3], many broadcast algorithms [3,4,5] have been proposed to reduce computation, rounds and bits complexity. However, based on our knowledge, their results are only proved in a theoretical perspective and are not tested in a practical network system assuming computational power and bandwidth constraints. In order to study how practical system constraints affect reliable broadcast algorithms, we implemented several broadcast algorithms from previous papers as the baseline for the new algorithms we have designed.

In order to carefully and thoroughly benchmark reliable broadcast algorithms, We have designed a benchmark tool on top of Mininet [8]. Mininet allows us to manipulate the networks such as bandwidth constraint, cpu computation for each node, and network topology. The goal of our paper is to understand the performance of reliable broadcast protocols in a practical asynchronous network system. Basically, a reliable broadcast protocol is a problem in distributed computing, focusing on sending messages to different processes or servers under the presence of nodes having byzantine behaviours such as send the wrong message or stop failure.

Motivation:

The following are our observations when trying to apply previous fault tolerant RB protocols in practice:

1. Existing reliable broadcast algorithms are not efficient in terms of bandwidth usage and/or computation (Table1)

2. Most Reliable Broadcast protocols assume unlimited bandwidth, and use flood mechanisms to send redundant messages over the network. (Table 1)
3. In many real world applications, source may not reside in the same system as other nodes. In such a case, bandwidth limitation is often much higher between source and other nodes compared to nodes in the same network system. One of the examples is that the source can be a client, and the rest of the nodes are in the same data center with an optimized network system.

Main Contributions:

First, we proposed a family of reliable broadcast algorithms.

1. Crash-tolerant erasure-based reliable broadcast
2. Byzantine hash reliable broadcast
3. Byzantine erasure code reliable broadcast

Table 1 provides a summary of both theoretical and practical results of different reliable broadcast algorithms. The bottleneck below indicates the scenario that may throttle the performance of the individual algorithm. We used cryptographic hash for our hash reliable broadcast and [n,k] MDS-code for erasure based algorithms. MDS-code known as maximum distance separable code, where n represents individual block length and k represents the dimension of the codeword.

Our algorithms have bit complexity $O(nL + nfL)$. When source is non-faulty, then our algorithms achieve $O(nL)$. However, when source can equivocate, the bit complexity becomes $O(nL + nfL)$. Furthermore, our algorithm when using erasure code achieves $O(nL / k)$ bits complexity between source node and others. Since most practical systems have a small number of faulty servers, our algorithms perform well in practical systems.

Table 1:

Algorithm	Bit Complexity	System Size(Resilience)	Round Complexity	Error-free	Uses MDS codes	Bottleneck
CRB[7]	O(n^2L)	> f	1	Yes	No	
EC-CRB	O(n^2L/k)	> f	2	Yes	Yes	MDS-Code
Bracha[3]	O(n^2L)	> 3f	3	Yes	No	flooding
Raynal[5]	O(n^2L)	> 3f	2	Yes	No	flooding
Patra[4]	O(nL)	> 3f	8	Yes	Yes	local computation
Hash-BRB[3f+1]	O(nL)+O(nfL)	> 3f	3	No	No	Hash Fun
Hash-BRB[5f+1]	O(nL)+O(nfL)	> 5f	2	No	No	Hash Fun
EC-BRB[3f+1]	O(nL)+O(nfL)	> 3f	3	No	Yes	Hash Fun + Coding
EC-BRB[4F+1]	O(nL)+O(nfL)	> 4f	3	No	Yes	Hash + Coding

Second, We built a benchmark platform called Reliably-Mininet-Benchmark (RMB) specifically to benchmark the performance of broadcast algorithms. Future developers can use this platform easily as they only need to write their own protocols without worrying about benchmark calculation and network settings.

Special Recognition:

Sapta Kumar analyzes the theoretical performance for all algorithms in table1 and proofread the proof of the new Algorithms.

Haochen (Roger) Pan helps designing reliable broadcast protocols, formulate proof and mininet setup.

Prof. Tseng leads the project and designs reliable broadcast protocols.

Organization of the Paper:

Section 2: System Models, notations and problem specification

Section 3: Hash-Based Reliable Broadcast protocol

Section 4: Coded-Based Reliable Broadcast protocol

Section 5: Design of our benchmark tool (RMB) and benchmark result.

2 PRELIMINARIES

2.1 Model and Notation

Our network system models [7,9] contain a number of nodes, which are connected to all the other nodes inside the network. Furthermore, this is an asynchronous system, which can a maximum of f number of faulty nodes.

Network: “Asynchronous” means that individual nodes have their own clock instead of a uniform clock. In our network assumption, we assume reliable channels meaning no packet should be dropped. Our protocol also ensures authentication, which is that individual nodes can have the access of the sender of a packet. In an asynchronous network, the delay for a packet varies, meaning sometimes a packet can take a really long time to arrive at the final destination. However, if a sender is non-faulty, then its message will be eventually received by other nodes inside the network.

Fault Model: Our network system tolerates up to f number of faulty nodes. A Byzantine node can perform arbitrary behavior such as not sending a packet, shutting itself down, or sending wrong messages to other nodes inside the network. Furthermore, our protocol also allows the source nodes to be faulty so a faulty source node can even equivocate the messages.To equivocate a message, a faulty source will send a message m1 to a partition of nodes and m2 to another partition of nodes, or can send nothing to the third partition of nodes.

Notations: For each message m that a non-faulty source wants all non-faulty nodes to reliably accept, m is associated with a tuple (s,h), where s is the identifier of the sender and h is the h-th instance of the reliable broadcast. In all of our algorithms, we use MsgSet_i [*s*, h] to denote the set of messages that the node *i* collects, in which are candidates that can be identified with (*s*, h). When the context is clear, we omit the subscript *i*. We use Counter[∗] to denote a local counter of certain type of messages that is initialized to 0. We use H(∗) to denote the cryptographic hash function.

2.2 Reliable Broadcast Properties

We adopted the definition and properties of reliable broadcast from **[**1,3,4**]**.

Reliable-Broadcast(m,h): a source reliably broadcast a message at round h.

Reliable-Accept(m,h): When a non-faulty node has received enough number of certain types of messages, the server would relable-accept this message, meaning for the hth round, message m is the one to be accepted.

A reliable broadcast protocol is correct if it satisfies the following properties.

1. (Non-faulty Broadcast Termination). If a non-faulty source *s* with a message *m* of index h performs Reliable-Broadcast(*m*, h), then all non-faulty nodes will eventually Reliable-Accept(*s*, *m*, h).

2. (Validity). If a non-faulty source *s* does not perform Reliable-Broadcast(*m*, h) then no non-faulty node will ever perform Reliable-Accept(*s*, *m*, h).

3. (Agreement). If a non-faulty node performs Reliable- Accept(*s*, *m*, h) and another non-faulty node will eventually perform Reliable-Accept(*s*,*m′*,h) then *m* =*m′*.

4. (Integrity). Anon-faultynodereliablyacceptsatmost one message of index h from a source *s*.

5. (EventualTermination). If a non-faulty node performs Reliable-Accept(*s*, *m*, h), then all non-faulty nodes eventually per- form Reliable-Accept(*s*, *m*, h).

Note that it is possible when a faulty source node broadcasts a message, non-faulty nodes will never accept the message, and this is different from synchronous systems, where each non-faulty node needs to output a value.

2.3 Five types of Messages:

As mentioned before, individual nodes inside the network when receiving different types of messages need to do certain action. These different types of messages are required for the correctness of our algorithm.

MSG**:** This is the message directly received from the source, and if the message sender is different from the source, then the server should never accept the message.

ECHO**:** This message tells individual servers what messages other servers have received from other servers. Note that we use broadcast H(m) here instead of m in order to reduce bit complexity.

ACC**:** This message tells other servers that one server is ready to accept a message. Again, the server sends H(m) instead of m throughout the process.

REQ**:** Sometimes, due to network delay or faulty source equivocate, a server may never receive the real message. Thus it needs to send a REQ to other servers to get the real message.

FWD**:** This message helps servers who have sent a REQ message, meaning the REQ senders have not received the real message, to get the information of the real message.

3. Hash Reliable Broadcast:

Below, We presented the first algorithm using a cryptographic hash function. Indicated by the name as “3f+1”, this means that in order to ensure the reliable-broadcast properties, we need n >= 3f + 1.

Hash Function: One of the assumptions we have made for our hash function is that it is collision free. However, this is not really practical in that if one server has unlimited computed power, it can construct a faulty message say m’, such that H(m) = H(m’). Even though hash function has such limitations, it has still been a widely adopted technique. For example, Bitcoin [10] has adopted the hash function for the miners when the miners try to solve puzzles. Thus, hash functions could also be applied under our reliable broadcast context. Since all the nodes run the same hash function, If a hash function is chosen appropriately, computation of a collision message takes time. Furthermore, if the broadcast process is fast, finding a collision message before reliable-accept in the same round is nearly impossible in practical network settings

3.1: Hash-BRB[3f+1]

3.1.2 Pseudo code:

Algorithm 1 Hash-BRB[3f+1]: source s with message m of index h

```
function RELIABLE-BROADCAST($m, h$)
    SEND(MSG, $s, m, h$) to all nodes
```

Algorithm 2 Hash-BRB[3f+1]: all node i (including s) when receiving a message from node j

```
function RECEIVING(MSG, $s, m, h$)
    if $j = s$ and first (MSG, $s, *, h$) then
        MsgSet[$s, h$] $\leftarrow$ MsgSet[$s, h$] $\cup \{m\}$
        Counter[ECHO, $s, \mathbb{H}(m), h$] ++
        if never sent (ECHO, $s, *, h$) then
            SEND (ECHO, $s, \mathbb{H}(m), h$) to all nodes
function RECEIVING(ECHO, $s, H, h$)
    if first (ECHO, $s, *, h$) from $j$ then
        Counter[ECHO, $s, H, h$] ++
        CHECK($s, H, h$)
function RECEIVING(ACC, $s, H, h$)
    if first (ACC, $s, *, h$) from $j$ then
        Counter[ACC, $s, H, h$] ++
        if Counter[ACC, $s, H, h$] = $f + 1$ then
            if $\nexists m' \in$ MsgSet[$s, h$] s.t. $\mathbb{H}(m') = H$ then
                SEND (REQ, $s, H, h$) to these $f + 1$ nodes
        CHECK($j, H, h$)
function RECEIVING(REQ, $s, H, h$)
    if first (REQ, $s, h$) from $j$ then
        if $\exists m' \in$ MsgSet[$s, h$] s.t. $\mathbb{H}(m') = H$ then
            SEND (FWD, $s, m', h$) to $j$
function RECEIVING(FWD, $s, m, h$)
    if have sent (REQ, $s, \mathbb{H}(m), h$) to $j$ then
        if first (FWD, $s, m, h$) from $j$ then
            MsgSet[$s, h$] $\leftarrow$ MsgSet[$s, h$] $\cup \{m\}$
            CHECK($s, \mathbb{H}(m), h$)
```

Algorithm 3 Hash-BRB[3f+1]: helper function for all node i (including s)

```
function CHECK($s, H, h$)
    if $m \in$ MsgSet[$s, h$] s.t. $\mathbb{H}(m) = H$ then
        if Counter[ECHO, $s, H, h$] $\geq f + 1$ then
            if never sent (ECHO, $s, *, h$) then
                SEND (ECHO, $s, H, h$) to all nodes
        if Counter[ECHO, $s, H, h$] $\geq n - f$ then
            if never sent (ACC, $s, *, h$) then
                SEND (ACC, $s, H, h$) to all nodes
        if Counter[ACC, $s, H, h$] $\geq f + 1$ then
            if never sent (ACC, $s, *, h$) then
                SEND (ACC, $s, H, h$) to all nodes
        if Counter[ACC, $s, H, h$] $\geq n - f$ then
            RELIABLE-ACCEPT($s, m, h$)
```

3.1.3 Proof of the correctness

In this section we are going to prove that Hash-BRB[3f+1] satisfied Property 1 - 5 of reliable broadcast mentioned in 2.2.

We begin with three important lemmas, the first two lemmas follow directly from the reliable and authenticated channel.

Lemma1. If a non-faulty source *s* performs Reliable-Broadcast(*m*,h), then $\text{MsgSet}_i\ [s, h] \subseteq \{m\}$ at each non-faulty node *i*.

Lemma2. If a non-faulty node *s* never performs Reliable-Broadcast (*m*, h), then $\text{MsgSet}_i[s, h] = \varnothing$ at each non-faulty node *i*.

Lemma3. If two non-faulty nodes *i* and *j* send(ACC,*s*,H(*m*),h) and (ACC, *s*, H(*m′*), h) messages, respectively, then *m* = *m′*.

Proof Property 1 - 4: Here we are going to prove with contradiction. Supposed server i is the first server to send (ACC, s, H(m), h) and server j is the first server to send (ACC, s H(m'), h) messages. Since both of them are the first servers to send their corresponding ACC message, server i and j send the ACC through line 8 in Algorithm 3.

Now let's first look at server i. According to line 6, server i must have collected at least n - f echo messages and at least n - f - f >= f + 1 are from non-faulty servers. This means that server j can at most collect 2f Echo messages that are corresponding to message m'. However, since individual servers can only send one type of message in each round, it is impossible for server j to send Acc in line 8. Thus we have a contradiction.

Thus from the above proof, we show that Property 1-4 are satisfied.

Now let's look at property 5, which is the guarantee of eventual termination.

Proof Property 5: Supposed server i has reliably accepted the message (line 13, Algorithm 3). Then it has received at least n - f messages from other servers. Among these messages at least n - f - f >= f + 1 messages are from non-faulty servers. These f + 1 messages will eventually be received by all the non-faulty nodes. Supposed that a server never gets the original message directly from the source, then it will get the original message by sending REQ messages to those f + 1 servers (line 14 Algorithm 2). After eventually receiving these f + 1 Acc messages from the non-faulty servers, the remaining non-faulty servers will also broadcast the same ACC message as server i. Thus eventually all the non-faulty servers will receive at least n - f number of ACC messages, which implies that all the non-faulty servers will reliably accept message m (line 13, Algorithm 3).

3.2 Hash-BRB[5f+1]

Inspired by a recent paper [5] that sacrifices resilience for lower message and round complexity, we adapt Hash-BRB[3f+1] in a similar way. More specifically, we can get rid of the ACC Phase. We need 5f + 1 in order to ensure that at least n - 2f > 3f non faulty nodes have received the same message, basically an idea of majority quorum. By increasing n to be at least 5 f + 1, this algorithm successfully reduces the round complexity to 2.

4. Erasure Code Reliable Broadcast:

One of the drawbacks in Hash-BRB[3f+1] is that the bit complexity is still high. For example, when a source broadcasts a message, the bit complexity is O(nl). Thus, instead of using hash, We tried to use Erasure Code to further reduce bit complexity.

4.1: MDS Erasure Code Preliminaries

For completeness, we first discuss basic concepts and notations from coding theory. We use linear [n,k] MDS (Maximum Distance Separable) erasure code over a finite field Fq to encode the message. n represents the total number of codes we need, and k represents the total number of elements needed to decode back a message.

When encoding operation is performed, m is first divided into k pieces with each size L/k, which then will output n coded elements.

At the beginning of the algorithm, the source broadcasts one unique message to each server. We use ENC_i to denote the coded element sending to the ith server, where $1 <= i <= n$.

4.2 EC-BRB[3f+1]

This algorithm requires n >= 3f + 1 in order to ensure the reliable broadcast properties. One of the downsides of this algorithm is that this algorithm requires exponential calculation when permuting all the possible coded elements. The complexity is $O(\binom{n}{f+1})$. When f is small, the computation complexity is minimal but as f increases, this algorithm does not scale.

Notice that between algorithm 12 and 16, it is possible that one server decodes faulty messages and includes them into the message set.

4.2.1 Pseudo Code

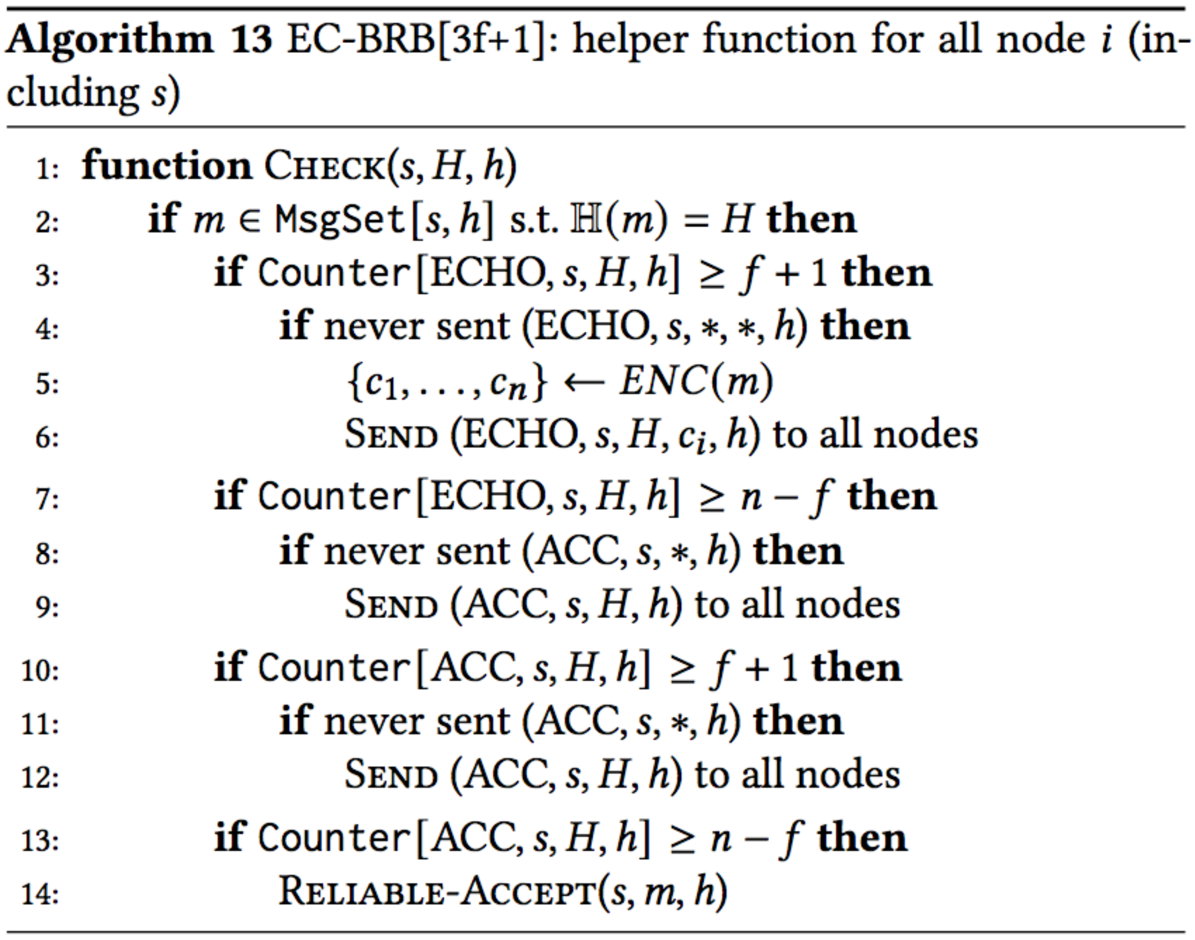
Algorithm 11 EC-BRB[3f+1]: source s with message m of index h

```
1: function RELIABLE-BROADCAST($m, h$)
2:     $\{c_1, c_2, \ldots, c_n\} = \mathrm{ENC}(m)$
3:     SEND(MSG, $s$, $\mathbb{H}(m)$, $c_k$, $h$) to node $k$
```

Algorithm 12 EC-BRB[3f+1]: all node i (including s) when receiving a message from node j

```
function Receiving(MSG, $s, H, c, h$)
    if $j = s$ and first (MSG, $s, *, *, h$) then
        $\texttt{CodeSet}[s, H, h] \leftarrow \texttt{CodeSet}[s, h, H] \cup \{c\}$
        $\texttt{Counter}[\text{ECHO}, s, H, h]++$
        if never sent (ECHO, $s, *, h$) then
            Send(ECHO, $s, H, c, h$) to all nodes
function Receiving(ECHO, $s, H, c, h$)
    if first (ECHO, $s, *, *, h$) from $j$ then
        $\texttt{Counter}[\text{ECHO}, s, H, h]++$
        $\texttt{CodeSet}[s, H, h] \leftarrow \texttt{CodeSet}[s, h, H] \cup \{c\}$
        if $\texttt{Counter}[\text{ECHO}, s, H, h] \geq f + 1$ then
            if $\nexists m \in \texttt{MsgSet}[s, h]$ s.t. $\mathbb{H}(m) = H$ then
                for each $C \subseteq \texttt{CodeSet}[s, H, j]$, $|C| = f + 1$ do
                    $m \leftarrow DEC(C)$
                    if $\mathbb{H}(m) = H$ then
                        $\texttt{MsgSet[s, h]} \leftarrow \texttt{MsgSet[s, h]} \cup \{m\}$
        Check($s, H, h$)
function Receiving(ACC, $s, H, h$)
    if first (ACC, $s, *, h$) from $j$ then
        $\texttt{Counter}[\text{ACC}, s, H, h]++$
        if $\texttt{Counter}[\text{ACC}, s, H, h] \geq f + 1$ then
            if $\nexists m' \in \texttt{MsgSet}[s, h]$ s.t. $\mathbb{H}(m') = H$ then
                Send (REQ, $s, H, h$) to nodes if have not sent (REQ, $s, H, h$) to them before
        Check($j, H, h$)
function Receiving(REQ, $s, H, h$)
    if first (REQ, $s, h$) from $j$ then
        if $\exists m' \in \texttt{MsgSet}[s, h]$ s.t. $\mathbb{H}(m') = H$ then
            Send (FWD, $s, m', h$) to $j$
function Receiving(FWD, $s, m, h$)
    if have sent (REQ, $s, \mathbb{H}(m), h$) to $j$ then
        if first (FWD, $s, m, h$) from $j$ then
            $\texttt{MsgSet}[s, h] \leftarrow \texttt{MsgSet}[s, h] \cup \{m\}$
            Check($s, \mathbb{H}(m), h$)
```

4.2.2: Proof of correctness



Algorithm 13 EC-BRB[3f+1]: helper function for all node i (including s)

```
1:  function CHECK($s, H, h$)
2:      if $m \in$ MsgSet$[s, h]$ s.t. $\mathbb{H}(m) = H$ then
3:          if Counter$[\text{ECHO}, s, H, h] \geq f + 1$ then
4:              if never sent (ECHO, $s, *, *, h$) then
5:                  $\{c_1, \ldots, c_n\} \leftarrow ENC(m)$
6:                  SEND (ECHO, $s, H, c_i, h$) to all nodes
7:          if Counter$[\text{ECHO}, s, H, h] \geq n - f$ then
8:              if never sent (ACC, $s, *, h$) then
9:                  SEND (ACC, $s, H, h$) to all nodes
10:         if Counter$[\text{ACC}, s, H, h] \geq f + 1$ then
11:             if never sent (ACC, $s, *, h$) then
12:                 SEND (ACC, $s, H, h$) to all nodes
13:         if Counter$[\text{ACC}, s, H, h] \geq n - f$ then
14:             RELIABLE-ACCEPT($s, m, h$)
```

The proof of this algorithm is similar to our proof for our hash based reliable broadcast algorithm.

We begin with three important lemmas, the second lemmas follow directly from the reliable and authenticated channel. But now the first lemma is much more complicated as the original message needs to be decoded back from the codes

Lemma1. If a non-faulty source s performs Reliable-Broadcast(H(m), c_k, h), then MsgSet$_i$ [s, h] $\subseteq \{m\}$ at each non-faulty node i.

proof of Lemma 1:

A non-faulty server j will eventually include a message into its message set in line 16 or in line 32. Since there are at least n - f >= 2f + 1> f + 1, server i can get the message back in line 16 by the echo message from other servers.

Lemma2. If a non-faulty node *s* never performs Reliable-Broadcast (H(*m*), c_k, h), then MsgSet$_i$[*s*, h] = ∅ at each non-faulty node *i*.

Lemma3. If two non-faulty nodes *i* and *j* send(ACC,*s*,H(*m*),h) and (ACC, *s*, H(*m′*), h) messages, respectively, then *m* = *m′*.

Proof Property 1 - 4: Here We are going to prove with contradiction. Supposed server i is the first server to send (ACC, s, H(m), h) and server j is the first server to send (ACC, s H(m'), h) messages. Since both of them are the first servers to send their corresponding ACC message, server i and j send the ACC through line 8 in Algorithm 3.

Now let's first look at server i. According to (line 7 Algorithm 13), server i must have collected at least n - f echo messages and at least n - f - f >= f + 1 are from non-faulty servers, which also implies that they can decode back the original message m, where H(m) = H. This means that server j can at most collect 2f Echo messages that are corresponding to message m'. However, since individual servers can only send one type of message in each round, it is impossible for server j to send Acc in line 9 Algorithm 13. Thus we have a contradiction.

Thus from the above proof, we show that Property 1-4 are satisfied.

Now let's look at property 5, which is the guarantee of eventual termination.

Proof Property 5: Supposed server i has reliably accepted the message (line 14, Algorithm 13). Then it has received at least n - f (ACC, s, H(m),h] messages from other servers. Among these messages at least n - f - f >= f + 1 messages are from non-faulty servers. These f + 1 messages will eventually be received by all the non-faulty nodes. Supposed that a server never gets the original message directly from the source, then it will get the f + 1 codes by sending REQ messages to those f + 1 servers (line 23 Algorithm 12). After eventually receiving these f + 1 Acc messages from the non-faulty servers, the remaining non-faulty servers will also broadcast the same ACC message as server i. Thus eventually all the non-faulty servers will receive at least n - f number of ACC messages, which implies that all the non-faulty servers will reliably accept message m (line 14, Algorithm 13).

4.3 EC-BRB[4f+1] (Written by Prof.Tseng)

In order to solve the scalability problem presented in EC-BRB[3f+1], resilience needs to be sacrificed such that in this algorithm, n >= 4f + 1 and k = n - 3f;

4.3.1 Pseudo code

Algorithm 4 EC-BRB[4f+1]: source s with message m of index h

```
1: function RELIABLE-BROADCAST($m, h$)
2:     RELIABLE-BROADCAST(HashTag|$\mathbb{H}(m), h$)
3:     $\{c_1, c_2, \ldots, c_n\} = \mathrm{ENC}(m)$
4:     SEND(MSG, $s, c_k, h$) to node $k$
```

Algorithm 5 EC-BRB[4f+1]: all node i (including s) when receiving a message from node j

```
1: function RECEIVING(MSG, $s, c, h$)
2:     if $j = s$ and first (MSG, $s, *, h$) then
3:         CodeSet[$s, h$] $\leftarrow$ CodeSet[$s, h$] $\cup \{c\}$
4:         if never sent (ECHO, $s, *, h$) then
5:             SEND(ECHO, $s, c, h$) to all nodes
6: function RECEIVING(ECHO, $s, c, h$)
7:     if first (ECHO, $s, *, h$) from $j$ then
8:         CodeSet[$s, h$] $\leftarrow$ CodeSet[$s, h$] $\cup \{c\}$
9:         if |CodeSet[$s, h$]| $\geq n - f$ then
10:            $m \leftarrow$ DEC(CodeSet[$s, h$])
11:            if $m \neq$ ERROR then
12:                MsgSet[$s, h$] $\leftarrow$ MsgSet[$s, h$] $\cup \{m\}$
13:                wait until $\exists x \in$ HashSet[$s, h$] s.t. $x = \mathbb{H}(m)$
                                                  ▷ successful decoding
14:                if never sent (ACC, $s, *, h$) before then
15:                    SEND(ACC, $s, \mathbb{H}(m), h$) to all nodes
16: function RECEIVING(ACC, $s, x, h$)
17:     if first (ACC, $s, *, h$) from $j'$ then
18:         Counter[ACC, $s, x, h$] + +
19:         if Counter[ACC, $s, x, h$] $= f + 1$, and
20:                                  never sent (ACC, $s, *, h$) before then
21:             SEND(ACC, $s, x, h$) to all nodes
22:         Check($s, h$)
23: function RECEIVING(REQ, $s, x, h$)
24:     if first (REQ, $s, x, h$) from $j$ then
25:         if $\exists m \in$ MsgSet[$s, h$] s.t. $\mathbb{H}(m) = x$ then
26:             SEND (FWD, $s, m, \mathbb{H}(m), h$) to $j$
27: function RECEIVING(FWD, $j, m, x, h$)
28:     if have sent (REQ, $j, x, h$) to $j'$ then
29:         if first (FWD, $j, m, x, h$) from $j'$, and $\mathbb{H}(m) = x$ then
30:             MsgSet[$j, h$] $\leftarrow$ MsgSet[$j, h$] $\cup \{m\}$
31:             Check($j, h$)
32: function RELIABLE-ACCEPTING(HashTag|$H', h$) from source $j$
33:     HashSet[$j, h$] $\leftarrow$ HashSet[$j, h$] $\cup \{H'\}$
```

Algorithm 6 EC-BRB[4f+1]: helper function for all node i (including s)

```
1: function CHECK($s, h$)
2:     if $\exists x \in$ HashSet[$s, h$] s.t.
3:                          Counter[ACC, $s, x, h$] $\geq n - f$ then
4:         if $\exists m \in$ MsgSet[$s, h$] s.t. $\mathbb{H}(m) = x$ then
5:             RELIABLE-ACCEPT($s, m, h$)
6:         else
7:             SEND (REQ, $s, \mathbb{H}(m), h$) to these $n - f$ nodes
```

4.4 EC-CRB[f+1]

This algorithm does not handle byzantine failure but is practical in the sense that in most of the systems, nodes do not have byzantine behavior. The common failure case for a server is simply shutting down.

4.4.1 Pseudo Code:

Algorithm 7 EC-CRB: source s with message m of index h

```
1: function RELIABLE-BROADCAST($m, h$)
2:     $\{c_1, c_2, \ldots, c_n\} \leftarrow ENC(m)$                ▹ Encoding message
3:     for each $i$ do
4:         SEND(MSG, $s, c_i, h$) to node $i$
```

Algorithm 8 EC-CRB: all node i (including s) when receiving a message from sender j

```
1: function RECEIVING(MSG, $s, c, h$)
2:     SEND(ECHO, $s, c, h$) to all nodes
3:     CodeSet[$s, h$] $\leftarrow$ CodeSet[$s, h$] $\cup \{c\}$
4: function RECEIVING(ECHO, $s, c, h$)
5:     CodeSet[$s, h$] $\leftarrow$ CodeSet[$s, h$] $\cup \{c\}$
6:     if $|$CodeSet[$s, h$]$| \geq k$ for the first time then
7:         $m \leftarrow DEC($CodeSet[$s, h$]$)$                ▹ Decoding
8:         RELIABLE-ACCEPT($s, m, h$)
9:         SEND (ACC, $s, m, h$) to all peers
```

4.4.2 Proof of Correctness and Complexity:

Correctness: As long as k >= n - f, individual servers can get back the original message from these k elements. In the worst case, f number of servers can crash, and the other servers have to wait for all the coded elements from other servers.

Round complexity is 2; one round for the source to other nodes, and an extra round for individual servers to send their coded elements received from the source to other servers.

Message complexity is O(n^2), as all the servers need to send n messages to other nodes in one round.

Bit complexity is O(n^2L / k). since individual code has size O(L/k).

5 Evaluation:

Various reliable broadcast algorithms have been implemented in Golang. We used Golang as it is a lightweight language for distributed programming. We further evaluate our algorithms using the benchmark tool we built called RMB (Reliable mininet Benchmark). RMB is appropriate for evaluating protocols over a network within a datacenter or a cluster. In this section, we will first introduce the architecture of RMB and then present as well as analysis on the results we collect.

5.1 Architecture of RMB

RMB is built on top of Mininet. The architecture of RMB is presented in Figure 2. The Github repo for RMB is at https://github.com/yingjianwu199868/HRB

The generator layer helps to generate data, collect data and calculate statistics such as throughput and latency. The protocol layer contains the RB protocols that we implement. RMB is extensible in the sense that developers can easily write their own reliable broadcast algorithms without worrying about network communication and benchmark. The only thing they need to do is to conform to the communication protocol between generator and the manager. Finally, the network layer is simulated by Mininet and the network manager that we implement. RMB users can easily use the script we provide to configure the network conditions, e.g., delay, jitter, bandwidth, network topology, etc. Each RMB component runs inside a container, and the entire RMB is simulated on a single machine using mininet. These layers are implemented in Go and python scripts are provided to launch RMB.

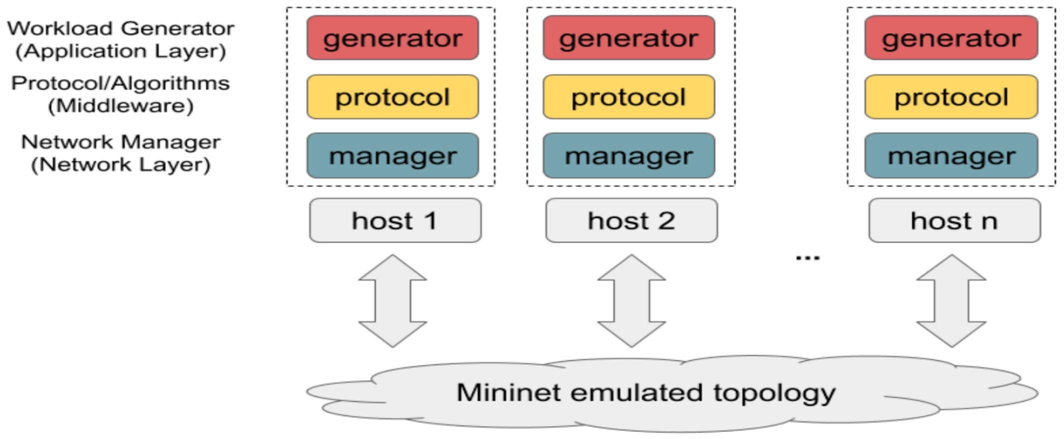

Workload Generator (Application Layer)
Protocol/Algorithms (Middleware)
Network Manager (Network Layer)
generator
protocol
manager
host 1
host 2
...
host n
Mininet emulated topology

Figure 2: The architecture of RMB

Configuration file: Protocol parameters can be specified in a yaml file, including information including number of trusted nodes, number of faulty nodes, whether source byzantine or not, etc. Benchmark managers, protocol and generator are invoked by a python script. Advanced users can also use another yaml file to control the network parameters such as network topology, link bandwidth, and individual node computational power.

Mininet: The bottom layer (gray boxes in figure 2) is a virtualized network, created by Mininet [20, 26]. Mininet is a battle-tested software that is widely used in prototyping Software-De ned Networks (SDNs). The python start-up script calls the Mininet library to start a virtual network consisting of hosts, links, switches and a controller before the start of simulation. A virtual host (container) emulates a node with an OS kernel in a real system. Other three layers of RMB (essentially, Linux applications) run on each host inside Mininet. Hosts do not communicate with each other directly, instead, they connect to switches through Mininet links. Switches are also connected to each through links.

Manager Layer: We have one (network) manager for each host, which is to manage data communication between the protocol layer and other hosts in the network. There are four go routines for separate re- sponsibilities: (i) receiving messages from the protocol layer, (ii) receiving messages from other hosts, (iii) sending to the protocol layer, and (iv) sending messages to other hosts. Another responsibility is to control the faulty behavior if the current node is configured to be Byzantine node, e.g., randomly corrupt messages.

Protocol Layer (RB Algorithms): The middle layer implements the protocol that we want to evaluate. For our purpose, we implement RB protocols here. Each instance is paired up with a manager we discussed above, and thus does not need to know explicitly the existence of other manager/protocol instances. Such a design choice allows researchers to implement new protocols and benchmark them at ease. In our RB protocols, there are two goroutines in this layer. One is responsible for sending messages to the manager layer, and the other one is responsible for reading messages from the manager layer and then performing corresponding action. That is, we implemented an event-driven algorithm as in our pseudo-code. Note that we make minimal assumptions in this layer; hence, potentially, future RMB users can implement their favorite programming language and the algorithms do not have to be event-driven. For the hash function, We used Golang default package hmac512 , and for the erasure coding, We used an open source package written by Klaupost.

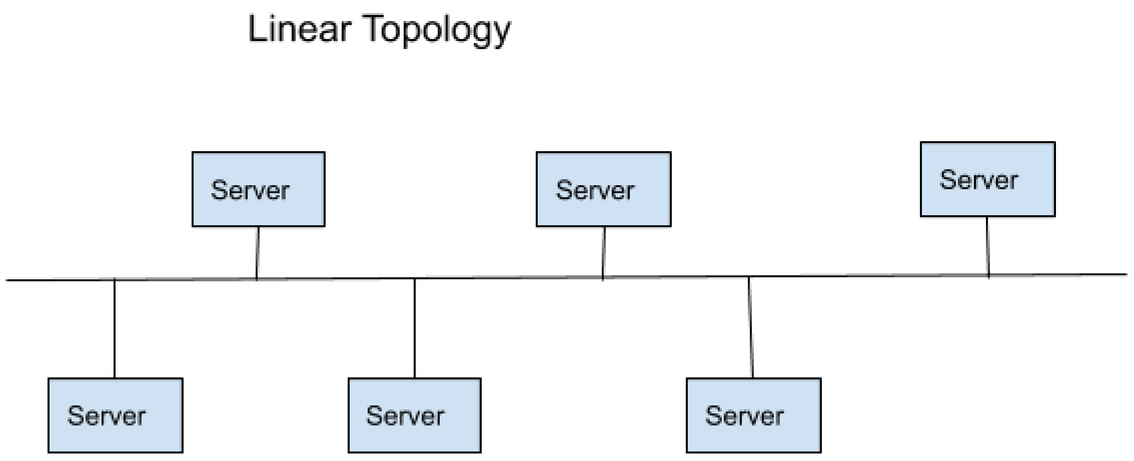
Application Layer (Workload Generator): The top layer implements the workload generator in RMB. There are two roles: (i) issue reliable-broadcast commands following a specified workload (e.g., size, frequency), and (ii) collect and calculate statistics (latency and throughput).

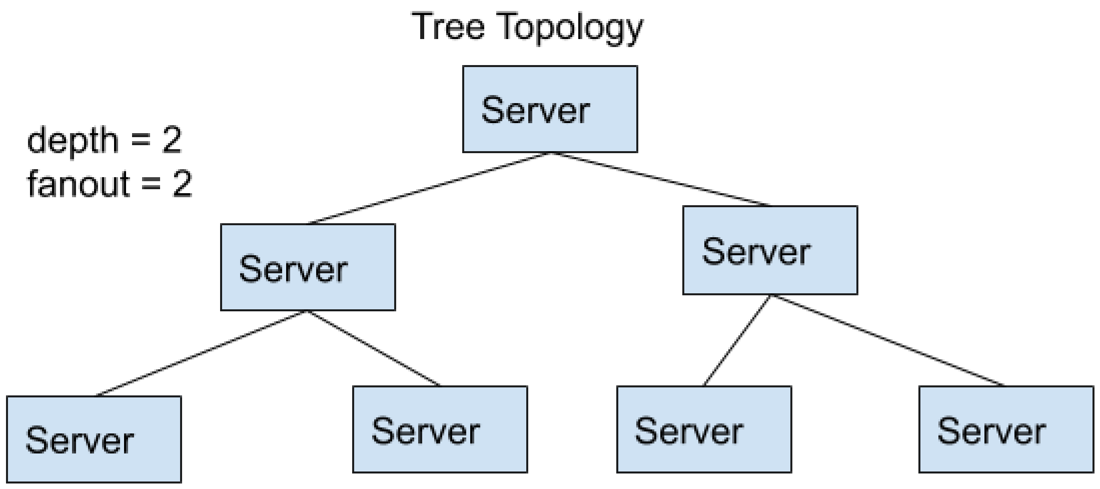
5.2 Performance Evaluation

***Simulation Setup*:** We perform the performance evaluation using RMB on a single virtual machine (Google Cloud Platform instance) with 24vCPU and 48 GB memory. By default, the round trip time between individual nodes is between 0.06 ms and 0.08 ms.

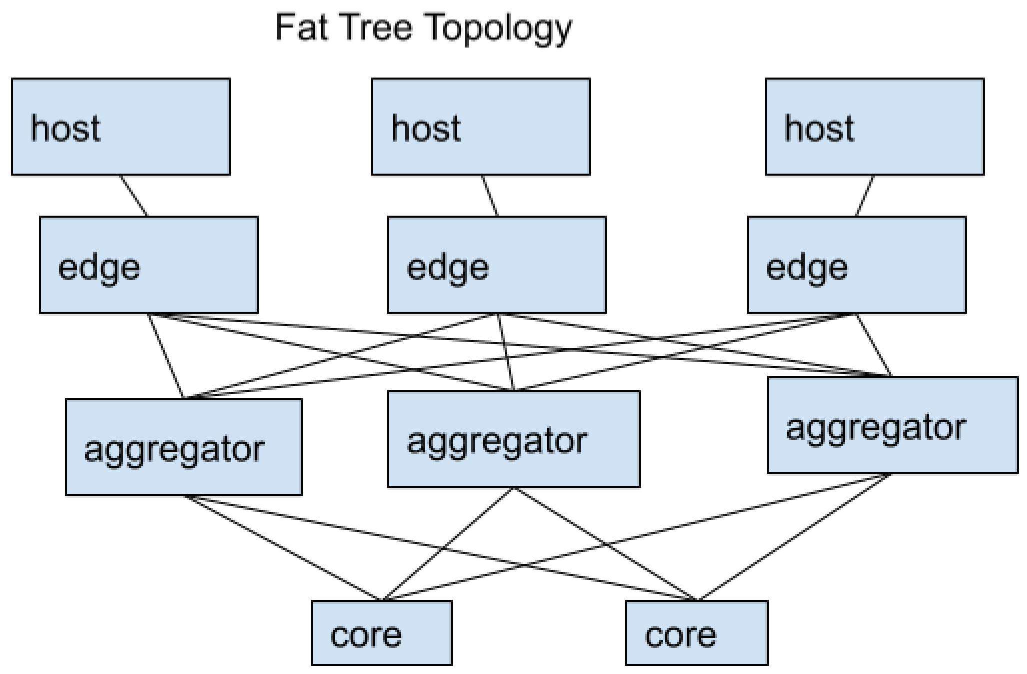
We have done various benchmarks under different scenarios, and have picked 3 of them that give some practical insights.

Evaluation 1 (Different network Topologies): RMB allows developers to easily evaluate different protocols with different network parameters. We first test our protocols with 5 servers and 0 faulty servers under three different network topologies. (1) Linear topology: 5 switches with one host per switch. (2) Tree Topology: tree depth: 3 and fan-out = 2. (3) Fat tree topology: 5 edges, with each host per edge.





depth here means the height of the tree and fanout means number of children for each server.



Fat tree topology is much more complicated that the previous two topologies because oversubscription of links can prevent failure in the network.

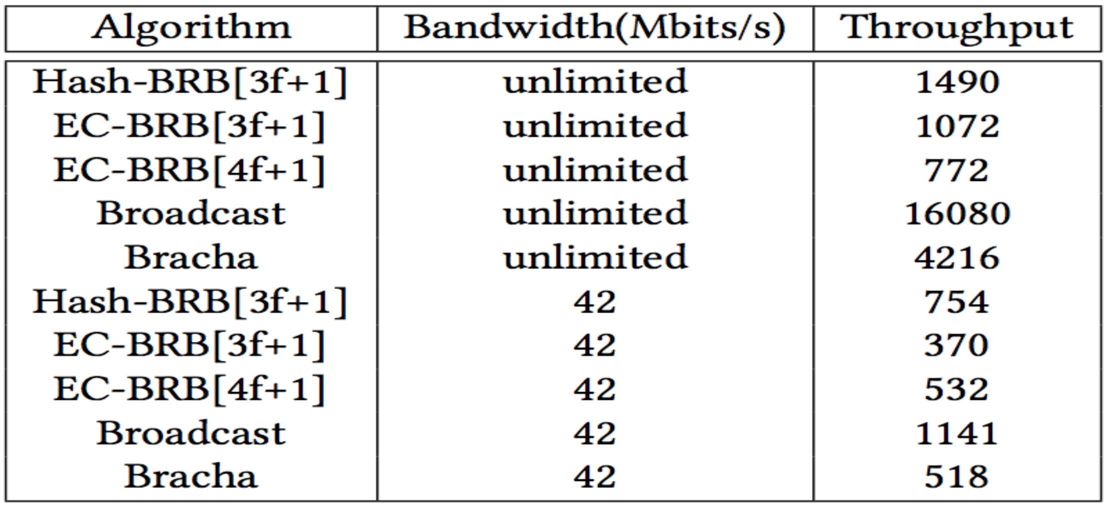
For each data point below, the source broadcast 2000 messages with each message size 1024 bytes. The throughput is calculated as the number of reliable-accept / seconds. We have also implemented a simple non-fault tolerant broadcast as our baseline. The result is presented in table 2, 3, 4.

Algorithm	Bandwidth(Mbits/s)	Throughput
Hash-BRB[3f+1]	unlimited	1041
EC-BRB[3f+1]	unlimited	1047
EC-BRB[4f+1]	unlimited	967
Broadcast	unlimited	16303
Bracha	unlimited	4604
Hash-BRB[3f+1]	42	746
EC-BRB[3f+1]	42	301
EC-BRB[4f+1]	42	426
Broadcast	42	1131
Bracha	42	338

Table 2: Linear Topology

Algorithm	Bandwidth(Mbits/s)	Throughput
Hash-BRB[3f+1]	unlimited	1117
EC-BRB[3f+1]	unlimited	1152
EC-BRB[4f+1]	unlimited	965
Broadcast	unlimited	14604
Bracha	unlimited	4521
Hash-BRB[3f+1]	42	699
EC-BRB[3f+1]	42	301
EC-BRB[4f+1]	42	414
Broadcast	42	1139
Bracha	42	334

Table 3: Tree Topology



Algorithm	Bandwidth(Mbits/s)	Throughput
Hash-BRB[3f+1]	unlimited	1490
EC-BRB[3f+1]	unlimited	1072
EC-BRB[4f+1]	unlimited	772
Broadcast	unlimited	16080
Bracha	unlimited	4216
Hash-BRB[3f+1]	42	754
EC-BRB[3f+1]	42	370
EC-BRB[4f+1]	42	532
Broadcast	42	1141
Bracha	42	518

Table 4: Fat Tree Topology

Under these three different network topologies, when bandwidth consumption is not limited, Bracha outperforms our algorithms as our algorithm incurs computational power. However, as expected, when bandwidth consumption is limited, our algorithms except EC-BRB[3f+1] outperforms Bracha. Moreover, Hash-BRB[3f+1] is 50% of Broadcast's performance.

Evaluation2 (Synchronous vs Asynchronous): Even though a synchronous algorithm does not work in an asynchronous network, it serves as a good baseline (Some might adapt synchronous algorithms to work in a practical setting. For example, in [27], it is argued that the proposed synchronous algorithms are appropriate in a datacenter setting.) In this set up we adopt the single-switch topology, with n = 4, message size 1024 bytes, and 2000 messages broadcast. We compared our algorithms with two synchronous algorithms called Digest, which used hash and NCBA, which used coding. The result is shown in Table 5.

Algorithm	Throughput
Hash-BRB[3f+1]	2038
EC-BRB[3f+1]	1411
EC-BRB[4f+1]	1295
Digest	1674
NCBA	1601

Table 5: Synchronous vs Asynchronous

Interestingly, the throughput performance is close, and Hash-BRB even beat Digest and NCBA by around 20%, even though we get rid of the most expensive stage in Digest and NCBA, which is the dispute control phase, which helps to detect and then blacklist faulty nodes.

Evaluation3 (Hash vs EC): In the final set of experiments, we provide guidance for choosing the best algorithms under an application scenair. We used a single switch topology, n = 20, and 100 rounds of reliable broadcast. Each experiment is with the following setup:

exp1: f = 4, source's bandwidth limitation = 0.4 Mbits/s, message size = 1096 bytes.

exp2: f = 4, source's bandwidth limitation = 4 Mbits/s, message size = 1096 bytes.

exp3: f = 1, source's bandwidth limitation = 0.4 Mbits/s, message size = 1020 bytes

exp4: exp3: f = 1, source's bandwidth limitation = 4 Mbits/s, message size = 1020 bytes.

The result is presented in Table 6.

	Hash-BRB[3f+1]	EC-BRB[3f+1]	EC-BRB[4f+1]
exp1	1.6	2.5	1.2
exp2	19	7	1.2
exp3	1.6	1.3	2.3
exp4	19.3	12.6	16

Table 6: Hash vs. EC

The result in table 6 conforms to our theoretical analysis.

(i) When bandwidth limitation is high, Hash-RRB performs worse than the other two.

(ii) EC-BRB[3f+1] performs better with larger f.

(iii) EC-BRB[4f+1] performs better with smaller f.

6.Conclusion:

In this honor thesis, seeing the importance of bridging theoretical algorithms into practical network systems, we have designed and presented a family of reliable broadcast algorithms using techniques such as cryptographic hash function and MDS codes. We have also implemented other reliable broadcast algorithms from the previous paper as a baseline for our designed protocol. Furthermore, we have built a benchmark tool that is designed for benchmarking the performance of reliable broadcast protocols. This benchmark tool called RMB is extensible and future developers only need to write their own protocol codes without worrying about network configuration and benchmark statistics calculation. Finally, we have used RMB to benchmark the algorithms we have implemented in different scenarios. Based on the benchmark results, we have also included some practical insights on when to choose different algorithms based on different situations.

References

[1] Ittai Abraham, Yonatan Amit, and Danny Dolev. 2005. Optimal Resilience Asyn- chronous Approximate Agreement. In Principles of Distributed Systems, Teruo Higashino (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–239.

[2] Hagit Attiya and Jennifer L. Welch. 2004. Distributed computing - fundamentals, simulations, and advanced topics (2. ed.). Wiley.

[3] GabrielBracha.1987.AsynchronousByzantineAgreementProtocols.Inf.Comput. 75, 2 (Nov. 1987), 130–143. https://doi.org/10.1016/0890-5401(87)90054-X

[4] Damien Imbs and Michel Raynal. 2015. Simple and E cient Reliable Broad- cast in the Presence of Byzantine Processes. CoRR abs/1510.06882 (2015). arXiv:1510.06882 http://arxiv.org/abs/1510.06882

[5]Damien Imbs and Michel Raynal. 2016. Trading o t-resilience for e ciency in asynchronous Byzantine reliable broadcast. Parallel Processing Letters 26, 04 (2016), 1650017.
[6] Arpita Patra and C. Pandu Rangan. 2011. Communication Optimal Multi-valued Asynchronous Byzantine Agreement with Optimal Resilience. In Information Theoretic Security - 5th International Conference, ICITS 2011, Amsterdam, The Netherlands,May21-24,2011.Proceedings.206–226. https://doi.org/10.1007/978-3- 642- 20728-0_19
[7] Michel Raynal. 2018. Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach. Springer. mhttps://doi.org/10.1007/978-3-319-94141-7
[8] Mininet. http://mininet.org/
[9] Nancy A.Lynch 1996. Distributed Algorithms. Morgan Kaufmann
[10] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. Cryptography Mailing list at https://metzdowd.com (03 2009).