Experimental Performance Comparison of Byzantine Fault-Tolerant Protocols for Data Centers

Guanfeng Liang, Benjamin Sommer and Nitin Vaidya
Department of Electrical and Computer Engineering, and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Email: {gliang2,sommer3,nhv}@illinois.edu

***Abstract*—In this paper, we implement and evaluate three different Byzantine Fault-Tolerant (BFT) state machine replication protocols for data centers: (1) BASIC: The classic solution from Pease, Shostak, and Lamport [1]; (2) Digest: A simplified version of the seminal practical BFT protocol PBFT by Castro and Liskov [2]; and (3) NCBA: a network coding based BFT protocol that we propose in this paper. Unlike existing practical BFT protocols such as PBFT, which utilize collision-resistant hash functions to reduce traffic load for BFT, NCBA uses a computationally efficient error-detection network coding scheme. Since NCBA does not rely on any hash function, it is always correct rather than correct only with high probability as PBFT. Moreover, even though NCBA introduces 50% more communication cost than PBFT does, it is compensated by reducing the computational cost of hashing. Through extensive experiments, we verified that NCBA performs at least as well as Digest, without relying on any cryptographic assumption on the hardness of breaking the hash function. To the best of our knowledge, this is the first implementation of BFT with network coding.**

I. Introduction

In the last couple years, we observe a tremendous growth in the popularity of data oriented online services, such as cloud computing, data centers and online storage. More and more enterprises run their critical business applications on data centers. Individual users also have become increasingly dependent on the Internet to store their personal data such as photos, musics, videos, etc. As the reliance of industry, government, and individuals on data centers and other similar online information services increases, the threat posed by malicious attacks and software errors has also become increasingly prominent. For example, software errors have brought down the Amazon S3 storage system for several hours [3], and have caused well-known email services such as Gmail [4] to wipe out customer's emails. Consequently, being able to provide reliable and consistent access to the data and services that they host has become the most basic and most important Quality-of-Service requirement that these online services must fulfill.

Byzantine Fault-Tolerant (BFT) provides a powerful state machine replication approach for providing highly reliable and consistent services in spite of the presence of failures. In BFT state machine replication, $n \geq 3f + 1$ replicas collectively behave as one *fault-free* server, even if up to f replicas are faulty and deviate from the protocol, i.e., *misbehave*, in arbitrary (Byzantine) fashions [1]. The key of BFT is to make sure that all replicas agree upon the same sets as well as the order of requests, and execute them in the agreed upon order. This guarantees that all fault-free replicas always have consistent states and produce the same output. Then the correct output can be obtained by taking the majority of the individual outputs, since at least $2n/3$ of the replicas are fault-free.

Unfortunately, BFT has been barely adopted in practice for the three decades following its introduction in 1980 [1], mainly because of the perception that *error-free* BFT requires prohibitive communication overhead ($\Omega(n^2)$ bits of communication in order to agree on 1 bit) among the replicas.

In the last decade, there are numerous efforts devoted to making BFT systems *practical* [2], [5], [6], [7], [8]. The overhead of BFT has been significantly reduced. A common theme of these BFT protocols/systems is that, in order to check the consistency of a request (or a piece of data) received by different replicas, the replicas exchange hash values (sometimes called *"digests"*) computed from the request using some collision-resistant hash function and check the hash values against the original request, instead of exchanging the entire request. Since the digest is much smaller than the original request in size, communication cost is significantly reduced (roughly by an order of n). Despite the impressive performance improvement achieved by these systems, the use of a collision-resistant hash function may, in fact, be problematic, for the following two reasons:

(1) First of all, the correctness of the aforementioned protocols relies on the collision-resistant property of the hash function used. With the rapid improvement in modern cryptanalysis and computational power of computers, defeating the hash functions may become computationally feasible in the future, and a malicious adversary will be able to find collisions of the hash function and then break the system. For example, Castro and Liskov's seminal Practical Byzantine Fault Tolerance (PBFT) [2] protocol and the follow-up Zyzzyva [5] by Kotla et al. both use MD5, which can now be broken in a few seconds by a regular laptop [9].

(2) The second reason is related to the first one. In a later implementation of PBFT in 2002 [6], MD5 was replaced by SHA-1 in order to improve the reliability of the protocol. However, SHA-1 was then broken in 2008 [10]. It is likely that these protocols need to use more and more secure hash

functions (e.g. SHA-256 and SHA-512), trying to stay ahead of the development in technology and cryptanalysis. However, the more secure a hash function is, the more expensive it is and the longer it takes to compute. So the improvement we gain from reducing communication overhead with hashing can become overwhelmed by the increasing computational/time cost we pay for using a stronger hash function.

The discussion above motivates our work in this paper:

Is it possible to design a practical BFT system with the following two properties: (1) Reliable: Always correct (in other words error-free) and does not rely on hash functions; and (2) Efficient: Performance is comparable to the aforementioned systems that use hash functions.

We give a *positive* answer to this question in this paper:

- We introduce the NCBA protocol, a network coding based BFT protocol that does not use any hash function, and can be proved to be error-free, i.e., reliable.
- Experimental results on our testbed show that the NCBA protocol is at least as efficient as the protocols that use hash functions.

This paper is structured as follows. We begin by giving formal problem formulation and describing our system and failure models. Related work is discussed in Section III. Then we briefly discussed Digest, a PBFT-like protocol, in Section IV. Our NCBA protocol is introduced in Section V. In Sections VI and VII we discuss the details of our implementation the protocols, as well as experimentation for performance evaluation. At last, we summarize this paper in Section VIII

II. Problem Formulation and System Models

A. Byzantine Agreement (Broadcast)

Consider a state machine replication system with n servers/replicas. One server is designated as the *source*, denoted as S. The other servers are designated as the *peers*, denoted as $P_1, \cdots, P_{n-1}$. The goal is for all the fault-free servers to "agree on" the messages (or requests) being sent by the source, despite the possibility that some of the nodes may be faulty. In particular, the following conditions must be satisfied:

- **Agreement:** All fault-free servers must agree on an identical message. (Once a server agrees on a certain message, it cannot change its decision.)
- **Validity:** If the source is fault-free, then the agreed message must be identical to the source's message.
- **Termination:** Agreement between fault-free servers is eventually achieved.

B. Models

We assume a *synchronous* distributed system in which nodes are connected by a network. By synchronous, we mean that there are a priori known upper bounds on these quantities:

- Each server has a bounded time between its execution steps;
- Each message transmitted is received in a bounded time;
- Node's local clocks may drift either from each other or from global physical time only by a bounded rate.

Reasonable upper bounds on these quantities are usually easy to obtain in state-of-the-art systems. Nevertheless, the idea of the proposed protocols can be applied to asynchronous settings, and similar comparative performance results are expected in asynchronous versions of these protocols.

To capture the behavior of faulty servers, we assume a Byzantine adversary model. That is, the adversary has complete knowledge on the BFT protocol being used, as well as all information flowing in the system. The adversary can take over up to any $f < n/3$ servers, including the source, over the whole lifetime of the system. These replicas are said to be *faulty* or *compromised*. The faulty servers can engage in any kind of deviations from the protocol, including collusion, crash failures, as well as sending corrupted messages.

C. Batching of Requests

Batching is a commonly used optimization technique in most existing BFT systems such as PBFT and Zyzzyva. We also assume that there are a large number of requests that the source is trying to broadcast. The requests are grouped into batches, each of size L bits[1]. The source assigns sequence numbers to the batches and broadcast them in increasing order of the sequence number.

III. Related Work

Prior work on Byzantine Fault-Tolerance: Byzantine Agreement [1] and Byzantine Fault-Tolerant state machine replication have been studied in both theoretical and practical setting [11], [12], [13], [2], [6], [5]. In theory, it has been proved that at least $n = 3f + 1$ replicas are necessary to tolerate any f Byzantine failures [11] and at least $\Omega(n^2)$ bits are necessary to be communicated, in order to achieve agreement on 1 bit [14]. Algorithms that achieve both lower bounds simultaneously have been designed [12], [13].

However, in practice, the $\Omega(n^2)$ communication cost to agree on just one single bit in a BFT system is considered prohibitively expensive to be implemented. Efforts have been devoted to make BFT practical. Castro and Liskov's Practical Byzantine Fault-Tolerant (PBFT) state-machine replication protocol [2] showed for the first time that BFT can be made practical. PBFT adopts the client-server model. One designated server called the *"primary"* (or source in our terminology). The clients send their requests to the primary. Then the primary authenticates and orders the requests. The ordered requests are then broadcast to the other replicas (peers in our terminology) from the primary. In order to make sure that no two fault-free replicas accept different requests, hash values computed from the requests are exchanged among the replicas. Due to the use of hashing, the communication cost is significantly reduced (to roughly $O(n)$). Follow ups of PBFT, such as Zyzzyva [5] and Aardvark [15], all take the similar

[1] Sizes of batches may be different in practice. The uniform size assumption here is just to simplify the discussion.

hashing approach. However, due to the use of hashing, these protocols are not error-free. The probability of error depends on the probability of collision of the hash function they use, and also depends on the adversary's ability to break it.

Beerliova-Trubiniova and Hirt have presented an error-free Byzantine broadcast protocol without hashing in [16]. Their protocol also uses the idea of coding to reduce communication complexity, as NCBA does. The NCBA protocol evaluated in this paper improves on the protocol in [16]: while both protocols have a similar structure, the communication cost of [16] is approximately 2 to 2.7 times as high as the NCBA protocol we consider, depending on the actual values of n and f. The main difference between the two protocols is in the manner in which a code is used for error detection.

Prior work on error-free communication using network coding: While the early work on fault tolerance typically relied on replication [17] or source coding [18] as mechanisms for tolerating packet tampering, *network coding* has been recently used with significant success as a mechanism for tolerating attacks or failures. In traditional routing algorithms, a node serving as a router simply forwards packets on their way to a destination. With network coding, a node may "mix" (or *code*) packets from different neighbors [19], and forward the coded packets. This approach has been demonstrated to improve throughput, being of particular benefit in *multicast* scenarios [19], [20], [21]. The problem of *multicast* is related to *agreement*. There has been much research on multicast with network coding in presence of a Byzantine attacker (e.g., [22], [23], [24], [25]).

The significant difference between Byzantine broadcast and multicasting is that the multicast problem formulation assumes that the source of the data is always fault-free. In addition, most of the existing work on fault-tolerant network coding assume a link-failure model, while Byzantine agreement considers the nodes to be faulty. In fact, the unicast/multicast problem with node-failure is an open problem in general, and only a few small networks have been solved [26], [27].

IV. Digest: A PBFT-Like Protocol

In this section we briefly describe a simple PBFT-like protocol: Digest. Digest provides the readers a general idea how protocols such as those in [2], [6], [15], [28] utilize collision-resistant hash functions to achieve Byzantine broadcast. It will also serve as a baseline for the performance evaluation of the NCBA protocol.

A. Normal-Case Operation

Here we describe the operation of Digest for one batch, when no failure occurs. We discuss the operation when failures occur in Section IV-B. Figure 1 illustrates the failure-free operations:

The source S first gather requests from the clients, and assign order to the requests. Then S tries to broadcast a batch of requests m, starting by multicasting a *pre-prepare* message $\langle \texttt{PRE-PREPARE}, m\rangle$ to all of the peers.

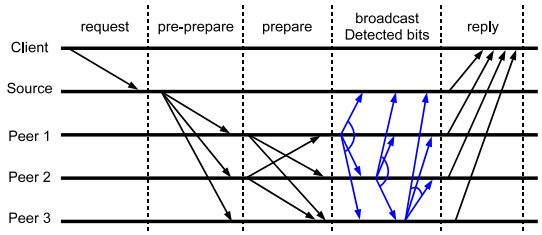


Fig. 1. Normal Case Operation of Digest. Blue arrows represent broadcasting using `Broadcast_Binary`.

After a peer P_i receives pre-prepare message $\langle \texttt{PRE-PREPARE}, m_i\rangle$ from the source S, it sends one *prepare* message $\langle \texttt{PREPARE}, k_{i,j}, d_{i,j}\rangle$ to each peer P_j, where $k_{i,j}$ is a randomly generated key and $d_{i,j} = H(m_i, k_{i,j})$ is m's digest with random key $k_{i,j}$ computed from the pre-determined collision-resistant hash function $H(*,*)$.

A peer P_i waits until it receives all $n-2$ prepare messages from the other peers. Then it checks if $d_{j,i} = H(m_i, k_{j,i})$ for all P_j. If yes, then P_i sets $Detected_i$ to `FALSE`. Otherwise, it sets $Detected_i$ to `TRUE`. Then P_i broadcasts a one-bit message $\langle Detected_i\rangle$ to the rest of the network (including the source S), using a error-free Byzantine broadcast algorithm such as [13], [12], referred as `Broadcast_Binary`. Since `Broadcast_Binary` is error-free, all fault-free servers receive identical $Detected_i$ from each peer P_i.

When all $n-1$ instances of `Broadcast_Binary` terminate, every server (including the source S) checks if all $Detected_j$'s are `FALSE`. If yes, it agrees on its local copy of m (m_i if the server is a peer P_i), executes the requests according to the order in m, sends the output to the client, and moves on to the next batch. Otherwise, extra operations are performed, as discussed next.

B. Failure-Case Operation

It should not be hard to see that whenever at least one of the $Detected_i$ is `TRUE`, the faulty server(s) must have misbehaved in one or some combination of the following ways: (1) a faulty source sends different m to different peers; (2) a faulty peer P_i sends incorrect prepare message to one or more peers; or (3) a faulty peer P_i broadcasts $Detected_i$ =`TRUE` even though it should be `FALSE`. Different protocols handle failures differently. Here we present one technique known as "Fault Diagnosis" [29], [30] or "Dispute Control" [31].

In fault diagnosis, when failure is detected, every server broadcasts all messages it has sent and received, using `Broadcast_Binary`. The peers first agree on the batch of requests broadcast by the source using `Broadcast_Binary`. Then, by comparing the information broadcast by each pair of servers, the fault-free servers will be able to jointly identify at least one pair of servers whose "claims" conflict, and at least one of the two conflicting servers must be faulty. Once a fault-free server X finds itself conflicts with another server Y, X knows that Y is faulty, and will consequently ignore any message received from Y in the future. In the subsequent batches, multicast of the pre-prepare messages from the source S is done only using links between

pairs of servers that have never conflicted with each other.

It has been shown that at least one new pair of conflicting servers will be identified every time this diagnosis process is performed, and a server must be faulty if it conflicts with at least $f+1$ other servers. It follows that the diagnosis process will be performed at most $f(f+1)$ times throughout the lifetime of the system. By then, all faulty servers will be correctly identified. Once a server is found faulty, all fault-free servers will ignore any message received from it and the identified faulty server is then effectively removed from the system. As a result, when the number of batches is much larger than $f(f+1)$, which is typical in practical settings, the overhead for diagnosis is negligible. In practice, if the source is found faulty, a new source will be elected among the servers that are not found faulty.

Dispute control shares similar idea of fault diagnosis, and has the same ability to identify at least one pair of conflicting servers every time it is performed as fault diagnosis does. In dispute control, instead of having all servers to broadcast their sent and received messages, they are required to send this information to one designated server, for example the source S. After collecting the information from the other servers, S tries to find a pair of servers that conflict. Then S broadcast the ID of this pair and the content they conflict in with `Broadcast_Binary`. Then each of this pair of servers broadcasts with `Broadcast_Binary` one bit indicating whether it agrees with S's finding or not. Then a "real" conflict is guaranteed to be found among S and the pair of servers it identified. Since dispute control requires fewer execution of `Broadcast_Binary`, its communication cost is about one order of n lower than fault diagnosis.

C. Correctness and Traffic Load Analysis

The correctness of Digest follows from the correctness of PBFT [2]. We focus on the traffic load that Digest imposes during the normal-case, since this is the scenario that determines the performance of the system [2]. Let κ be the size of the key-digest pair $(k_{i,j}, d_{i,j})$ and B be the communication cost of each execution of `Broadcast_Binary`. The per-batch traffic imposed by Digest is

$$(n-1)L + (n-1)(n-2)\kappa + (n-1)B \text{ bits.} \qquad (1)$$

There exist error-free Byzantine broadcast algorithms [12], [13] whose communication cost is $\Theta(n^2)$, so we assume $B = \Theta(n^2)$, which is much smaller than L in practice. Also, κ is chosen much smaller than L (usually $O(\log L)$), otherwise there is no benefit for hashing. So the per-batch traffic load of Digest is dominated by the $(n-1)L$ term.

V. NCBA: A Network Coding Byzantine Broadcast Protocol

In this section, we present NCBA, a network coding based error-free Byzantine broadcast protocol that does not rely on the use of any hash function and has comparable efficiency as protocols that use hashing as in Digest. NCBA in fact has a similar structure as Digest, except that, instead of counting on the hash function to detect failure/misbehavior as Digest does, NCBA uses a carefully designed linear network code for failure detection.

A. Network Code for Failure Detection

The network code we use is based on Reed-Solomon codes (potentially, other codes may be used instead). In particular, we represent a batch m of L bits as a vector of $n-f$ symbols from Galois Field $GF(2^c)$, where $c = L/(n-f)$. A $(2(n-1), n-f)$ Reed-Solomon code C is used to encode $n-f$ data symbols from $GF(2^c)$ into a codeword consisting of n symbols from $GF(2^c)$. Each symbol from $GF(2^c)$ can be represented using c bits. Thus, a data vector of $n-f$ symbols contains $(n-f)c$ bits, and the corresponding codeword contains $2(n-1)c$ bits.

Each symbol of the codeword is computed as a linear combination of the $n-f$ data symbols, such that every subset of $n-f$ coded symbols represents a set of linearly independent combinations of the $n-f$ data symbols. This property implies that any subset of $n-f$ symbols from the $2(n-1)$ symbols of a given codeword can be used to determine the corresponding data vector. Similarly, knowledge of any subset of $n-f$ symbols from a codeword suffices to determine the remaining symbols of the codeword. These properties can be achieved for sufficiently large c such that $2(n-1) \leq 2^c - 1$. This implies $L = \Omega(n \log n)$, which is usually true in practice.

In our protocol, we also assume the availability of a null ($\perp$) symbol that is distinguished from all other symbols. The encoding and decoding functions of the $(2(n-1), n-f)$ Reed-Solomon code C are denoted as $C(m)$ and $C^{-1}(Z)$, where Z is a $2(n-1)$-dimension vector of symbols from $\{\perp\} \cup GF(2^c)$ with at least $n-f$ non-null ($\neq\perp$) symbols. The decoding function returns an output $m = C^{-1}(Z)$ if there exists a codeword $Z' = C(m)$ such that all the non-null symbols of Z are equal to the corresponding ones of Z'; otherwise $C^{-1}(Z) =$ `FAILURE`.

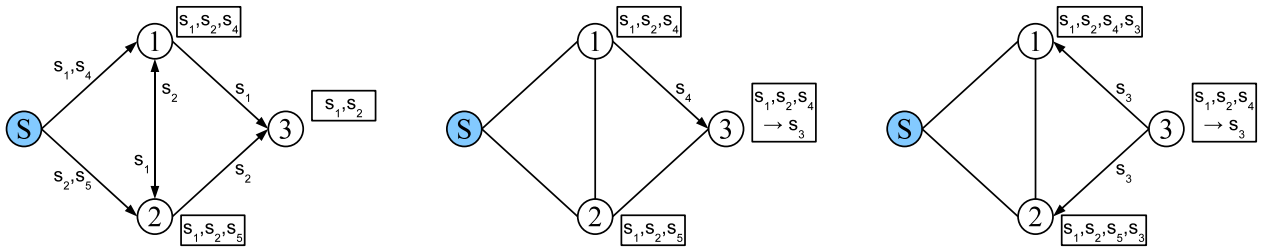
B. The NCBA Protocol

Now we discuss the detail of operations of NCBA for each batch of requests. For convenience of discussion, we will say two servers *"trust"* each other if no conflict has been found between them in previous batches. Since two fault-free servers will never conflict with each other [31], [29], [30], they always trust each other. We also assume that every server trusts every other server at the beginning of the first batch.

The pseudo-code of NCBA is presented in Algorithm 1. NCBA has a similar structure as Digest:

`Lines 1 to 3:` These steps correspond to the source S multicasting the pre-prepare message to the peers in Digest. The difference lies in that, in NCBA, the source S encodes the batch of requests m into n coded symbols with C and only sends a small set of the coded symbols to each of the trusted peers, rather than sending the whole batch m to every peer as it does in Digest. In particular, two symbols $(s_i, s_{i+(n-1)})$ are sent to every peer P_i that S trusts.

`Lines 4 to 7:` These steps correspond to the peers exchanging prepare messages in Digest. Each peer P_i trusted



(a) Source S does not trust P_3. Transmissions by S, P_1 and P_2 are shown. P_3 receives $2 < n-f = 3$ coded symbols by the end of step 4.

(b) Peer P_3 request s_4 from P_4 so that it has $3 = n-f$ coded symbols. Then P_3 uses them to reconstruct s_3 in step 6.

(c) Peer P_3 sends s_3 to all its trusted peers. By the end of step 7, all peers share packets s_1, s_2, s_3.

Fig. 2. Examples of some peer does not trust the source with $n = 4$ and $f = 1$. Two servers connected by an edge trust each other. The direction of an arrow indicates the direction of a transmission along that edge. Next to each arrow, the coded symbol(s) transmitted on that edge is listed. The boxes near to the peers indicate the coded symbols that are available to the peers by the end of steps 4, 6 and 7, respectively.

Algorithm 1 NCBA Protocol

In the following steps, r_i is a $2(n-1)$-dimension vector, which is initialized as all $\perp$. For every peer P_i: $r_i[k] \leftarrow r_j[k]$ whenever P_i receives $r_j[k]$ from its trusted peer P_j.

Source S:

1) Encode the batch m into $(s_1, \cdots, s_{2(n-1)}) = C(m)$.
2) For each trusted peer P_i: Send $(s_i, s_{i+(n-1)})$ to P_i.

Each peer P_i trusted by S:

3) $(r_i[i], r_i[i+(n-1)]) \leftarrow (s_i, s_{i+(n-1)})$ received from S in step 2.
4) Send $r_i[i]$ to all trusted peers.

Each peer P_i not trusted by S:

5) If $k < n-f$ peers are trusted by both P_i and S, P_i receives $< n-f$ symbols in step 4:
   Request $r_j[j+(n-1)]$ from each P_j trusted by both P_i and S, until r_i has $n-f$ non-null symbols.
6) Now r_i has $n-f$ non-null symbols:
   Decode $Z = C^{-1}(r_i)$ and set $r_i[i]$ to be the i-th symbol of Z. If the decoding fails, set $r_i[i]$ to some default.
7) Send $r_i[i]$ to all trusted peers.

Every peer P_i:

8) If $C^{-1}(r_i) =$ `FAILURE` then $Detected_i \leftarrow$ `TRUE`;
   Otherwise $Detected_i \leftarrow$ `FALSE`.
9) Broadcast $Detected_i$ using `Broadcast_Binary`.
10) Receive $Detected_j$ from each processor P_j (broadcast in step 9):
    If $Detected_j =$ `FALSE` for all P_j, agree on $C^{-1}(r_i)$;
    Otherwise, perform fault diagnosis (Similar to Digest.).
11) Proceed to the next batch.

by the source S simply relays s_i to all of its trusted peers after its received from S (Figure 2(a)).

For a peer P_i not trusted by S, it first gathers enough ($\geq n-f$) coded symbols from the peers trusted by both itself and S (Figure 2(b)). Notice that both of S and P_i must trust at least $n-f-1$ other peers, otherwise at least one of them conflicts with at least $f+1$ other servers, and should have already been identified as faulty and removed from the system. Through simple counting, there must be at least $n-2f$ peers that are trusted by both S and P_j. Given that $n \geq 3f+1$, these peers together receive at least $2(n-2f) \geq n-f+1$ coded symbols from S. So it is guaranteed that P_i will receive enough coded symbols in `line 5`, which are then used to generate the i-th symbol of the codeword that P_i should have received from S if they trust each other. Then P_i sends the i-th symbol to all its trusted peers (Figure 2(c)).

`Lines 8 and 9`: Every fault-free peer P_i checks whether the set of symbols received from its trusted server belong to part of a valid codeword of C by trying to apply the decoding function $C^{-1}(r_i)$. If the decoding fails, i.e., $C^{-1}(r_i) =$ `FAILURE`, we say that P_i detects the failure/misbehavior. Then P_i broadcasts $Detected_i$ using `Broadcast_Binary`. This is the counterpart of checking m_i against the received hash value and key pairs in Digest.

`Line 10`: If no peer claims that it detects the failure, then each peer P_i agrees on $C^{-1}(r_i)$ as the current batch of requests. Otherwise, fault diagnosis is performed in the same way as in Digest.

C. Correctness and Traffic Load of NCBA

The correctness of NCBA is guaranteed by Theorem 1. The proof can be found in Appendix A.

Theorem 1: If none of the peers claims failure detected, the peers agree on an identical output m'. In the case the source S is fault-free, $m' = m$.

To compute the normal-case traffic load of NCBA, we count the number of coded symbols communicated from `lines 2 to 7`. It is easy to verify that each peer receives at most n symbols per batch. Since there are $n-1$ peers, it ends up with no more than $n(n-1)$ symbols communicated per batch. Taking the size of each symbol as $c = L/(n-f)$, the cost of NCBA in normal-cases is obtained by simple calculation:

$$\frac{n(n-1)}{n-f}L + (n-1)B \text{ bits.} \tag{2}$$

Here $(n-1)B$ is the cost of broadcasting $Detected_i$ from $(n-1)$ peers. Since B is much smaller than L in practice, the per-batch traffic imposed by NCBA is roughly $\frac{n(n-1)}{n-f}L < 1.5(n-1)L$, since $f < n/3$. Recall that the cost of Digest is approximately $(n-1)L$. So in terms of traffic load, NCBA

is about 50% more expensive than Digest. But the decoding function $C^{-1}()$ is much less expensive in computational complexity for small f, compared to the hash function used in Digest. The efficiency of the protocols are determined by the combination of communication and computational costs. As we will see later, according to experimental results, NCBA is at least as efficient as Digest, without the need for a secure hash function and avoids its disadvantages (as discussed earlier).

VI. Implementation

In this section, we discuss some of our implementation choices and some optimization we adopted for NCBA.

A. MultiThreading

We implement both Digest and NCBA in a multithreading fashion. On every server, there is one listener thread for each of the other servers, which simply accepts incoming connection requests from the server that it listens to. TCP is used for communication between every pair of servers. Also since Digest and NCBA share the same structure, it also makes it easier to share the same code between the two protocols.

A worker thread is launched whenever a listener thread receives a TCP connection request. The worker thread establishes a TCP session with the requesting server. When the TCP session completes, the worker thread checks whether the received information follows the specification of the protocol, and if enough messages have been received from different servers. If yes, it proceeds to the consistency checking part (checking the digests against the local batch in Digest, and decoding $C^{-1}(r_i)$ in NCBA) and the rest of the protocol.

Of course, the protocols can be implemented in the sequential manner as well. However, this will require careful coordination among the servers about when and who should transmit to whom. Otherwise the system will end up deadlocked when two servers try to send messages to each other. This can be very complicated in large systems. With multithreading, this complication is avoided.

Multithreading also makes it very easy to incorporate different optimizations. For example, with "speculative execution" technique in Zyzzyva [5], a server executes the request speculatively when it receives a minimum number of matching prepare messages, even if it cannot confirm that all other servers have accepted the same request. So by the time it confirms an agreement of the request, the output has already been computed and is waiting to be sent to the client. This technique can be incorporated into our implementation by having the worker thread launch a "speculate" thread for performing the request speculatively.

B. Synchrony

Although we assume a synchronous system model in the previous discussion, bad things can happen in real life. Synchrony might be violated temporarily in practice. For example, packets might be lost in the network and never reach the destination, probably due to queue overflow at the router/gateway. Since we use TCP as the underlying communication protocol, packet losses in the middle of a TCP session have been taken care of by the retransmission mechanism of TCP. So the only problem packet losses can cause is when one server attempts to create a connection with another server. Our implementation incorporates this concern by allowing each server to try to connect to another server for no more than `MAX_ATTEMPTS` times. If a server fails to create a connection with another server after `MAX_ATTEMPTS` attempts, it considers that server as being faulty. The value of `MAX_ATTEMPTS` can be chosen according to the capability of the hardwares of the communication infrastructure used and other specification of the system. In our experiments, `MAX_ATTEMPTS` is set to 2.

C. Optimization for single failure

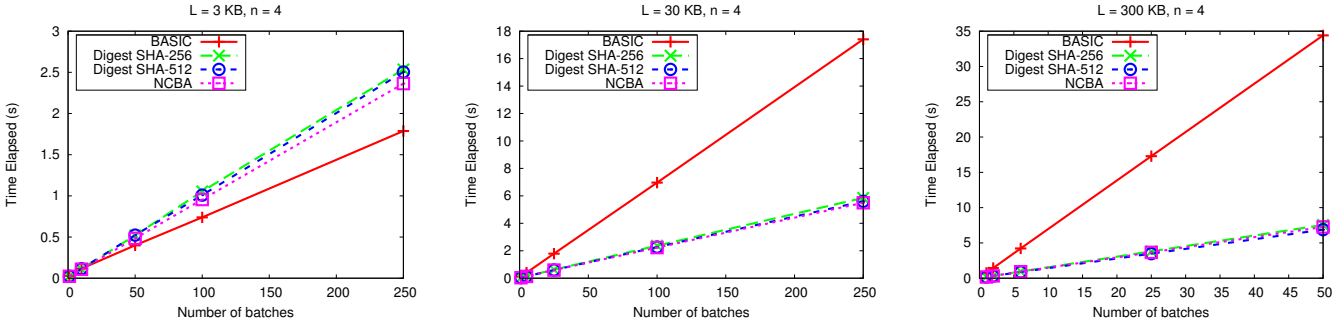
In replication systems where each individual replica is reasonably reliable, the probability that more than one servers fail at the same time is very small. Most of the time, at most one server may fail. In this case, $f = 1$ and NCBA can be optimized to further reduce the computational cost as follows:

Instead of using a $(2(n-1), n-1)$ Reed-Solomon code, we use an $(n, n-1)$ Reed-Solomon code, which is in fact a parity code: $s_1, \cdots, s_{n-1}$ are just raw data symbols, and the parity symbol $s_n = \oplus_{i=1}^{n-1} s_i$, where $\oplus$ denotes bitwise exclusive or (XOR). Then we simply substitute all $s_{i+(n-1)}$ in Algorithm 1 with s_n. For failure detection, a peer just needs to check if the received symbols pass the parity check. If all peers claim with `Broadcast_Binary` that their symbols passed the parity check, then the output is simply $s_1, \cdots, s_{n-1}$. The computation for this modified NCBA protocol is very efficient: each server just needs to perform L bitwise XOR's per batch.

VII. Experimental Evaluation

We dedicate this section to investigations of how Digest and NCBA perform in real systems. We implemented Digest and NCBA in Linux. We conduct our experiments in systems of both four and five servers: four (or five) Dell Inspiron 1545 laptops connected by a Netgear GS108 gigabit switch. Each Inspiron 1545 was running Ubuntu 9.04 Jaunty, Gnome 2.26.1, and Linux Kernel 2.6.28-11-generic on 2.9 GiB of RAM and a 2.0 GHz Intel Core 2 Duo Processor T6400. One Inspiron machine was designated to be the source server and the remaining three (or four) machines made up the group of peers. In order to ensure consistency between test runs, the role performed by each machine remained constant throughout the experiment. In Digest, we use both SHA-256 and SHA-512 as the collision-resistant hash function. We use the realization of SHA-256 and SHA-512 from the OpenSSL toolkit [32]. The results with five servers are very similar to the ones with four servers. So we only present results for four servers in this paper.

For comparison, we also implemented the classic protocol by Pease, Shostak and Lamport [1], denoted here as BASIC. BASIC is also used to realize `Broadcast_Binary` in Digest and NCBA. When there is at most one faulty server ($f = 1$), BASIC is very simple: the source sends the batch



(a) Batch Size: 3 KB (b) Batch Size: 30 KB (c) Batch Size: 300 KB

Fig. 3. Time it takes to finish Byzantine broadcast when each batch has fixed size.

m to every peer and the peer forwards that to every other peer. The output at each peer is computed as the majority of the received messages. If there is no clear majority, then all peers know that source must be faulty and will agree on some default message. It is easy to see that the communication cost of BASIC is $(n-1)^2L = 9L$ for $n = 4$ and $f = 1$, which is one order of n higher than Digest and NCBA, which are $(n-1)L = 3L$ and $n(n-1)L/(n-1) = 4L$, respectively.

To minimize any external error introduced by thread scheduling and background services, each data point in the following discussion/figures represents an average of 100 identical trials. To further reduce any skew, the trials between the three different protocols were interleaved so that any unforeseeable incident would affect the algorithms as evenly as possible. In each trial, the actual content of each batch was randomly generated. This random generation occurred at run time and differed for each algorithm. We see no reason for the content of each batch to have any bearing on the results of our experiment. For each set of 100 trials, the batch size L and the number of batches g is fixed. All protocols were timed from the moment the source begins executing at the first batch until the moment that broadcast of all g batches terminates at the source server. Timing only the source is an accurate measure since for every protocol the source can not terminate until it has reliably received all $n-1$ $Decided_i$ bits as `FALSE` from `Broadcast_Binary`. So when the source terminates, it is guaranteed that all peers have agreed upon the correct information.

A. Fixed Batch Size

We first test the normal-case performance of BASIC, Digest and NCBA when the batch size L is fixed and vary the number of batches g. In Figure 3 we plot the time taken to finish a certain number of batches g, with the batch size L chosen from 3 KB, 30 KB and 300 KB.

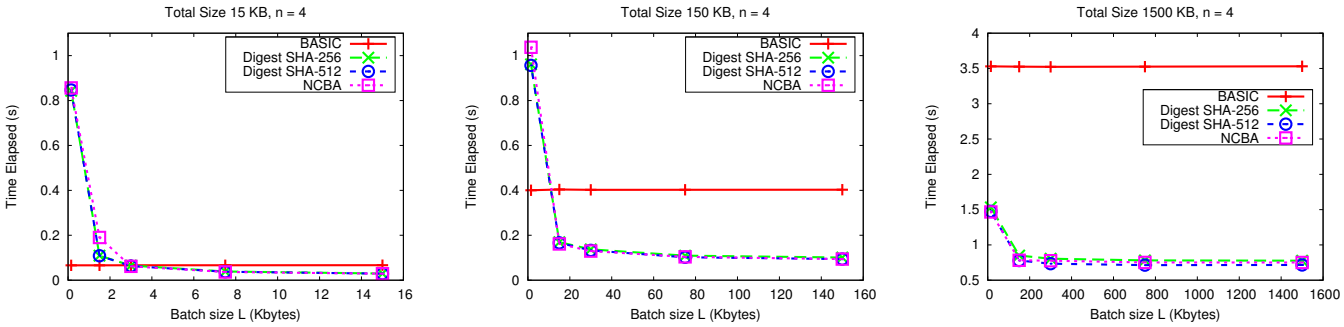
We first observe that in all three settings, NCBA is at least as fast as Digest, both with SHA-256 and SHA-512. This confirms our earlier argument in this paper that although NCBA introduces almost 50% more traffic than Digest does, its overall efficiency is still comparable to Digest. Since NCBA does not need to compute any collision-resistant hash function, its computational cost is much lower than Digest. If we look closer, NCBA is about 7% faster than both Digest protocols when $L = 3$ KB, and it is roughly as fast as Digest when $L = 30$ KB and 300 KB. We believe this is due to the fact that both SHA-256 and SHA-512 reach their full speed (highest number of bits processed per second) only when the input size is large enough. So when L is small (3 KB), the per-bit computational cost of Digest is higher than it is when L is larger (30 KB and 300 KB). The per-bit computational cost of NCBA remains roughly constant when L varies between 3 KB and 300 KB.

We also observe that although, as we have expected, BASIC is much slower than Digest and NCBA when L is large: Digest and NCBA are approximately 3 times and 4.7 times faster than BASIC when $L = 30$ KB and 300 KB, respectively. However BASIC is in fact more efficient than both Digest and NCBA when the batch size is small (3 KB). The reason for this is that when L is small, the communication cost for the $n-1$ executions of `Broadcast_Binary` is not negligible but comparable to the other communication cost in Digest and NCBA. Although `Broadcast_Binary` does not pose too much traffic (each execution only broadcasts 1 bit ($Detected_i$)), it introduces a large number of message transmissions which results in a long cumulated delay and hence, affects the performance.

B. Effects of Different Batch Sizes

To further understand how the batch size L affects the performance of the protocols, we perform the second set of tests for normal-case, in which the total amount of requests, i.e., gL is fixed, and L varies. Figure 4 shows the time taken to finish total workload of 15 KB, 150 KB and 1500 KB, with different batch sizes.

For each setting, the time required using BASIC remains constant when the batch size changes. This is because the per-bit communication and computational costs of BASIC are determined just by the size of the system and are independent of the batch size. On the other hand, performance of Digest and NCBA changes when we use different batch sizes. When L is small, Digest and NCBA need more time to complete broadcasting in comparison to when L is large. This is because when the total workload is fixed, the smaller L is, the more batches there are. As a result, more instances of `Broadcast_Binary` are executed when L is small, since the number of executions of `Broadcast_Binary` per batch



(a) Total Size: 15 KB (b) Total Size: 150 KB (c) Total Size: 1500 KB

Fig. 4. Time it takes to finish batches with fixed total size

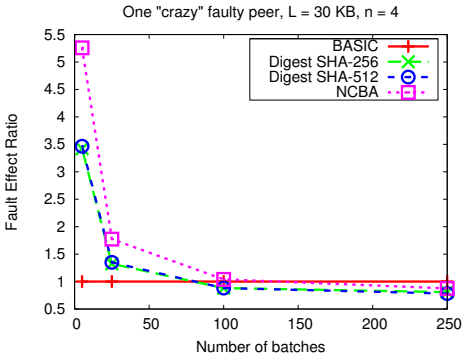


Fig. 5. One "crazy" faulty peer keeps sending arbitrary corrupted messages to all other servers, starting from the first batch.

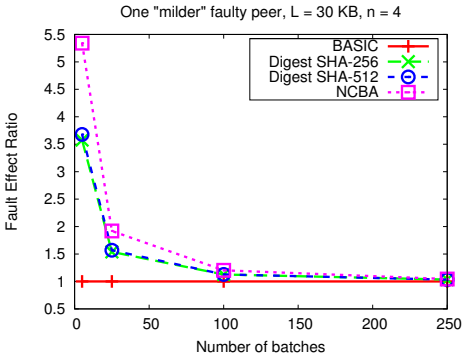


Fig. 6. One "milder" faulty peer keeps sending arbitrary corrupted messages to only one other server, starting from the first batch.

is fixed ($n-1$ per batch). So for total workload of 15 KB and 150 KB, Digest and NCBA are actually slower than BASIC for small L. When L increases, the time it takes to finish the workload in Digest and NCBA quickly converges to a constant that is smaller than the time BASIC takes, in all three settings. This also complements our previous test results for fixed L in Figure 3.

C. Performance with Failure

We also conduct tests for the failure cases. To quantify the effect of the presence of one faulty server on the performance of the protocols, we define *"fault effect ratio"* R as:

$$R = \frac{Time\ with\ failure}{Time\ without\ failure}. \quad (3)$$

It reflects how much slower/faster a protocol becomes when one of the servers goes "bad" (or fails).

We first test the scenario when a faulty peer goes "crazy": the faulty peer sends arbitrary corrupted messages to all other servers, starting from the first batch. It is easy to see that BASIC's performance does not depend on the presence of failures, so its R is always 1. On the other hand, as we can see from Figure 5, when the total number of batches is small, R is as large as 5.3 for NCBA and about 3.5 for Digest. This means when a faulty server starts to misbehave and number of batches is small, NCBA and Digest are slowed down for 5.3 and 3.5 times, respectively. The reduction in efficiency is due to the fault diagnosis process, in which every server is required to broadcast everything it has received or sent using `Broadcast_Binary`. As the number of batches increases, R decreases. When the number of batches reaches 250, R becomes smaller than 1 (roughly 0.75 - 0.8), which implies that when a faulty server misbehaves, the system in fact becomes faster in the long run. It may seem counter intuitive that a faulty server could improve the performance. The explanation for this phenomenon is that since the faulty peer sends corrupted messages to all other servers, conflicts are found between the faulty peer and all other servers during the fault diagnosis process. Then the faulty peer is immediately identified by the others. So after the first batch, all fault-free servers will not communicate with the faulty peer, and do not need to check the consistency of the faulty peer's messages. So both communication and computation costs are reduced after the first batch, and hence the protocols become more efficient. The cost of diagnosis in the first batch became negligible once amortized over a large number of batches. Since the computational cost of the hash function in Digest is much higher than the decoding function of NCBA, Digest saves more than NCBA from identifying the faulty server. As a result, the fault effect ratio of Digest is always slightly smaller than that of NCBA.

We also test a "milder" faulty peer, which only sends corrupted messages to one fault-free peer. In this scenario, only one conflict is found and the faulty server is not exactly identified. In Figure 6, similar trend as in Figure 5 is observed. R converges to 1 when the number of batches is large, which means that the performance is the same as in normal-case,

because the faulty server is not isolated.

Both tests show that with the fault diagnosis process in Digest and NCBA, the misbehavior of faulty servers will only degrade the performance of the system temporarily, and the long term performance will not be degraded (or even will be improved), even if the faulty servers persistently misbehave.

VIII. Conclusion

In this paper, we present NCBA: a network coding based Byzantine broadcast protocol for BFT state machine replication in data centers. Compared with most existing practical BFT protocols that utilize collision-resistant hash functions, NCBA has the following advantages:

1) The correctness of NCBA is guaranteed in all cases, and it does not rely on any cryptographic assumption of unbreakable hash functions. So the system is always reliable even if the adversary is powerful enough to break the hash functions with acceptable cost.
2) Since NCBA only uses a simple linear coding scheme, the computational burden it imposes onto the system is small. On the other hand, the computational cost of the hashing-based protocols can be much higher, and it is expected to increase as stronger hash functions are needed to protect the system against the adversary that is increasingly capable as technology improves.

We show that, through experimentation that NCBA is at least as efficient as as existing hashing-based protocol.

Acknowledgment

This research is supported in part by Army Research Office grant W-911-NF-0710287 and National Science Foundation award 1059540. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

References

[1] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *JOURNAL OF THE ACM*, 1980.
[2] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, Berkeley, CA, USA, 1999, pp. 173–186.
[3] Amazon, "Amazon s3 availability event," July 2008. [Online]. Available: http://status.aws.amazon.com/s3-20080720.html
[4] Gmail, "Gmail disaster: Reports of mass email deletions," December 2006. [Online]. Available: http://techcrunch.com/2006/12/28/gmail-disaster-reports-of-mass-email-deletions
[5] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, pp. 7:1–7:39, January 2010.
[6] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, pp. 398–461, November 2002.
[7] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *SIGOPS Oper. Syst. Rev.*, 2005.
[8] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hibrid quorum protocol for byzantine fault-tolerance," in *OSDI*, 2006.
[9] T. Xie and D. Feng, "How to find weak input differences for md5 collision attacks," Cryptology ePrint archive, May 2009.
[10] S. Manuel, "Classification and generation of distrubance vectors for collision attacks against sha-1," in *Designs, Codes and Cryptography*, 2011.
[11] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. on Programming Languages and Systems*, 1982.
[12] B. A. Coan and J. L. Welch, "Modular construction of a byzantine agreement protocol with optimal message bit complexity," *Inf. Comput.*, vol. 97, no. 1, pp. 61–85, 1992.
[13] P. Berman, J. A. Garay, and K. J. Perry, "Bit optimal distributed consensus," *Computer science: research and applications*, 1992.
[14] D. Dolev and R. Reischuk, "Bounds on information exchange for byzantine agreement," *J. ACM*, vol. 32, no. 1, pp. 191–204, 1985.
[15] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX symposium on Netowrked Systems Design and Implementation*, 2009.
[16] Z. Beerliova-Trubiniova and M. Hirt, "Perfectly-secure mpc with linear communication complexity," in *TCC*, 2008.
[17] E. C. Cooper, "Replicated distributed programs," in *SOSP'85*, 1985.
[18] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. ACM*, 1989.
[19] S.-Y. Li, R. Yeung, and N. Cai, "Linear network coding," *Information Theory, IEEE Transactions on*, vol. 49, no. 2, pp. 371–381, Feb. 2003.
[20] R. Koetter and M. Medard, "An algebraic approach to network coding," in *ISIT'01*, 2001.
[21] C. Fragouli, D. Lun, M. Medard, and P. Pakzad, "On feedback for network coding," in *CISS'07*, 2007.
[22] N. Cai and R. W. Yeung, "Network error correction, part ii: Lower bounds," *Communications in Information and Systems*, 2006.
[23] T. Ho, B. Leong, R. Koetter, M. Medard, M. Effros, and D. Karger, "Byzantine modification detection in multicast networks using randomized network coding," 2004.
[24] S. Jaggi, M. Langberg, S. Katti, T. Ho, D. Katabi, and M. Medard, "Resilient network coding in the presence of byzantine adversaries," in *INFOCOM'07*, 2007.
[25] S. Kim, T. Ho, M. Effros, and S. Avestimehr, "New results on network error correction: capacities and upper bounds," in *ITA'10*, 2010.
[26] O. Kosut and L. Tong, "Nonlinear network coding is necessary to combat general byzantine attacks," in *Allerton*, October 2009.
[27] G. Liang, R. Agarwal, and N. Vaidya, "Secure capacity of wireless broadcast networks," *Technical Report, CSL, UIUC*, September 2009.
[28] M. Fitzi and M. Hirt, "Optimally efficient multi-valued byzantine agreement," in *PODC '06*, 2006.
[29] G. Liang and N. Vaidya, "Capacity of byzantine agreement with finite link capacity," in *INFOCOM 2011*, 2011.
[30] ——, "Error-free multi-valued consensus with byzantine failures," in *ACM PODC*, 2011.
[31] Z. Beerliova-Trubiniova and M. Hirt, "Efficient multi-party computation with dispute control," in *TCC*, 2006.
[32] "Openssl project." [Online]. Available: http://www.openssl.org/

Appendix A
Proof of Theorem 1

Proof: First consider the case when S is faulty. In this case, every fault-free peer P_i sends $r_i[i]$ to all peers that it trusts. Also, as we have discussed in Sections IV-B and V-B, fault-free servers always trust each other. These two facts together imply that among coded symbols received by the fault-free peers, at least $n-f$ of them are shared identically. According to the property of the $(2(n-1), n-f)$ Reed-Solomon code C, either all fault-free peers succeed in decoding the received symbols to some identical output m', or at least one of them cannot decode and detects the failure.

In the case, S is fault-free. So each fault-free peer P_i receives at least $n-f$ coded symbols either directly from S or through other fault-free peers. So these symbols must be the same as the corresponding ones in $C(m)$, and hence $C^{-1}(r_i) = m$ if it succeeds. ■