BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration

Sisi Duan[1], Hein Meling[2], Sean Peisert[1], and Haibin Zhang[1]

[1] University of California, Davis
{sduan,speisert,hbzhang}@ucdavis.edu
[2] University of Stavanger, Norway
hein.meling@uis.no

Abstract. In this paper, we describe the design and implementation of BChain, a Byzantine fault-tolerant state machine replication protocol, which performs comparably to other modern protocols in fault-free cases, but in the face of failures can also quickly recover its steady state performance. Building on chain replication, BChain achieves high throughput and low latency under high client load. At the core of BChain is an efficient Byzantine failure detection mechanism called *re-chaining*, where faulty replicas are placed out of harm's way at the end of the chain, until they can be replaced. Our experimental evaluation confirms our performance expectations for both fault-free and failure scenarios. We also use BChain to implement an NFS service, and show that its performance overhead, with and without failures, is low, both compared to unreplicated NFS and other BFT implementations.

1 Introduction

Building online services that are both highly available and correct is challenging. Byzantine fault tolerance (BFT), a technique based on state machine replication [25, 31], is the only known *general* technique that can mask *arbitrary* failures, including crashes, malicious attacks, and software errors. Thus, the behavior of a service employing BFT is indistinguishable from a service running on a correct server.

There are two broad classes of BFT protocols that have evolved in the past decade: broadcast-based [1, 5, 12, 24] and chain-based protocols [19, 34]. The main difference between these two classes is their performance characteristics. Chain-based protocols aim at achieving high throughput, at the expense of higher latency. However, as the number of concurrent client requests grows, it turns out that chain-based protocols can actually achieve lower latency than broadcast-based protocols. The downside however, is that chain-based protocols are less resilient to failures, and typically relegate to broadcasting when failures are present. This results in a significant performance degradation.

In this paper we propose *BChain*, a fully-fledged BFT protocol addressing the performance issues observed when a BFT service experiences failures. Our evaluation shows that BChain can quickly recover its steady-state performance, while Aliph-Chain [19] and Zyzzyva [24] experience significantly reduced performance, when subjected to a simple crash failure. At the same time, the steady-state performance of

Table 1. Characteristics of state-of-the-art BFT protocols tolerating f failures with batch size b. Bold entries mark the protocol with the lowest cost. The critical path denotes the number of one-way message delays. *Two message delays is only achievable with no concurrency.

	PBFT	Q/U	HQ	Zyzzyva	Aliph	Shuttle	BChain-3	BChain-5
Total replicas	$3f+1$	$5f+1$	$3f+1$	$3f+1$	$3f+1$	$\mathbf{2f+1}$	$3f+1$	$5f+1$
Crypto ops	$2+\frac{8f+1}{b}$	$2+8f$	$4+4f$	$2+\frac{3f}{b}$	$1+\frac{f+1}{b}$	$2+\frac{2f}{b}$	$\mathbf{1+\frac{3f+2}{b}}$	$1+\frac{4f+2}{b}$
Critical path	4	$\mathbf{2}^*$	4	3	$3f+2$	$2f+2$	$2f+2$	$3f+2$
Additional Requirements	None	None	None	Correct Clients	Protocol Switch	Olympus; Reconfig.	Reconfig.	None

BChain is comparable to Aliph-Chain, the state-of-the-art, chain-based BFT protocol. BChain also outperforms broadcast-based protocols PBFT [5] and Zyzzyva with a throughput improvement of up to 50% and 25%, respectively. We have used BChain to implement a BFT-based NFS service, and our evaluation shows that it is only marginally slower (1%) than a standard NFS implementation.

BChain in a nutshell. BChain is a self-recovering, chain-based BFT protocol, where the replicas are organized in a chain. In common case executions, clients send their requests to the head of the chain, which orders the requests. The ordered requests are forwarded along the chain and executed by the replicas. Once a request reaches a replica that we call the *proxy tail*, a reply is sent to the client.

When a BFT service experiences failures or asynchrony, BChain employs a novel approach that we call *re-chaining*. In this approach, the head reorders the chain when a replica is suspected to be faulty, so that a fault cannot affect the critical path.

To facilitate re-chaining, BChain makes use of a novel failure detection mechanism, where any replica can suspect its successor and only its successor. A replica does this by sending a signed suspicion message up the chain. No proof that the suspected replica has misbehaved is required. Upon receiving a suspicion, the head issues a new chain ordering where the accused replica is moved out of the critical path, and the accuser is moved to a position in which it cannot continue to accuse others. In this way, correct replicas help BChain make progress by suspecting faulty replicas, yet malicious replicas cannot *constantly* accuse correct replicas of being faulty.

Our re-chaining approach is inexpensive; a single re-chaining request corresponds to processing a single client request. Thus, the steady-state performance of BChain has minimal disruption. The latency reduction caused by re-chaining is dominated by the failure detection timeout.

Our contributions in context. We consider two variants of BChain—BChain-3 and BChain-5, both tolerating f failures. BChain-3 requires $3f+1$ replicas and a reconfiguration mechanism coupled with our detection and re-chaining algorithms, while BChain-5 requires $5f+1$ replicas, but can operate without the reconfiguration mechanism. We compare BChain-3 and BChain-5 with state-of-the-art BFT protocols in Table 1. All protocols use MACs for authentication and request batching with batch size b. The number of MAC operations for BChain at the bottleneck server tends to one for gracious executions. While this is also the case for Aliph-Chain [19], Aliph requires

that clients take responsibility for switching to another slower BFT protocol in the presence of failures, to ensure safety and liveness. Thus, a single dedicated adversary might render the system much slower. Shuttle [34] can tolerate f faulty replicas using only $2f+1$ replicas. However, it relies on a trusted auxiliary server. BChain does not require an auxiliary service, yet its critical path of $2f + 2$ is identical to that of Shuttle.

Our contributions can be summarized as follows:

1. We present BChain-3 and its sub-protocols for re-chaining, reconfiguration, and view change (§3). Re-chaining is a novel technique to ensure liveness in BChain. Together with re-chaining, the reconfiguration protocol can replace failed replicas with new ones, outside the critical path. The view change protocol deals with a faulty head.
2. We present BChain-5 and how it can operate without reconfiguration (§4).
3. In §5 we evaluate the performance of BChain for both gracious and uncivil executions under different workloads, and compare it with other BFT protocols. We also ran experiments with a BFT-NFS application and assessed its performance compared to the other relevant BFT protocols.

2 System Model

We assume a Byzantine fault tolerant system, where replicas communicate over pairwise channels and may behave arbitrarily. Our system can mask up to f faulty replicas, using n replicas. We write t, where $t \leq f$, to denote the number of faulty replicas that the system currently has. A computationally bounded adversary can coordinate faulty replicas to compromise safety only if more than f replicas are compromised.

Safety of our system holds in any asynchronous environment, where messages may be delayed, dropped, or delivered out of order. Liveness is ensured assuming *partial synchrony* [16]: synchrony holds only after some unknown global stabilization time, but the bounds on communication and processing delays are themselves unknown.

We use non-keyed *message digests*. The digest of a message m is denoted $D(m)$. We also use *digital signatures*. The signature of a message m signed by replica p_i is denoted $\langle m \rangle_{p_i}$. We say that a signature is *valid* on message m, if it passes the verification w.r.t. the public-key of the signer and the message. A vector of signatures of message m signed by a set of replicas $\mathcal{U} = \{p_i, \ldots, p_j\}$ is denoted $\langle m \rangle_{\mathcal{U}}$.

We classify the replica failures according to their behaviors. Weak semantics levy fewer restrictions on the possible behaviors than strong semantics. Apart from the weakest failure semantics (i.e., Byzantine failure), we are also interested in various other stronger failure semantics. *Crash failures*, occur when the replicas might halt permanently and no longer produce any output. By *timing failures*, we mean any replica failures that produce correct results but deliver them outside of a specified time window.

3 BChain-3

We now describe the main protocols and principles of BChain. Our description here uses digital signatures; later we show how they can be replaced with MACs, along with other optimizations. BChain-3 has five sub-protocols: (1) chaining, (2) re-chaining,

(3) view change, (4) checkpoint, and (5) reconfiguration. The *chaining* protocol orders clients requests, while *re-chaining* reorganizes the chain in response to failure suspicions. Faulty replicas are moved to the end of the chain. The *view change* protocol selects a new head when the current head is faulty, or the system is slow. Our *checkpoint* protocol is similar to that of PBFT [5]. It is used to bound the growth of message logs and reduce the cost of view changes. We do not describe it in this paper. The *reconfiguration* protocol is responsible for reconfiguring faulty replicas.

To tolerate f failures, BChain-3 needs n replicas such that $f \leq \lfloor \frac{n-1}{3} \rfloor$. In the following, we assume $n = 3f + 1$ for simplicity.

3.1 Conventions and Notations

In BChain, the replicas are organized in a metaphorical *chain*, as shown in Figure 1. Each replica is uniquely identified from a set $\Pi = \{p_1, p_2, \cdots, p_n\}$. Initially, we assume that replica IDs are numbered in ascending order. The first replica is called the *head*, denoted p_h, the last replica is called the *tail*, and the $(2f+1)^{\text{th}}$ replica is called the *proxy tail,* denoted p_p. We divide the replicas into two subsets. Given a specific chain order, $\mathcal{A}$ contains the first $2f+1$ replicas, initially p_1 to p_{2f+1}. $\mathcal{B}$ contains the last f replicas in the chain, initially p_{2f+2} to p_{3f+1}. For convenience, we also define $\mathcal{A}^{\not p} = \{\mathcal{A} \setminus p_p\}$, excluding the proxy tail, and $\mathcal{A}^{\not h} = \{\mathcal{A} \setminus p_h\}$, excluding the head.

The chain order is maintained by every replica and can be changed by the head and is communicated to replicas through message transmissions. (This is in contrast to Aliph-Chain, where the chain order is fixed and known to all replicas and clients beforehand.) For any replica except the head, $p_i \in \mathcal{A}^{\not h}$, we define its *predecessor* $\overleftarrow{p}_i$, initially p_{i-1}, as its preceding replica in the current chain order. For any replica except the proxy tail, $p_i \in \mathcal{A}^{\not p}$, we define its *successor* $\overrightarrow{p}_i$, initially p_{i+1}, as its subsequent replica in the current chain order.

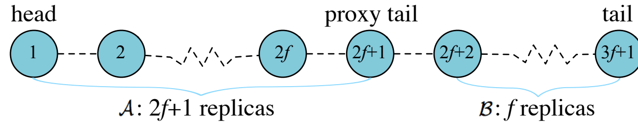


Fig. 1. BChain-3. Replicas are organized in a chain.

For each $p_i \in \mathcal{A}$, we define its *predecessor set* $\mathcal{P}(p_i)$ and *successor set* $\mathcal{S}(p_i)$, whose elements depend on their individual positions in the chain. If a replica $p_i \neq p_h$ is one of the first $f+1$ replicas, its predecessor set $\mathcal{P}(p_i)$ consists of all the preceding replicas in the chain. For every other replica in $\mathcal{A}$, the predecessor set $\mathcal{P}(p_i)$ consists of the preceding $f+1$ replicas in the chain. If p_i is one of the last $f+1$ replicas in $\mathcal{A}$, the successor set $\mathcal{S}(p_i)$ consists of all the subsequent replicas in $\mathcal{A}$. For every other replica in $\mathcal{A}$, the successor set $\mathcal{S}(p_i)$ consists of the subsequent $f+1$ replicas. Note that the cardinality of any replica's predecessor set or successor set is at most $f+1$.

3.2 Protocol Overview

In a gracious execution, as shown in Figure 2, the first $2f+1$ replicas (set $\mathcal{A}$) reach an agreement while the last f replicas (set $\mathcal{B}$) correspondingly update their states based on

the agreed-upon requests from set $\mathcal{A}$. BChain transmits two types of messages along the chain: $\langle$CHAIN$\rangle$ messages transmitted from the head to the proxy tail, and $\langle$ACK$\rangle$ messages transmitted in reverse from the proxy tail to the head. A request is *executed* after a replica accepts the $\langle$CHAIN$\rangle$ message; a request *commits* at a replica if it accepts the $\langle$ACK$\rangle$ message.

Upon receiving a client request, the head sends a $\langle$CHAIN$\rangle$ message representing the request to its successor. As soon as the proxy tail accepts the $\langle$CHAIN$\rangle$ message, it sends a reply to the client and generates an $\langle$ACK$\rangle$ message, which is sent backwards along the chain until it reaches the head. Once a replica in $\mathcal{A}$ accepts the $\langle$ACK$\rangle$ message, it completes the request and forwards its $\langle$CHAIN$\rangle$ message to replicas in $\mathcal{B}$ to ensure that the message is committed at all the replicas.

To handle failures and ensure liveness, BChain incorporates failure detection and re-chaining protocol that works as follows: Every replica in $\mathcal{A}^{\not{p}}$ starts a timer after sending a $\langle$CHAIN$\rangle$ message. Unless an $\langle$ACK$\rangle$ is received before the timer expires, it sends a $\langle$SUSPECT$\rangle$ message to the head and also along the chain towards the head. Upon seeing $\langle$SUSPECT$\rangle$ messages, the head starts the re-chaining, by moving faulty replicas to set $\mathcal{B}$ where, if needed, replicas may be replaced in the reconfiguration protocol. In this way, BChain remains robust until new failures occur.

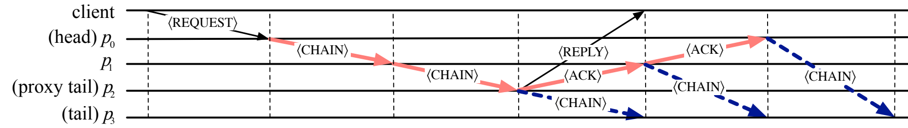


Fig. 2. BChain-3 common case communication pattern. All the signatures can be replaced with MACs. All the $\langle$CHAIN$\rangle$ and $\langle$ACK$\rangle$ messages can be batched. The $\langle$CHAIN$\rangle$ messages with dotted, blue lines are the forwarded messages that are stored in logs. *No* conventional broadcast is used at any point in our protocol. For a given batch size b.

3.3 Chaining

We now describe the sequence of steps of the chaining protocol, used to order requests, when there are no failures.

Step 0: *Client sends a request to the head.* A client c requests the execution of state machine operation o by sending a request $m = \langle \text{REQUEST}, o, T, c \rangle_c$ to the replica that it believes to be the head, where T is the timestamp.

Step 1: *Assign sequence number and send chain message.* When the head p_h receives a valid $\langle \text{REQUEST}, o, T, c \rangle_c$ message, it assigns a sequence number and sends message $\langle \text{CHAIN}, v, ch, N, m, c, \mathcal{H}, R, \Lambda \rangle_{p_h}$ to its successor, where v is the view number, ch is the number of re-chainings that took place during view v, $\mathcal{H}$ is the hash of its execution history, R is the hash of the reply r to the client containing the execution result, and Λ is the current chain order. Both of $\mathcal{H}$ and R are empty in this step.

Step 2: *Execute request and send chain message.* A replica p_j receives from its predecessor a valid $\langle \text{CHAIN}, v, ch, N, m, c, \mathcal{H}, R, \Lambda \rangle_{\mathcal{P}(p_j)}$ message, which contains valid signatures by replicas in $\mathcal{P}(p_j)$. The replica p_j updates $\mathcal{H}$ and R fields if necessary, appends its signature to the $\langle$CHAIN$\rangle$ message, and sends to its successor. Note that the

$\mathcal{H}$ and R fields are empty if p_j is among the first f replicas, and both $\mathcal{H}$ and R must be verified before proceeding.

Each time a replica $p_j \in \mathcal{A}^{\not{p}}$ sends a $\langle$CHAIN$\rangle$ message, it sets a timer, expecting an $\langle$ACK$\rangle$ message, or a $\langle$SUSPECT$\rangle$ message signaling some replica failures.

Step 3: *Proxy tail sends reply to the client and commits the request.* If the proxy tail p_j accepts a $\langle$CHAIN$\rangle$ message, it computes its own signature and sends the client the reply r, along with the $\langle$CHAIN$\rangle$ message it accepts. It also sends to its predecessor an $\langle \text{ACK}, v, ch, N, D(m), c\rangle_{p_j}$ message. In addition, it *forwards* to all replicas in $\mathcal{B}$ the corresponding $\langle \text{CHAIN}, v, ch, N, m, c, \mathcal{H}, R, \Lambda\rangle_{p_j}$ message . The request commits at the proxy tail.

Step 4: *Client completes the request or retransmits.* The client completes the request if it receives a $\langle$REPLY$\rangle$ message from the proxy tail with signatures by the last $f + 1$ replicas in the chain. Otherwise, it retransmits the request to all replicas.

Step 5: *Other replicas in $\mathcal{A}$ commit the request.* A valid $\langle \text{ACK}, v, ch, N, D(m), c\rangle_{\mathcal{S}(p_j)}$ message is sent to replica p_j by its successor, which contains valid signatures by replicas in $\mathcal{S}(p_j)$. The replica appends its own signature and sends to its predecessor.

Step 6: *Replicas in $\mathcal{B}$ execute and commit request.* The replicas in $\mathcal{B}$ collects $f + 1$ matching $\langle$CHAIN$\rangle$ messages, and executes the operation, completing the request. Thus, the request commits at each correct replica in $\mathcal{B}$.

3.4 Re-chaining

To facilitate failure detection and ensure that BChain remains live, we introduce a protocol we call *re-chaining*. With re-chaining, we can make progress with a bounded number of failures, despite incorrect suspicions. The algorithm ensures that, eventually all faulty replicas are identified and appropriately dealt with. The strategy of the re-chaining algorithm is to move replicas that are *suspected* to set $\mathcal{B}$, where if deemed necessary, they are rejuvenated.

Algorithm 1 Failure detector at replica p_i

1: **upon** $\langle$CHAIN$\rangle$ sent by p_i
2:     $starttimer(\Delta_{1,p_i})$
3: **upon** $\langle$Timeout, $\Delta_{1,p_i}\rangle$ {Accuser p_i}
4:     send $\langle \text{SUSPECT}, \overrightarrow{p}_i, m, ch, v\rangle_{p_i}$ to $\overleftarrow{p}_i$ and p_h
5: **upon** $\langle$ACK$\rangle$ from $\overrightarrow{p}_i$
6:     $canceltimer(\Delta_{1,p_i})$
7: **upon** $\langle \text{SUSPECT}, p_y, m, ch, v\rangle$ from $\overrightarrow{p}_i$
8:     forward $\langle \text{SUSPECT}, p_y, m, ch, v\rangle$ to $\overleftarrow{p}_i$
9:     $canceltimer(\Delta_{1,p_i})$

BChain failure detector. The objective of the BChain failure detector is to identify faulty replicas, and issue a new chain configuration and to ensure that progress can be made. It is implemented as a timer on $\langle$CHAIN$\rangle$ messages, as shown in Algorithm 1. On sending a $\langle$CHAIN$\rangle$ message m, replica p_i starts a timer, Δ_{1,p_i}. If the replica receives an $\langle$ACK$\rangle$ for the message before the timer expires, it cancels the timer and starts a new one for the next request in the queue, if any. Otherwise, it sends a $\langle \text{SUSPECT}, \overrightarrow{p}_i, m, ch, v\rangle$ to both the head and its predecessor to signal the failure of its successor. Moreover, if p_i receives a $\langle$SUSPECT$\rangle$ message from its successor, the message is forwarded to p_i's

predecessor, along the chain until it reaches the head. To prevent that a faulty replica fails to forward the ⟨SUSPECT⟩ message, it is also sent directly to the head. Passing it along the chain allows us to cancel timers and reduce the number of suspect messages.

Let p_i be the *accuser*; then the *accused* can only be its successor, $\vec{p}_i$. This is ensured by having the accuser sign the ⟨SUSPECT⟩ message, just as an ⟨ACK⟩ message.

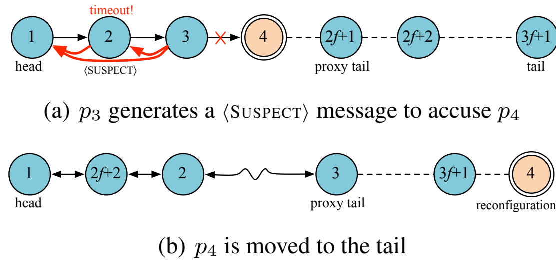
Algorithm 2 BChain-3 Re-chaining-I (At head, p_h)

1: **upon** $\langle\text{SUSPECT}, p_y, m, ch, v\rangle$ from p_x
2: **if** $p_x \neq p_h$ **then** {p_x is not the head}
3: p_z is put to the 2nd position {$p_z = \mathcal{B}[1]$}
4: p_x is put to the $(2f+1)^{\text{th}}$ position
5: p_y is put to the end

On receiving a ⟨SUSPECT⟩, the head starts re-chaining via a new ⟨CHAIN⟩ message. If the head receives multiple ⟨SUSPECT⟩ messages, only the one *closest* to the proxy tail is handled. Handling a ⟨SUSPECT⟩ message is done by increasing ch, selecting a new chain order Λ, and sending a ⟨CHAIN⟩ message to order the same request again.

Re-chaining algorithms. We provide two re-chaining algorithms for BChain-3 as shown in Algorithm 2 and 3. To explain these algorithms, assume that the head, p_h, has received a ⟨SUSPECT⟩ message from a replica p_x suspecting is successor p_y. Let p_z be the first replica in set $\mathcal{B}$. Both algorithms show how the head selects a new chain order. Both are *efficient* in the sense that the number of re-chainings needed is proportional to the number of existing failures t instead of the maximum number f. We levy no assumptions on how failures are distributed in the chain.

Re-chaining-I—crash failures handled first. Algorithm 2 is reasonably efficient; in the worst case, t faulty replicas can be removed with at most $3t$ re-chainings. More specifically, if the head is correct and $3t \leq f$, the faulty replicas are moved to the end of chain after at most $3t$ re-chainings; if $3t > f$, at most $3t$ re-chainings are necessary and at most $3t - f$ replicas are replaced in the reconfiguration protocol (§3.6), assuming that any individual replica can be reconfigured within f re-chainings. Algorithm 2 is even more efficient when handling timing and omission failures, with one such replica being removed using only one re-chaining. Despite the succinct algorithm, the proof of correctness for the general case is complicated. We omit the details due to lack of space. To help grasp the underlying idea, consider the following *simple* examples.



(a) p_3 generates a ⟨SUSPECT⟩ message to accuse p_4

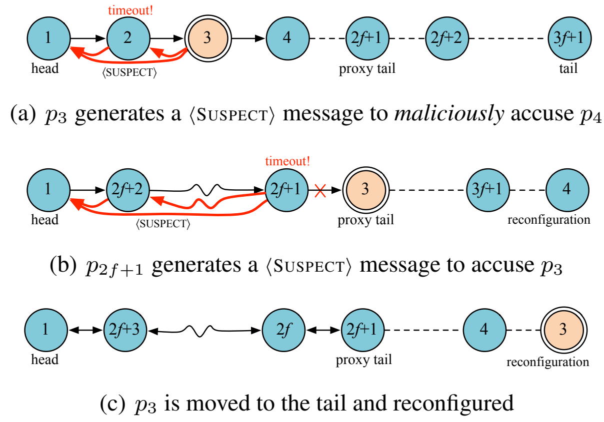
(b) p_4 is moved to the tail

Fig. 3. Example (1). A faulty replica is denoted by a double circle. After the timer expires, replica p_3 issues a ⟨SUSPECT⟩ message to accuse p_4 (which is faulty). The head moves p_3 to the proxy tail position and the faulty replica p_4 to the tail.

▷ Example (1): In Figure 3, replica p_4 has a timing failure. This causes p_3 to send a ⟨SUSPECT⟩ message up the chain to accuse p_4. According to our re-chaining algorithm, p_3 is moved to the $(2f+1)^{\text{th}}$ position and becomes the proxy tail, and p_4 is moved

to the end of the chain and becomes the tail. Our fundamental design principle is that timing failures should be given top priority.

▷ Example (2): In Figure 4, p_3 is the only faulty replica. We consider the circumstance where p_3 sends the head a ⟨SUSPECT⟩ message to frame its successor p_4 even if p_4 follows the protocol. According to our re-chaining algorithm, replica p_4 will be moved to the tail, while p_3 becomes the new proxy tail. However, from then on, p_3 can no longer accuse any replicas. It either follows the specification of the protocol, or chooses not to participate in the agreement, in which case p_3 will be moved to the tail. The example illustrates another important designing rationale that an adversarial replica cannot constantly accuse correct replicas.



(a) p_3 generates a ⟨SUSPECT⟩ message to *maliciously* accuse p_4

(b) p_{2f+1} generates a ⟨SUSPECT⟩ message to accuse p_3

(c) p_3 is moved to the tail and reconfigured

Fig. 4. Example (2). Replica p_3 maliciously accuse p_4 by sending a ⟨SUSPECT⟩ message. The head moves p_3 to the proxy tail and p_4 to the tail. If p_3 does not behave, it will be accused by its predecessor p_{2f+1} such that in another round of re-chaining p_3 is moved to the tail.

Re-chaining-II—improved efficiency. Algorithm 3 can improve efficiency for the *worst* case. The underlying idea is simple: every time the head receives a ⟨SUSPECT⟩ message, both the accuser and the accused are moved to the end of the chain. Algorithm 3 does not prioritize crash failures, and relies on a stronger reconfiguration assumption. If the head is correct and $2t \leq f$, the faulty replicas are moved to the end of chain after at most $2t$ re-chainings; if $2t > f$, at most $2t$ re-chainings are necessary and at most $2t - f$ replica reconfigurations (§3.6) are needed, assuming that any individual replica can be reconfigured within $\lfloor f/2 \rfloor$ re-chainings. When an accused replica is moved to the end of chain, the reconfiguration process is initialized, either offline or online. The replicas moved to the end of the chain are all "tainted" and reconfigured, as we discuss in §3.6.

Algorithm 3 BChain-3 Re-chaining-II

1: **upon** $\langle \text{SUSPECT}, p_y, m, ch, v \rangle$ from p_x
2:     **if** $p_x \neq p_h$ **then**   {p_x is not the head}
3:         p_x is put to the $(3f)^{\text{th}}$ position
4:     p_y is put to the end

Timer setup and preventing timer-based performance attacks. Existing BFT protocols typically only keep timers for view changes, while BChain also requires timers for ⟨ACK⟩ and ⟨CHAIN⟩ messages. To achieve accurate failure detection, we need different values for each timer in each replica in the chain.

The timeout for each replica $p_i \in \mathcal{A}$ is defined as $\Delta_{1,i} = \mathcal{F}(\Delta_1, l_i)$, where $\mathcal{F}$ is a fixed and efficiently computable function, Δ_1 is the base timeout, and l_i is p_i's position in the chain order. Note that for p_h, we have that $l_h = 1$ and thus $\mathcal{F}(\Delta_1, 1) =$

Δ_1. Correspondingly, for p_p we have that $l_p = 2f + 1$ and $\mathcal{F}(\Delta_1, 2f + 1) = 0$. It is reasonable to adopt a *linear function* with respect to the position of each replica as the timer function, e.g., $\mathcal{F}(\Delta_1, l_i) = \frac{2f+1-l_i}{2f}\Delta_1$. As an example with $n=4$ and $f=1$, we may set $\Delta_{1,p_1} = \mathcal{F}(\Delta_1, 1) = \Delta_1$, $\Delta_{1,p_2} = \mathcal{F}(\Delta_1, 2) = \Delta_1/2$, and $\Delta_{1,p_3} = \mathcal{F}(\Delta_1, 3) = 0$.

To detect and deter misbehaving replicas that always delay requests to the upper bound timeout value to increase system latency, we also verify the processing delays for the average case and allow replicas to suspect other replicas who frequently do so. Concretely, each replica p_i maintains an additional performance threshold timer Δ'_{1,p_i} such that $\Delta'_{1,p_i} < \Delta_{1,p_i}$, which is used to detect slow or faulty replicas as mentioned above. That is, we ask the replica to further suspect its successor if their average delay exceeds Δ'_{1,p_i}. This will allow us to thwart dedicated performance attacks on messages delays while preventing temporarily slow replicas from being accused prematurely. We will show in §5.1 how to efficiently set up and maintain the timers in actual experiments.

3.5 View Change

The view change protocol has two functions: (1) to select a new head when the current head is deemed faulty, and (2) to adjust the timers to ensure eventual progress, despite deficient initial timer configuration.

A correct replica p_i votes for view change if either (1) it suspects the head to be faulty, or (2) it receives $f + 1$ ⟨VIEWCHANGE⟩ messages. The replica votes for view change and moves to a new view by sending all replicas a ⟨VIEWCHANGE⟩ message that includes the new view number, the current chain order, a set of valid checkpoint messages, and a set of requests that commit locally with proof of execution. For each request that commits locally, if $p_i \in \mathcal{A}$, then a proof of execution for a request contains a ⟨CHAIN⟩ message with signatures from $\mathcal{P}(p_i)$ and an ⟨ACK⟩ message with signatures from $\mathcal{S}(p_i)$. Otherwise, a proof of execution contains $f + 1$ ⟨CHAIN⟩ messages. Upon sending a ⟨VIEWCHANGE⟩ message, p_i stops receiving messages except ⟨CHECKPOINT⟩, ⟨NEWVIEW⟩, or other ⟨VIEWCHANGE⟩ messages. When the new head collects $2f + 1$ ⟨VIEWCHANGE⟩ messages, it sends all replicas a ⟨NEWVIEW⟩ message which includes the new chain order, in which the head of the old view has been moved to the end of the chain, a set of valid ⟨VIEWCHANGE⟩ messages, and a set of ⟨CHAIN⟩ messages.

The other function of view change is to adjust the timers. In addition to the timer Δ_1 maintained for re-chaining, BChain has two timers for view changes, Δ_2 and Δ_3. Δ_2 is a timer maintained for the current view v when a replica is waiting for a request to be committed, while Δ_3 is a timer for ⟨NEWVIEW⟩, when a replica votes for a view change and waits for the ⟨NEWVIEW⟩. Algorithm 4 describes how to initialize, maintain, and adjust these timers.

The view change timer Δ_2 at a replica is set up for the first request in the queue. A replica sends a ⟨VIEWCHANGE⟩ message to all replicas and votes for view change if Δ_2 expires or it receives $f + 1$ ⟨VIEWCHANGE⟩ messages. In either case, when a replica votes for view change, it cancels its timer Δ_2. After a replica collects $2f + 1$ ⟨VIEWCHANGE⟩ messages (including its own), it starts a timer Δ_3 and waits for the ⟨NEWVIEW⟩ message. If the replica does not receive ⟨NEWVIEW⟩ message before Δ_3 expires, it starts a *new* ⟨VIEWCHANGE⟩ and updates Δ_3 with a new value $g_3(\Delta_3)$. When a replica receives the ⟨NEWVIEW⟩ message, it sets Δ_1 and Δ_2 using $g_1(\Delta_1)$ and $g_2(\Delta_2)$,

Algorithm 4 View Change Handling and Timers at p_i

```
1: Δ2 ← init_Δ2;   Δ3 ← init_Δ3
2: voted ← false
3: upon ⟨Timeout, Δ2⟩
4:     send ⟨VIEWCHANGE⟩
5:     voted ← true
6: upon f + 1 ⟨VIEWCHANGE⟩ ∧ ¬voted
7:     send ⟨VIEWCHANGE⟩
8:     voted ← true
9:     canceltimer(Δ2)
10: upon 2f + 1 ⟨VIEWCHANGE⟩
11:     starttimer(Δ3)
12: upon ⟨Timeout, Δ3⟩
13:     Δ3 ← g3(Δ3)
14:     send new ⟨VIEWCHANGE⟩
15: upon ⟨NEWVIEW⟩
16:     canceltimer(Δ3)
17:     Δ1 ← g1(Δ1)
18:     Δ2 ← g2(Δ2)
```

respectively. In practice, the functions $g_1(\cdot)$, $g_2(\cdot)$, and $g_3(\cdot)$ could simply double the current timeouts. However, to avoid the circumstance that the timeouts for $\varDelta_1$ and $\varDelta_2$ increase without bound, we introduce upper bounds for both of them. Once either timer exceeds the prescribed bound, the system starts reconfiguration.

3.6 Reconfiguration

Reconfiguration [26] is a general technique, often abstracted as stopping the current state machine and restarting it with a new set of replicas, usually reusing non-faulty replicas in the new configuration. In BChain we use reconfiguration in concert with re-chaining to replace faulty replicas with new ones. The reconfiguration operates *out-of-band* in the $\mathcal{B}$ replica set, and imposes only negligible overhead on client request processing being done by replicas in $\mathcal{A}$. We omit the details due to lack of space.

3.7 Optimizations

In general, signatures for ⟨CHAIN⟩ and ⟨ACK⟩ cannot be replaced with MACs. However, we can replace other signatures with MACs. Moreover, we can combine all-MAC-based and signature-based BChain approaches such that the failure-free case uses MACs only and re-chaining uses signatures. We also developed a highly efficient *purely* MAC-based variant of BChain for $n=4$ and $f=1$, which does not rely on reconfiguration.

4 BChain without Reconfiguration

We now discuss BChain-5, which uses $n = 5f + 1$ replicas to tolerate f Byzantine failures, just as Q/U [1] and Zyzzyva5 [24]. With $5f + 1$ replicas at our disposal, we design an efficient re-chaining algorithm, which allows the faulty replicas to be identified easily without relying on reconfiguration. Meanwhile, a Byzantine quorum of replicas can reach agreement. BChain-5 relies on the concept of Byzantine quorum protocols [28]. Set $\mathcal{A}$ is a Byzantine quorum which consists of $\lceil \frac{n+f+1}{2} \rceil = 3f + 1$ replicas, while set $\mathcal{B}$ consists of the remaining of $2f$ replicas.

BChain-5 has four sub-protocols: chaining, re-chaining, view change, and checkpoint. In contrast, BChain-3 additionally requires a reconfiguration protocol. The protocols for BChain-3 and BChain-5 are identical with respect to message flow. The main

difference lies in the size of the $\mathcal{A}$ set, which now consists of $3f+1$ replicas. BChain-5 also uses Algorithm 3, modifying only Line 3 to put p_x to the $(5f)^{\text{th}}$ position.

Assuming the timers are accurately configured and that the head is non-faulty, it takes at most f re-chainings to move f failures to the tail set $\mathcal{B}$. The proofs for safety and liveness of BChain-5 are easier than those of BChain-3 due to a different re-chaining algorithm and the absence of the reconfiguration procedure.

To reconfigure or not to reconfigure? The primary benefit of BChain-5 over BChain-3 is that it eliminates the need for reconfiguration to achieve liveness. This is beneficial, since reconfiguration needs additional resources, such as machines to host reconfigured replicas. However, since BChain-5 can identify and move faulty replicas to the tail set $\mathcal{B}$, we can still leverage the reconfiguration procedure on the replicas in $\mathcal{B}$, to provide long-term system safety and liveness. This does not contradict the claim that BChain-5 does not need reconfiguration; rather, it just makes the system more robust. Furthermore, BChain-5 provides flexibility with respect to when the system should be reconfigured. Specifically, reconfiguration can happen any time after the system achieves a stable state or simply has run for a "long enough" period of time.

5 Evaluation

This section studies the performance of BChain-3 and BChain-5 and compares them with three well-known BFT protocols—PBFT [5], Zyzzyva [24], and Aliph [19]. Aliph uses Chain for gracious execution under high concurrency. Aliph-Chain enjoys the highest throughput when there are no failures, however, as we will see, it cannot sustain its performance during failure scenarios by itself, where BChain is superior.

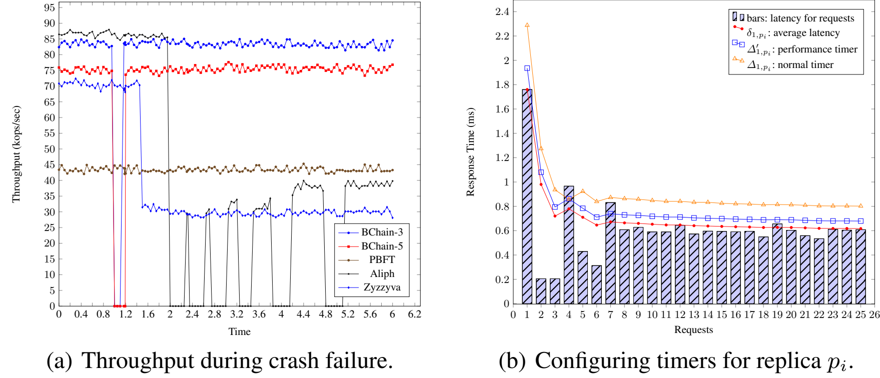
We study the performance using two types of benchmarks: the micro-benchmarks by Castro and Liskov [5] and the Bonnie++ benchmark [10]. We use micro-benchmarks to assess throughput, latency, scalability, and performance during failures of all the five protocols. In the x/y micro-benchmarks, clients send x kB requests and receive y kB replies. Clients invoke requests in a *closed-loop*, where a client does not start a new request before receiving a reply for a previous one. All the protocols implement batching of concurrent requests to reduce cryptographic and communication overheads.

All experiments were carried out on DeterLab [4], utilizing a cluster of up to 65 identical machines equipped with a 2.13 GHz Xeon processor and 4GB of RAM. They are connected through a 100Mbps switched LAN.

We have assessed the performance of all protocols under gracious execution, and find that both BChain-3 and BChain-5 achieve higher throughput and lower latency than PBFT and Zyzzyva especially when the number of concurrent client requests is large, while BChain-3 has performance similar to the Aliph-Chain protocol. Our experiment bolsters the point of view of Guerraoui *et al*. [19] that (authenticated) chaining replication can increase throughput and reduce latency under high concurrency. We omit the detailed evaluation for gracious execution.

In addition to micro-benchmarks, we have also evaluated a BFT-NFS service implemented using PBFT [5], Zyzzyva [24], and BChain-3. We show that performance overhead of BChain-3, with and without failure, is low, both compared to unreplicated NFS and other BFT implementations.

In case of failures, both BChain-3 and BChain-5 outperform all the other protocols by a wide margin, due to BChain's unique re-chaining protocol. Through the timeout adjustment scheme, we show that a faulty replica cannot reduce the performance of the system by manipulating the timeouts.



(a) Throughput during crash failure. (b) Configuring timers for replica p_i.

Fig. 5. Performance under failure.

5.1 Performance under Failures

We compare the performance of BChain with the other BFT protocols under two scenarios: a simple crash failure scenario and a performance attack scenario. As the results in Figure 5(a) show, BChain has superior reaction to failures. When BChain detects a failure, it will start re-chaining. At the moment when re-chaining starts, the throughput of BChain temporarily drops to zero. After the chain has been re-ordered, BChain quickly recovers its steady state throughput. The dominant factor deciding the duration of this throughput drop (i.e. increased latency) is the failure detection timeout, not the re-chaining. We also show that BChain can resist a timer-based performance attack, i.e., a faulty replica cannot intentionally manipulate timeouts to slow down the system.

Crash failure. We compare the throughput during crash failure for BChain-3, BChain-5, PBFT, Zyzzyva, and Aliph. The results are shown in Figure 5(a). We use $f = 1$, message batching, and 40 clients. To avoid clutter in the plot, we used different failure inject times for the protocols: BChain-3, BChain-5, and PBFT all experience a failure at 1s, while Zyzzyva and Aliph experience a failure at 1.5s and 2s, respectively.

We note that Aliph [19, 36] generally switches between three protocols: Quorum, Chain, and a backup, e.g., PBFT. For our experiments, we adopt the same setting as Aliph paper [19], i.e., it uses a combination of Chain and PBFT as backup and a configuration parameter k, denoting the number of requests to be executed when running with the backup protocol. We use both $k = 1$ and $k = 2^i$.

Even though Aliph exhibits slightly higher throughput than BChain-3 prior to the failure, its throughput takes a significant beating upon failure, dropping well below that of the PBFT baseline. The overall performance depends on how often failures occur and how often Aliph switches between main and backup protocols, i.e., parameter k. On the other hand, the throughput of PBFT does not change in any obvious way after

failure injection, showing its stability during failure scenarios. Zyzzyva, in comparison, in the presence of failures, uses its slower backup mode (i.e., clients collects and sends certificate) which exhibits even lower throughput than PBFT.

We configured BChain with a fairly high timeout value (100ms). In fact, BChain can use much smaller timeouts, since one re-chaining only takes about the same time as it takes for BChain to process a single request. While the signature-based, view-change like switching taken by Aliph introduces a significant time overhead.

We claim that even in presence of a Byzantine failure, the throughput of BChain-3 and BChain-5 would not significantly change, except that there might be two (instead of one) short periods where the throughput drops to zero. Note BChain-3 uses at most two re-chainings to handle a Byzantine faulty replica, while BChain-5 uses only one.

Timer setup and performance attack evaluation. We now show how to set up the timers for replicas in the chain as discussed in §3.4. Initially, there are no faulty replicas and we set the timers based on the average latency of the first 1000 requests. Figure 5(b) illustrates the timer setup procedure for a correct replica p_i, where each bar represents the actual latency of a request, line 1 is the average latency δ_{1,p_i}, line 2 is the performance threshold timer Δ'_{1,p_i} used to deter performance attacks, and line 3 is the normal timer Δ_{1,p_i}. In our experiment, we set $\Delta'_{1,p_i} = 1.1\delta_{1,p_i}$ and $\Delta_{1,p_i} = 1.3\delta_{1,p_i}$. That is, we expect the performance reduction to be bounded to 10% of the actual latency during a performance attack by a dedicated adversary.

To evaluate the robustness against a timer-based performance attack, we ran 10 experiments using the $0/0$ benchmark, each with a sequence of 10000 requests. We assume there are no faulty replicas initially and we use the first 1000 request to train the timers. For each experiment, starting from the 1001th request, we let a replica mount a performance attack by intentionally delaying messages sent to its predecessor. To simulate different attacks, we simply let the faulty replica sleep for an "appropriate" period of time following different strategies. However, as expected our findings show that the actions of a faulty replica is very limited: it either needs to be very careful not to be accused, thus imposing only a marginal performance reduction, or it will be suspected which will lead to a re-chaining and then a reconfiguration.

5.2 A BFT Network File System

We evaluate a BFT-NFS service implemented using PBFT [5], Zyzzyva [24], and BChain-3, respectively. The BFT-NFS service exports a file system, which can then be mounted on a client machine. Upon receiving client requests, the replication library and the NFS daemon is called to reach agreement on the order in which to process client requests. Once processing is done, replies are sent to clients. The NFS daemon is implemented using a fixed-size memory-mapped file.

Table 2. NFS DirOps evaluation in fault-free cases.

BChain-3	Zyzzyva	BFS	NFS-std
$41.66s(1.10\%)$	$42.47s(2.99\%)$	$43.04s(4.27\%)$	$41.20s$

We use the Bonnie++ benchmark [10] to compare our three implementations with NFS-std, an unreplicated NFS V3 implementation, using an I/O intensive workload. We

evaluate the Bonnie++ benchmark with the following directory operations (DirOps): (1) create files in numeric order; (2) stat() files in the same order; (3) delete them in the same order; (4) create files in an order that will appear random to the file system; (5) stat() random files; (6) delete the files in random order. We measure the average latency achieved by the clients while up to 20 clients run the benchmark concurrently. As shown in Table 2, the latency achieved by BChain-3 is 1.10% lower than NFS-std, in contrast to BFS and Zyzzyva.

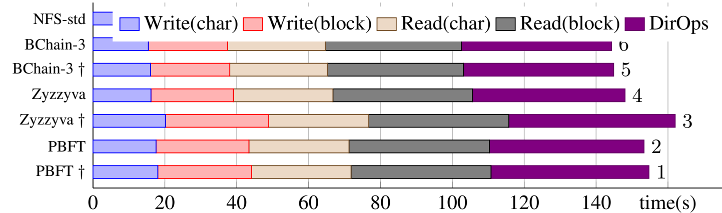


Fig. 6. NFS Evaluation with the Bonnie++ benchmark. The † symbol marks experiments with one failure.

We also evaluate the performance using the Bonnie++ benchmark when a failure occurs at time zero, as detailed in Figure 6. The bar chart also includes data points for the non-faulty case. The results shows that BChain can perform well even with failures, and is better than the other protocols for this benchmark.

6 Related Work

Failure detectors were introduced by Chandra and Toueg [7] for solving consensus problems in the presence of crash failures. For each replica, failure detector outputs the identities of each replica that it detects to have crashed. Quiet process and muteness detector [3, 13, 14, 27] extend failure detectors to address Byzantine failures and use them to solve consensus problem. Byzantine failures, in contrast to crash failures, are not context-free, so it is not possible to define and design failure detectors independently of the underlying protocols [14]. Therefore, for instance, consensus protocols from a muteness detector [13] have to handle Byzantine failures at the algorithmic level.

Fault diagnosis [2, 23, 29, 30, 32, 33, 37] aims to identify faulty replicas. The basic idea is that a *proof of misbehavior* for a is collected by executing a modified BFT protocol. However, it usually requires several rounds of protocols to collect a huge volume of exchanged messages to provide such proof. An adversary can render the system even less practical by intermittently following and violating the protocol specification. Similarly, PeerReview [20] can detect and deter failures by exploiting accountability. It also uses a “sufficient” number of witnesses to discover faulty ones. BChain fault diagnosis, though *not* perfectly accurate, does not have the above-mentioned properties. No evidence is required to be regularly collected, and no additional latency is induced by intermittent adversaries. We note that Hirt, Maurer, and Przydatek [22] used the idea of the “imperfect fault detection” to achieve general multi-party computation in synchronous environments, but their techniques are very different from ours.

7 Conclusion

We have presented BChain, a new chain-based BFT protocol that outperforms prior protocols in fault-free cases and especially during failures. In the presence of failures, instead of switching to a slower, backup BFT protocol, BChain leverages a novel technique—re-chaining—to efficiently detect and deal with the failures such that it can quickly recover its steady-state performance. BChain does not rely on any trusted components or unproven assumptions.

Acknowledgement

This research was supported in part by the National Science Foundation under grants CCF-1018871 and CNS-1228828. Hein Meling was supported by the Tidal News project under grant number 201406 from the Research Council of Norway. The authors thank Tiancheng Chang, Matt Franklin, Leander Jehl, Karl Levitt, Keith Marzullo, Phil Rogaway, Marko Vukolic, and anonymous reviewers for their helpful comments.

References

1. M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. *SOSP 2005*, pp. 59–74, ACM Press, 2005.
2. J. Adams and K. Ramarao. Distributed diagnosis of Byzantine processors and links. *ICDCS 1989*, pp. 562–569, IEEE Computer Society, 1989.
3. R. Baldoni, J. Helary, and M. Raynal. From crash fault-tolerance to arbitrary-fault tolerance: towards a modular approach. *DSN 2000*, pp. 273–282, 2000.
4. T. Benzel. The science of cyber security experimentation: the DETER project. *ACSAC*, 2011.
5. M. Castro and B. Liskov. Practical Byzantine fault tolerance. *OSDI*, pp. 173–186, 1999.
6. T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4): 685–722, 1996.
7. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2): 225–267, March 1996.
8. M. Chiang, S. Wang, and L. Tseng. An early fault diagnosis agreement under hybrid fault model. *Expert Syst. Appl*, 36(3): 5039–5050, 2009.
9. A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. *NSDI 2009*, pp. 153–168, USENIX Association, 2009.
10. R. Coker. www.coker.com.au/bonnie++.
11. A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. *SOSP '09*, pp. 277–290, ACM press, 2009.
12. J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. *OSDI*, pp. 177–190, USENIX Assn., 2006.
13. A. Doudou and A. Schiper. Muteness failure detectors for consensus with Byzantine processes, *Brief announcement in PODC*, pp. 315, ACM press, 1998.
14. A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. *Proc. Third EDCC*, LNCS vol. 1667, pp. 71–87, Springer, 1999.
15. A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. *Ada-Europe 2002*, pp. 24–50, Springer, 2002.

16. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM* 35(2): 288–323, 1988.
17. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM* 32(2): 374–382, 1985.
18. S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *SOSP*, pp. 29–43, 2003.
19. R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. *EuroSys 2010*, pp. 363–376, ACM, 2010.
20. A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. *SOSP 2007*, pp. 175–188, ACM, 2007.
21. J. Hendricks, S. Sinnamohideen, G. Ganger, and M. Reiter. Zzyzx: scalable fault tolerance through Byzantine locking. *DSN 2010*, pp. 363–372, IEEE Computer Society, 2010.
22. M. Hirt, U. Maurer, B. Przydatek. Efficient secure multi-party computation. *ASIACRYPT 2000*, pp. 143–161, 2000.
23. H. Hsiao, Y. Chin, and W. Yang. Reaching fault diagnosis agreement under a hybrid fault model. *IEEE Transactions on Computers*, vol. 49, no. 9, Sep. 2000.
24. R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *SOSP 2007*, pp. 45–58, ACM, 2007.
25. L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *Trans. on Programming Languages and Systems 6(2)*, 254–280, 1984.
26. L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News* 41(1): 63–73, 2010.
27. D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. *CSFW*, pp. 116–125, 1997.
28. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.
29. F. Preperata, G. Metze, and R. Chien. On the connection asssignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC–16(6): 848–854, December 1967.
30. K. Ramarao and J. Adams. On the diagnosis of Byzantine faults. *Proc. Symp. Reliable Distributed Systems*, pp. 144–153, 1988.
31. F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4): 299–319, 1990.
32. M. Serafini, A. Bondavalli, and N. Suri. Online diagnosis and recovery: on the choice and impact of tuning parameters. *IEEE Trans. Dependable Sec. Comput*, 4(4): 295–312, 2007.
33. K. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. *Proc. Symp. Fault-Tolerant Computing*, pp. 55–60, July 1987.
34. R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. *OPODIS*, 2012.
35. R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. *OSDI 2004*, pp. 91–104, USENIX Association, 2004.
36. M. Vukolic. Abstractions for asynchronous distributed computing with malicious players. PhD thesis. EPFL, Lausanne, Switzerland, 2008.
37. C. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Trans. Software Eng*, 23(11): 684–721, 1997.