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. We provide a number of optimizations and extensions and also take measures to make BChain more resilient to certain performance attacks. 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 failure, 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 [29, 35], 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 [6, 28, 1, 14] and chain-based protocols [21, 38]. 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 [21] and Zyzzyva [28] 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 [6] 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 [21], 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 [38] 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* [18]: 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 [6]. 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.

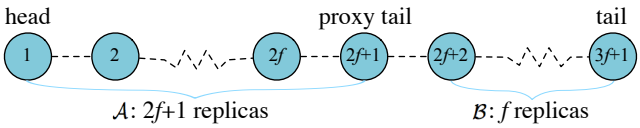


Fig. 1. BChain-3. Replicas are organized in a chain.

The chain order is maintained by every replica and can be changed 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.

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 $\mathcal{A}'$ 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.

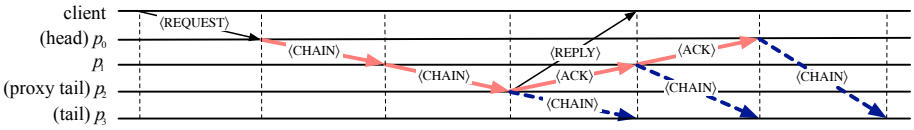


Fig. 2. BChain-3 common case communication pattern. (This and subsequent pictures are best viewed in color.) All the signatures can be replaced with MACs. All the $\langle$CHAIN$\rangle$ and $\langle$ACK$\rangle$ messages can be batched. The $\langle$CHAIN$\rangle$ messages with dotted, blue lines are 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, the number of MAC operations at the bottleneck server (i.e., the proxy tail) is $1 + \frac{3f+2}{b}$.

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 $p_j \in \mathcal{A}^{\not\nu}$ 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 $\langle$ACK$, v, ch, N, D(m), c\rangle_{p_j}$ message. In addition, it *forwards* to all replicas in $\mathcal{B}$ the corresponding $\langle$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 $\langle$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 $\langle$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 current round. 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, in a partially synchronous environment. The algorithm ensures that eventually all the faulty replicas be 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: $\quad starttimer(\Delta_{1,p_i})$
3: **upon** $\langle$Timeout, $\Delta_{1,p_i}\rangle$ {Accuser p_i}
4: $\quad$ send $\langle$SUSPECT$, \vec{p}_i, m, ch, v\rangle_{p_i}$ to $\overleftarrow{p}_i$ and p_h
5: **upon** $\langle$ACK$\rangle$ from $\vec{p}_i$
6: $\quad canceltimer(\Delta_{1,p_i})$
7: **upon** $\langle$SUSPECT$, p_y, m, ch, v\rangle$ from $\vec{p}_i$
8: $\quad$ forward $\langle$SUSPECT$, p_y, m, ch, v\rangle$ to $\overleftarrow{p}_i$
9: $\quad canceltimer(\Delta_{1,p_i})$

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 both the head and its predecessor a $\langle \text{SUSPECT}, \vec{p}_i, m, ch, v \rangle$ to signal the failure of its successor. Moreover, if p_i receives a $\langle \text{SUSPECT} \rangle$ 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 $\langle \text{SUSPECT} \rangle$ 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 $\langle \text{SUSPECT} \rangle$ message, just as an $\langle \text{ACK} \rangle$ message.

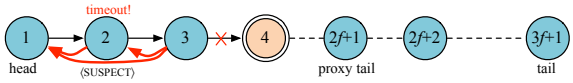
On receiving a $\langle \text{SUSPECT} \rangle$, the head starts re-chaining via a new $\langle \text{CHAIN} \rangle$ message. If the head receives multiple $\langle \text{SUSPECT} \rangle$ messages, only the one *closest* to the proxy tail is handled. Handling a $\langle \text{SUSPECT} \rangle$ message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

Algorithm 2 BChain-3 Re-chaining-I

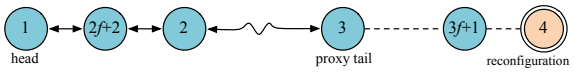
```
1: upon ⟨SUSPECT, p_y, m, ch, v⟩ from p_x      {Head p_h}
2:     if p_x ≠ p_h then                        {p_x is not the head}
3:         p_z is put to the 2nd position       {p_z = B[1]}
4:         p_x is put to the (2f + 1)th position
5:     p_y is put to the end
```

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 $\langle \text{SUSPECT} \rangle$ 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 the correctness for the general case is complicated, as shown in Appendix B. To help grasp the underlying idea, consider the following *simple* examples.



(a) p_3 generates a $\langle \text{SUSPECT} \rangle$ message to accuse p_4

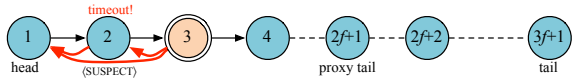


(b) p_4 is moved to the tail

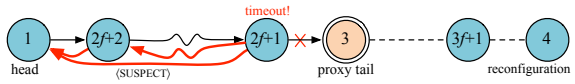
Fig. 3. Example (1). A faulty replica is denoted by a double circle. After the timer expires, replica p_3 issues a $\langle \text{SUSPECT} \rangle$ message to accuse p_4 (which is faulty). The head moves p_3 to the proxy tail position and the faulty replica p_4 to the end of the chain.

▷ 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.

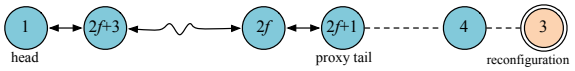
▷ Example (2): In Figure 4, p_3 is the only faulty replica. We consider the circumstance where p_3 sends the head a ⟨SUSPECT⟩ message to frame its successor p_4 even if p_4 follows the protocol. According to our re-chaining algorithm, replica p_4 will be moved to the tail, while p_3 becomes the new proxy 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 sends a ⟨SUSPECT⟩ message to accuse p_4. The head moves p_3 to the proxy tail and p_4 to the end of the chain. 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 end.

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 and §A.

Algorithm 3 BChain-3 Re-chaining-II

1: **upon** ⟨SUSPECT, p_y, m, ch, v⟩ 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

$\langle$ACK$\rangle$ and $\langle$CHAIN$\rangle$ 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 location in the chain order. Note that for p_h, we have that $l_h = 1$ and thus $\mathcal{F}(\Delta_1, 1) = \Delta_1$. 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. i.e., $\mathcal{F}(\Delta_1, l_i) = \frac{2f+1-l_i}{2f}\Delta_1$. As an example, in the case of $n = 4$ and $f = 1$, we set that $\Delta_{1,p_1} = \mathcal{F}(\Delta_1, 1) = \Delta_1$, $\Delta_{1,p_2} = \mathcal{F}(\Delta_1, 2) = \Delta_1/2$, 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 $f + 1$ $\langle$VIEWCHANGE$\rangle$ messages. The replica votes for view change and moves to a new view by sending all replicas a $\langle$VIEWCHANGE$\rangle$ 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 $\langle$CHAIN$\rangle$ message with signatures from $\mathcal{P}(p_i)$ and an $\langle$ACK$\rangle$ message with signatures from $\mathcal{S}(p_i)$. Otherwise, a proof of execution contains $f + 1$ $\langle$CHAIN$\rangle$ messages. Upon sending a $\langle$VIEWCHANGE$\rangle$ message, p_i stops receiving messages except $\langle$CHECKPOINT$\rangle$, $\langle$NEWVIEW$\rangle$, or other $\langle$VIEWCHANGE$\rangle$ messages.

When the new head collects $2f + 1$ $\langle$VIEWCHANGE$\rangle$ messages, it sends all replicas a $\langle$NEWVIEW$\rangle$ 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 $\langle$VIEWCHANGE$\rangle$ messages, and a set of $\langle$CHAIN$\rangle$ 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 $\langle$NEWVIEW$\rangle$, when a replica votes for a view change and waits for the $\langle$NEWVIEW$\rangle$. Algorithm 4 describes how to initialize, maintain, and adjust these timers.

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)
```

The view change timer Δ_2 at a replica is set up for the first request in the queue. A replica sends a $\langle$ViewChange$\rangle$ message to all replicas and votes for view change if Δ_2 expires or it receives $f + 1$ $\langle$ViewChange$\rangle$ messages. In either case, when a replica votes for view change, it cancels its timer Δ_2.

After a replica collects $2f + 1$ $\langle$ViewChange$\rangle$ messages (including its own), it starts a timer Δ_3 and waits for the $\langle$NewView$\rangle$ message. If the replica does not receive $\langle$NewView$\rangle$ message before Δ_3 expires, it starts a *new* $\langle$ViewChange$\rangle$ and updates Δ_3 with a new value $g_3(\Delta_3)$.

When a replica receives the $\langle$NewView$\rangle$ message, it sets Δ_1 and Δ_2 using $g_1(\Delta_1)$ and $g_2(\Delta_2)$, respectively. In practice, the functions $g_1(\cdot)$, $g_2(\cdot)$, and $g_3(\cdot)$ could simply double the current timeouts.

To avoid the circumstance that the timeouts for Δ_1 and Δ_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 [30] 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. This is possible because 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 $\mathcal{A}$. See §A for more details.

3.7 Optimizations and Extensions

We have developed several optimizations and extensions to BChain. Specifically, we developed means for replacing most signatures with MACs, and also means for combining MAC-based and signature-based BChain approaches. We also developed two variants of BChain, including a *pure* MAC-based protocol *without* reconfiguration when $n = 4$ and $f = 1$. However, due to lack of space, please refer to §D for details.

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 [28]. 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 [32]. 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 [6], Zyzzyva [28], and Aliph [21]. 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 [6] and the Bonnie++ benchmark [12]. 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.

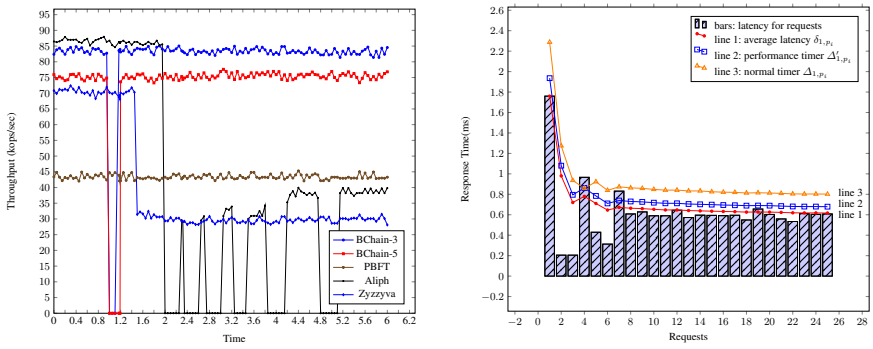
All experiments were carried out on DeterLab [5], utilizing a cluster of up to 65 identical machines equipped with a 2.13GHz Xeon processor and 4GB of RAM. They are connected through a 100Mbps switched LAN.

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*. [21] that (authenticated) chaining replication can increase throughput and reduce latency under high concurrency.

In addition to micro-benchmarks, we have also evaluated a BFT-NFS service implemented using PBFT [6], Zyzzyva [28], 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 this paper, our focus is on BChain's performance under failures, and thus we omit the detailed evaluation for gracious execution (§E.1) and the NFS use case (§E.2.)

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 [21, 40] generally switches between three protocols: Quorum, Chain, and a backup, e.g., PBFT. For our experiments, we adopt the same setting as Aliph paper [21], 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 rounds of 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 1001^{th} 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.

6 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.

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. I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. *INFOCOM 2004*, IEEE Computer and Communication Society, 2004.
4. 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.
5. T. Benzel. The science of cyber security experimentation: the DETER project. *ACSAC*, 2011.
6. M. Castro and B. Liskov. Practical Byzantine fault tolerance. *OSDI*, pp. 173–186, 1999.
7. M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst*, 20(4): 398–461, 2002.
8. T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4): 685–722, 1996.
9. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2): 225–267, March 1996.
10. M. Chiang, S. Wang, and L. Tseng. An early fault diagnosis agreement under hybrid fault model. *Expert Syst. Appl*, 36(3): 5039–5050, 2009.
11. 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.
12. R. Coker. www.coker.com.au/bonnie++.
13. 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.
14. 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.
15. A. Doudou and A. Schiper. Muteness failure detectors for consensus with Byzantine processes, *Brief announcement in PODC*, pp. 315, ACM press, 1998.
16. 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.
17. A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. *Ada-Europe 2002*, pp. 24–50, Springer, 2002.
18. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM* 35(2): 288–323, 1988.
19. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM* 32(2): 374–382, 1985.
20. S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *SOSP*, pp. 29–43, 2003.
21. R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. *EuroSys 2010*, pp. 363–376, ACM, 2010.

22. A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. *SOSP 2007*, pp. 175–188, ACM, 2007.
23. 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.
24. M. Hirt, U. Maurer, B. Przydatek. Efficient secure multi-party computation. *ASIACRYPT 2000*, pp. 143–161, 2000.
25. 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.
26. R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. Mohammadi, W. S-Preikschat, and K. Stengel. CheapBFT: resource-efficient byzantine fault tolerance. *EuroSys*, 2012.
27. S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC*, 18(4): 582–592, 2000.
28. R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *SOSP 2007*, pp. 45–58, ACM, 2007.
29. L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *Trans. on Programming Languages and Systems 6(2)*, 254–280, 1984.
30. L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News* 41(1): 63–73, 2010.
31. D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. *CSFW*, pp. 116–125, 1997.
32. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.
33. 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.
34. K. Ramarao and J. Adams. On the diagnosis of Byzantine faults. *Proc. Symp. Reliable Distributed Systems*, pp. 144–153, 1988.
35. F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4): 299–319, 1990.
36. 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.
37. 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.
38. R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. *OPODIS*, 2012.
39. R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. *OSDI 2004*, pp. 91–104, USENIX Association, 2004.
40. M. Vukolic. Abstractions for asynchronous distributed computing with malicious players. PhD thesis. EPFL, Lausanne, Switzerland, 2008.
41. C. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Trans. Software Eng*, 23(11): 684–721, 1997.

A BChain-3 Reconfiguration

Our reconfiguration technique works in concert with our re-chaining protocol. Recall that BChain-3 re-chaining protocol moves faulty replicas to set $\mathcal{B}$, while replicas that remain in $\mathcal{A}$ continues processing client requests. The reconfiguration procedure operates *out-of-band*, and thus does not disrupt request processing. Since it can be done out-of-band, it is not time sensitive, unless more failures occur.

An alternative to reconfiguration could be to recover suspected replicas. However, recovery is not possible for some types of failures, such as permanent failures. Recovery

may also take a long time, e.g. waiting for a machine to reboot, leaving the system vulnerable to further failures.

The key idea of our reconfiguration algorithm is to *replace* the replicas that were moved to set $\mathcal{B}$, with new replicas. A new replica first acquires a unique identifier. It also obtains a public-private key pair, and a shared symmetric key with each other replica in the system.

To initialize reconfiguration, a new replica in $\mathcal{B}$ with a unique identifier u sends a $\langle$RECONREQUEST$\rangle$ to all replicas in the system. Upon receiving the request, correct replicas send signed messages with their current $\langle$HISTORY$\rangle$ to replica u. Meanwhile, the replicas in $\mathcal{A}$ continue to execute the chaining protocol, where they also forward $\langle$CHAIN$\rangle$ messages to the newly joined replica u. In addition, replicas in $\mathcal{A}$ also retransmit missing $\langle$CHAIN$\rangle$ messages to the replicas in $\mathcal{B}$, including u, as the protocol requires. After collecting at least $f + 1$ matching authenticated $\langle$HISTORY$\rangle$ messages, u updates its state using the retrieved history and the $\langle$CHAIN$\rangle$ messages it has received. At this point, u can be promoted to $\mathcal{A}$ when deemed necessary.

It is clear that the reconfiguration algorithm can be performed concurrently with request processing, and as such is not time sensitive. This is because a newly joined replica is not immediately put into active use. Depending on the re-chaining algorithm, a new replica will not be used until f re-chainings have taken place (Algorithm 2), or $\lfloor f/2 \rfloor$ re-chainings with Algorithm 3.

Note that BChain-3 remains safe even if no reconfiguration procedure is used. In the case that there are only a small number of faulty replicas, e.g. $3t < f$, no regular reconfiguration is required to ensure liveness. Reconfiguration can be triggered periodically, as in other BFT protocols, or when frequent view changes and re-chainings occur.

Also note that, one might introduce a third set $\mathcal{C}$ that contains all of the "faulty" replicas, while $\mathcal{B}$ contains those that have been reconfigured and can be moved back to $\mathcal{A}$ on demand. The system has to wait if $\mathcal{B}$ is empty.

B Theorems and Proofs

B.1 BChain-3 Re-chaining-I

Theorem 1. *Let t denote the number of faulty replicas in the chain where $t \leq f$ and $n = 3f + 1$. 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 the head is correct and $3t > f$, the faulty replicas are moved to the end of chain with at most $3t$ re-chainings and at most $3t - f$ replica reconfigurations, assuming further that each individual replica can be reconfigured within f re-chainings.*

Proof: We assume all the timers are correctly set. We also assume that a single replica that is moved to set $\mathcal{B}$ can be correctly reconfigured within f re-chainings. Namely, it becomes correct before it is again moved from set $\mathcal{B}$ to set $\mathcal{A}$.

The proof is divided into four parts (Lemmas 2–5). Lemma 1 formally proves that if there is only one faulty replica in the chain, it will be moved to the end of the chain within at most two re-chainings. Lemma 2 captures an *essential* fact which is used on

multiple occasions. Lemma 3 shows the general result that all faulty replicas are eventually moved to set $\mathcal{B}$. Lemma 4 proves the maximum number of re-chainings required to remove t failures in the worst case. It also bounds the number of reconfigurations.

Faulty replicas can be divided into two types: first, a replica that does not behave according to the protocol so that the replica's predecessor fails to receive the valid $\langle$ACK$\rangle$ message on time, and second, a replica that sends a $\langle$SUSPECT$\rangle$ message maliciously, regardless of whether its successor is correct or not.

Lemma 1. *If there is only one faulty replica, it is moved to the end of the chain within two re-chainings. At most two replicas are moved to set $\mathcal{B}$.*

Proof of Lemma 1: First, if the only faulty replica, say, p_i, causes its (correct) predecessor $\overleftarrow{p}_i$ to fail to receive $\langle$ACK$\rangle$ message on time, it might trigger many $\langle$SUSPECT$\rangle$ messages sent from replicas ahead of p_i. However, since the head only deals with the $\langle$SUSPECT$\rangle$ message sent by the replica which is the closest to the proxy tail, the $\langle$SUSPECT$\rangle$ message sent from $\overleftarrow{p}_i$ will be handled. In this case, the faulty replica p_i is moved to the tail with only one re-chaining.

Second, we consider the case where the faulty replica p_i maliciously accuses its successor $\overrightarrow{p}_i$. According to our re-chaining algorithm, the faulty replica p_i (i.e., the accuser) becomes the proxy tail after one re-chaining. The proxy tail does not have a successor, so it is not capable of sending any $\langle$SUSPECT$\rangle$ messages to accuse any replicas. Therefore, p_i will be moved to the end of the chain if there is another re-chaining, in which case the $\overleftarrow{p}_i$ fails to receive the $\langle$ACK$\rangle$ message on time. In summary, the faulty replica p_i can be moved to the tail with at most two re-chainings.

In either case, a single faulty replica is moved to the end of the chain within at most two re-chainings, and furthermore, at most two replicas are moved to set $\mathcal{B}$. □

Lemma 2. *If a correct replica p_i sends a $\langle$SUSPECT$\rangle$ message to accuse its successor $\overrightarrow{p}_i$ while $\overrightarrow{p}_i$ does not send a $\langle$SUSPECT$\rangle$ message, $\overrightarrow{p}_i$ must be faulty.*

Proof of Lemma 2: Suppose $\overrightarrow{p}_i$ is correct. If the correct replica, p_i, sends a $\langle$CHAIN$\rangle$ message but fails to receive an $\langle$ACK$\rangle$ message on time, then p_i sends a $\langle$SUSPECT$\rangle$ message to accuse its successor. If $\overrightarrow{p}_i$ is correct but does not send a $\langle$SUSPECT$\rangle$ message then it must have received the corresponding $\langle$ACK$\rangle$ message on time. In this case, p_i can also receive the $\langle$ACK$\rangle$ message on time as well, since both of them are assumed to be correct. Therefore, p_i should not send a $\langle$SUSPECT$\rangle$ message in this case and $\overrightarrow{p}_i$ must be faulty. □

Lemma 3. *In the presence of t failures, assuming faulty replicas moved to set $\mathcal{B}$ are correctly reconfigured, one faulty replica is eventually moved to set $\mathcal{B}$. This results in $t-1$ faulty replicas in set $\mathcal{A}$. Therefore, all the faulty replicas are eventually moved to set $\mathcal{B}$.*

Proof of Lemma 3: We consider the suspect message which is the first one handled by the head. (Recall that the head only deals with one $\langle$SUSPECT$\rangle$ message that is sent from the replica that is closest to the proxy tail.) On the one hand, if the $\langle$SUSPECT$\rangle$ message

is generated by a correct replica, according to Lemma 2, a faulty replica is moved to set $\mathcal{B}$ with just this re-chaining, resulting in $t-1$ faulty replicas in set $\mathcal{A}$. On the other hand, if the $\langle$SUSPECT$\rangle$ message is generated by a faulty replica p_x, it will become the proxy tail after one re-chaining. Since the proxy tail is not capable of generating $\langle$SUSPECT$\rangle$ messages, the behavior of the p_x can be then either correct, or faulty, which will cause $\overleftarrow{p}_x$ to fail to receive $\langle$ACK$\rangle$ on time.

We describe four cases in additional detail: (1) $\overleftarrow{p}_x$ is faulty and generates a $\langle$SUSPECT$\rangle$ message to accuse p_x, and p_x is moved to the end of the chain with one re-chaining; (2) $\overleftarrow{p}_x$ is faulty and moved to the end of the chain in another re-chaining due to the $\langle$SUSPECT$\rangle$ message of the predecessor of $\overleftarrow{p}_x$; (3) $\overleftarrow{p}_x$ is correct and p_x behaves in a faulty manner. This means $\overleftarrow{p}_x$ failed to receive $\langle$ACK$\rangle$ message on time, so p_x is moved to the end of the chain due to the $\langle$SUSPECT$\rangle$ message from $\overleftarrow{p}_x$; (4) otherwise, after another re-chaining, p_x stays in set $\mathcal{A}$ and becomes the predecessor of the new proxy tail p_k. This indicates either of the following two cases: (4a) p_k is correct; (4b) p_k is faulty.

In any of the first three cases, a faulty replica is moved to the end of the chain, resulting in at most $t-1$ faulty replicas in the system.

We now discuss the last two cases and how the re-chaining algorithm eventually removes a faulty replica, resulting in $t-1$ faulty replicas in set $\mathcal{A}$.

For case (4a), a correct replica p_k becomes the proxy tail because it accuses its successor p_j in a previous re-chaining. According to Lemma 2, p_j must be faulty. Therefore, a faulty replica has been moved to the end of the chain.

In case (4b), p_x and p_k are both faulty and p_k is not capable of generating $\langle$SUSPECT$\rangle$ messages. Now the two faulty replicas p_x and p_k share the same "risk," in the sense that if either of the two replicas behaves in a faulty manner, one of them is moved to set $\mathcal{B}$ in another re-chaining. Indeed, if p_x generates a $\langle$SUSPECT$\rangle$ message to signal the failure of p_k, p_k is moved to the end of the chain, resulting in $t-1$ faulty replicas in set $\mathcal{A}$. If p_x or p_k causes $\overleftarrow{p}_x$ to fail to receive $\langle$ACK$\rangle$, p_x or p_k is moved to set $\mathcal{B}$. Therefore, in order to stay in set $\mathcal{A}$, both replicas must behave correctly. Inductively, if no more faulty replicas were to be removed afterwards, all the t faulty replicas would share the same risk. Since we assume that the faulty replicas moved to set $\mathcal{B}$ are correctly reconfigured, we do not need to worry about the cases where the faulty replicas again move back to set $\mathcal{A}$. With one more re-chaining, at least one faulty replica is moved to set $\mathcal{B}$, resulting in $t-1$ replicas in the chain.

We have proved that if there are t faulty replicas in the chain, the algorithm is able to move at least one faulty replica to the end of the chain, resulting in $t-1$ faulty replicas within $t+1$ re-chainings. Iteratively, all the faulty replicas are moved to set $\mathcal{B}$. □

Lemma 4. *All the faulty replicas are moved to set $\mathcal{B}$ within $3t$ re-chainings and at most $3t$ replicas have been moved to set $\mathcal{B}$. In the presence of t failures,* $\max(3t-f, 0)$ *reconfigurations are required.*

Proof of Lemma 4: In order to maximize the number of re-chainings, faulty replicas must accuse correct replicas without being moved to set $\mathcal{B}$. This is because otherwise at least one faulty replica is moved to set $\mathcal{B}$ in one re-chaining.

Initially, a faulty replica can accuse its successor while not being moved to set $\mathcal{B}$. After one re-chaining, this faulty replica becomes the proxy tail. It is able to accuse

another correct replica only if it moves forward later, in which case some other re-chaining must occur. Note that the reason that we put the first replica in set $\mathcal{B}$ just behind the head is therefore clear: to prevent correct replicas originally in set $\mathcal{B}$ from becoming the successors of faulty replicas after re-chainings. However, according to Lemma 2, such a correct replica accused by the proxy tail must have already accused a faulty replica so that it becomes the proxy tail. In other words, if each of the faulty replicas accuses more than one correct replica, the correct replica must have already accused a faulty replica. In summary, if there are t faulty replicas, they are able to accuse at most t correct replica before all of them become the proxy tail. Additionally, all t faulty replicas are able to accuse another $t-1$ correct replicas in total. Some of the faulty ones may accuse more than one correct replica but others will not get the chance before they are moved to set $\mathcal{B}$. Indeed, if the t faulty replicas had accused at least t correct replicas, the t correct replicas must have already accused t faulty replicas, resulting in no faulty replicas in the system. The maximum re-chainings for t failures is therefore $t+2(t-1)+2$, where the last two re-chainings is due to Lemma 1. Since set $\mathcal{B}$ contains f replicas, $3t-f$ replicas must be reconfigured to avoid the faulty replicas moved to set $\mathcal{B}$ going back to set $\mathcal{A}$. If $3t \leq f$ then no reconfigurations are required. Lemma 4 now follows. □

▮

B.2 BChain-3 Re-chaining-II

Theorem 2. *Let t denote the number of faulty replicas in the chain where $t \leq f$ and $n = 3f + 1$. 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 the head is correct and $2t > f$, assuming that each individual replica can be reconfigured within $\lfloor f/2 \rfloor$ re-chainings, then the faulty replicas are moved to the end of chain with at most $2t$ re-chainings and at most $2t - f$ replica reconfigurations.*

The proof for this theorem easily follows given that once a $\langle$SUSPECT$\rangle$ message is handled, there must be a faulty replica which has already moved to the tail of the chain. To justify the above fact, one simply needs to prove that for a $\langle$SUSPECT$\rangle$ message handled by the correct head, one of the accuser and the accused must each be faulty. The proof is relatively trivial and we therefore omit the details.

B.3 BChain-3 Safety

Theorem 3 (Safety). *If no more than f replicas are faulty, non-faulty replicas agree on a total order on client requests.*

Proof: The proof of the theorem is composed of two parts. First, we prove that if a request m commits at a correct replica p_i and a request m' commits at a correct replica p_j with the same sequence number, it holds that m equals m' within a view *and* across views. Then we prove that, for any two requests m and m' that commit with sequence number N and N' respectively and $N < N'$, the execution history $H_{i,N}$ is a prefix of $H_{i,N'}$ for at least one correct replica p_i. Together, they imply the safety of BChain-3.

► We first prove the first part *within a view* and begin by providing the following lemma.

Lemma 5. *If a request m commits at a correct replica p_i, at least $2f + 1$ replicas (including p_i) accept the $\langle$CHAIN$\rangle$ message with the same m and sequence number.*

Proof of Lemma 5: We consider two cases: $p_i \in \mathcal{A}$, and $p_i \in \mathcal{B}$.

$\triangleright$ $p_i \in \mathcal{A}$. We further consider two sub-cases: (1) p_i is among the first f replicas of the chain; (2) p_i is among the subsequent replicas (i.e., p_i is among the $(f+1)^{\text{th}}$ replica and the $(2f + 1)^{\text{th}}$ replica).

Case (1): It is easy to see that if p_i is among the first f replicas, p_i and all its preceding replicas accept a $\langle$CHAIN$\rangle$ message, since p_i receives a $\langle$CHAIN$\rangle$ message with valid signatures by $\mathcal{P}(p_i)$. It remains to be shown that all the subsequent replicas of p_i accept the $\langle$CHAIN$\rangle$ message.

To prove this, we must show that at least one correct replica p' among the last $f+1$ replicas in set $\mathcal{A}$ has sent an $\langle$ACK$\rangle$ message and all the replicas between p_i and p' have sent $\langle$ACK$\rangle$ messages. Note that if a correct replica sends an $\langle$ACK$\rangle$ message, it must have already accepted the corresponding $\langle$ACK$\rangle$ message and the $\langle$CHAIN$\rangle$ message. Meanwhile, since p' receives an $\langle$ACK$\rangle$ message with signatures from $\mathcal{S}(p_i)$, all the subsequent replicas of p' have already sent an $\langle$ACK$\rangle$ message. Combining all of this, all subsequent replicas of p_i in the chain send an $\langle$ACK$\rangle$ message and accept the $\langle$CHAIN$\rangle$ message with the same m and sequence number.

We now prove by induction that at least one correct replica p' among the last $f + 1$ replicas sends an $\langle$ACK$\rangle$ message with the same m and sequence number and all the replicas between p_i and p' send an $\langle$ACK$\rangle$ message. Clearly, p_i accepts an $\langle$ACK$\rangle$ message with $f+1$ signatures by $\mathcal{S}(p_i)$. Among $\mathcal{S}(p_i)$, at least one replica p'' is correct. If p'' is among the last $f + 1$ replicas, we are done here, since $\mathcal{S}(p_i)$ contains all the replicas between p_i and p''. Otherwise, inductively, we can eventually find at least one correct replica p' as required which is among the last $f + 1$ replicas. Meanwhile, each correct replica between p_i and p' ensures that all the replicas between p_i and p' have sent $\langle$ACK$\rangle$ messages.

Case (2): Likewise, it is easy to see that if p_i is among the last $f + 1$ replicas, p_i and all its subsequent replicas accept a $\langle$CHAIN$\rangle$ message since p_i receives an $\langle$ACK$\rangle$ message with valid signatures by $\mathcal{S}(p_i)$. We need to show all the preceding replicas of p_i accept the $\langle$CHAIN$\rangle$ message.

Similarly, we just need to prove that at least one correct replica p' among the first $f+1$ replicas has sent a $\langle$CHAIN$\rangle$ message and all the replicas between p_i and p' send an $\langle$CHAIN$\rangle$ message. We show this by induction. Note that p_i accepts $\langle$CHAIN$\rangle$ message with $f + 1$ signatures by $\mathcal{P}(p_i)$. Among $\mathcal{P}(p_i)$, at least one replica p'' is correct. If p'' is among the first $f + 1$ replicas, again we are done here. Otherwise, p'' receives $\langle$CHAIN$\rangle$ message with $f+1$ signatures from $\mathcal{P}(p'')$ and at least one replica in $\mathcal{P}(p'')$ is correct. Continually following the step, at least one correct replica p' as required can be found among the first $f + 1$ replicas. As each correct replica between p_i and p' sends a $\langle$CHAIN$\rangle$ message with $f + 1$ signatures, all the replicas between p_i and p' send a $\langle$CHAIN$\rangle$ message.

$\triangleright$ $p_i \in \mathcal{B}$. If p_i is in set $\mathcal{B}$, it receives $f + 1$ matching $\langle$CHAIN$\rangle$ messages from replicas in set $\mathcal{A}$. Among the $f+1$ replicas, at least one is correct. If the correct replica is among the first f replicas, following from the first case at least $2f + 1$ replicas accept and send

$\langle$CHAIN$\rangle$ message with m. If the correct replica is among the last $f+1$ replicas in set $\mathcal{A}$, following from the second case, at least $2f+1$ replicas then accept and send $\langle$CHAIN$\rangle$ message with m.

In either case ($p_i \in \mathcal{A}$ or $p_i \in \mathcal{B}$), if a request m commits at p_i, at least $2f+1$ replicas (including itself) accept and send $\langle$CHAIN$\rangle$ message for the same m. The lemma now follows. $\square$

We now show the proof and again address two cases—first where the two requests commit with the same re-chaining number, and second with different re-chaining numbers.

First, we need to prove that if m commits at p_i and m' commits at p_j with the same re-chaining number ch, m equals m'. Indeed, following Lemma 5, suppose m commits at p_i with ch, at least $2f+1$ replicas accept the $\langle$CHAIN$\rangle$ message with m, and at least $2f+1$ replicas accept the $\langle$CHAIN$\rangle$ message with m'. Since they accept the $\langle$CHAIN$\rangle$ message with the same chain order, at least one correct replica accepts and sends two conflicting $\langle$CHAIN$\rangle$ messages—one of them contains m while the other contains m'—which causes a contradiction. Thus, it must be case that m equals m'.

We now prove that if m commits at p_i and m' commits at p_j with different re-chaining numbers, the statement that m equals m' remains true. We assume that m commits at p_i with ch and m' commits at p_j with ch'. Without loss of generality, $ch' > ch$.

During the re-chainings, some replica(s) may be reconfigured. However, our re-chaining and reconfiguration algorithms ensure that once a replica is reconfigured it still has the same state as the non-faulty replicas by maintaining the history and (missing) messages from other replicas.

We now proceed in the proof via a sequence of *hybrids*. Any two consecutive hybrids differ from each other in their configurations. However, only one replica gets reconfigured in the latter hybrid. The initial hybrid is the just the configuration where m commits at a replica p_i with a re-chaining number ch, while the last hybrid is the one where m' commits at a replica p_j with a re-chaining number ch'.

Since m commits at p_i with ch, according to Lemma 5, at least $2f+1$ replicas accept and send an $\langle$CHAIN$\rangle$ message for m. The replica that has just been reconfigured must have the same state as the rest of the non-faulty replicas due to our reconfiguration algorithm. It is easy to prove via a *hybrid argument* that there exists two consecutive hybrids where at least $2f+1$ replicas accept an $\langle$CHAIN$\rangle$ message for m and N in the former hybrid, and at least $2f+1$ replicas accept an $\langle$CHAIN$\rangle$ message for m' and N in the latter hybrid.

Intersection of two Byzantine quorums would imply that at least one correct replica accepts two conflicting messages with the same sequence number, unless the replica that has been just reconfigured might be the correct one. Even in this case, it still causes a contradiction, as it must accept m with N according to our reconfiguration algorithm. However, if accepts the m' with N instead, this contradicts our reconfiguration assumption that reconfigured replica is correct after joining.

In either case, we have that if m commits at p_i and m' commits at p_j with the same sequence number during the same view, it holds that m equals m'.

Across views.

We now prove that if m commits at p_i with view number v and m' commits at p_j with view number v' where $v' > v$ and both with the same sequence number N, it still holds that m equals m'.

Since m commits at p_i in view v, according to Lemma 5, at least $2f + 1$ replicas accept m with N. Replica p_i includes a proof of execution for request m with N in the following view changes until it garbage collects the information about a request with sequence number N. Notice that reconfigured replicas still have the same state as the non-faulty replicas and the statement even with reconfigured replicas remains true.

Request m' commits in a later view v'. According to the protocol, the head in view v' sends a $\langle$CHAIN$\rangle$ message with m' and N after view change. This implies either of the following two cases in previous view(s). First, every view change message contains an empty entry for sequence number N. However, this cannot be true because p_i did not garbage collect its information about request m with sequence number N. The other case is that at least one view change message contains m' for sequence number N with a proof of execution. The proof of execution from a replica p in set $\mathcal{A}$ includes a $\langle$CHAIN$\rangle$ message with signatures by $\mathcal{P}(p)$ and an $\langle$ACK$\rangle$ message with signatures by $\mathcal{S}(p)$. The proof of execution from a replica in set $\mathcal{B}$ includes $f + 1$ $\langle$CHAIN$\rangle$ messages.

We now show that if at least one view change message in a view v_1 ($v \leq v_1 < v'$) contains m' and N with a proof of execution, at least $2f + 1$ replicas accept m' with N in view v_1. Assuming replica p sends a view change message with a proof of execution, there are three cases. First, if p is among the first f replicas, the proof of execution includes an $\langle$ACK$\rangle$ message with $f + 1$ signatures. In the chaining protocol, at least one correct replica signs and sends an $\langle$ACK$\rangle$ message. Therefore, request m' with sequence number N commits at a correct replica. According to Lemma 5, at least $2f + 1$ replicas accept m' with N. Second, if p is among the last $f + 1$ replicas in set $\mathcal{A}$, the proof of execution for m' with N includes a $\langle$CHAIN$\rangle$ message with $f + 1$ signatures and an $\langle$ACK$\rangle$ message with signatures by $\mathcal{S}(p)$. As proved in Lemma 5, at least $2f + 1$ replicas accept m' with N. Third, if p is in set $\mathcal{B}$, the proof of execution of m' includes $f + 1$ $\langle$CHAIN$\rangle$ messages, which are generated by at least one correct replica in the chaining protocol. Since a correct replica sends a $\langle$CHAIN$\rangle$ message to replicas in set $\mathcal{A}$ when the request is committed locally, according to Lemma 5, at least $2f + 1$ replicas accept m' with N.

Since a $\langle$NEWVIEW$\rangle$ message by the head includes all the view change messages, there exists a view v_2 ($v \leq v_2 \leq v_1 < v'$) in which p_i contains m and N with a proof of execution in its view change message while at least $2f + 1$ replicas accept m' in the chaining protocol. In other words, at least one correct replica accepts both m and m' in view v_2. This causes a contradiction. ∎

► Next we prove the second part of our theorem that for any two requests m and m' that commit with sequence number N and N' respectively, the execution history $H_{i,N}$ is a prefix of $H_{i,N'}$ for at least one correct replica p_i. Specifically, if m commits at any correct replica with sequence number N, according to Lemma 5, at least $2f + 1$ replicas accept m. Similarly, if m' commits at any correct replica with sequence number N', according to Lemma 5, at least $2f + 1$ replicas accept m'. Among the $2f + 1$ replicas, at least $f + 1$ replicas are correct. According to our protocol, correct replicas only accept $\langle$CHAIN$\rangle$ messages in sequence-number order. All the sequence numbers

between N and $N' - 1$ must have been assigned. On the other hand, at least $2f + 1$ replicas accept m with N. Since there are at least $2f + 1$ correct replicas, m and m' are assigned N and N' for at least one correct replica p_i. Therefore, $H_{i,N}$ is a prefix of $H_{i,N'}$. ∎

B.4 BChain-3 Liveness

Theorem 4 (Liveness). *If no more than f replicas are faulty, then if a non-faulty replica receives an request from a correct client, the request will eventually be executed by all non-faulty replicas. Clients eventually receive replies to their requests.*

Proof: BChain ensures liveness in a partially synchronous environment. We consider the system only after global stabilization time (i.e., only during periods of synchrony). Note that the bounds on communication delays and processing delays exist but are both probably unknown even to replicas. We now prove that BChain is live.

If the replicas in set $\mathcal{A}$ are all correct and timers are correctly maintained, then our chaining subprotocol (Section 3.3) guarantees that clients receive replies from the proxy tail.

We consider the case where the head is correct, timers are correctly maintained, and there might be faulty replicas. As long as the faulty replicas behave incorrectly, according to Theorem 1 or Theorem 2 (depending on which re-chaining algorithm one chooses), faulty replicas are moved to the tail of the chain (where, if needed, they are reconfigured), non-faulty replicas reach an agreement, and clients receive replies from proxy tail. If otherwise faulty replicas do not behave incorrectly then they still reach an agreement. (No further latency can be induced by intermittent or transient adversaries.) A minor corner case is that the proxy tail behaves correctly in reaching an agreement but fails to send a reply to some client, in which case the client will retransmit its request to all the replicas in set $\mathcal{A}$. Upon receiving $2f + 1$ consistent replies it accepts this reply. Alternatively, we could allow clients to suspect the proxy tail such that it can be removed in this case, just as in Zyzzyva and Shuttle.

It is possible that even in the case where the head is correct and timers are correctly set, view change can be triggered, since there might be too many re-chainings and some request is not completed in the current view. There are two additional cases that can inflict view changes: the head is faulty, and timers are not set correctly. As illustrated in Algorithm 4 in Section 3.5, the failure detection (re-chaining) timer Δ_1 and view change timer Δ_2 (for request processing) are adjusted in every view change when a replica receives the $\langle$NEWVIEW$\rangle$ message. They together can eventually move the system to some new view where the head is correct, timers are set correctly, and the re-chaining time is readily available. In the new view, replicas will reach an agreement and clients eventually receive their request replies.

To avoid frequent view changes, the timers are adjusted gradually. It is worth mentioning that in contrast to PBFT [6], we separate timer Δ_2 for request processing from the timer Δ_3 to wait for $\langle$NEWVIEW$\rangle$. Δ_3 will be adjusted to $g_3(\Delta_3)$, when a replica collects $2f + 1$ $\langle$VIEWCHANGE$\rangle$ messages but does not receive $\langle$NEWVIEW$\rangle$ message on time.

BChain follows the "amplification" step from $f + 1$ to $2f + 1$ $\langle$VIEWCHANGE$\rangle$. Namely, if a replica receives $f + 1$ valid $\langle$VIEWCHANGE$\rangle$ messages from other replicas with views greater than its current view, it also sends a $\langle$VIEWCHANGE$\rangle$ message for the smallest view. This prevents starting the next view change too late.

Note that faulty replicas (other than the head) cannot cause view changes, for the same reason as other quorum based BFT protocols. Also, although the faulty head can cause a view change, the head cannot be faulty for more than f consecutive views.

To prevent the timeouts Δ_1 and Δ_2 from increasing unbounded, we levy restrictions on the upper bounds for both. Slow replicas will be identified as faulty ones, which helps the system maintain its efficiency. ∎

B.5 BChain-5 Re-chaining

Theorem 5. *Let t denote the number of faulty replicas in the chain where $t \leq f$ and $n = 5f + 1$. If the head is correct, the faulty replicas can be moved to set $\mathcal{B}$ by the BChain-5 re-chaining algorithm after at most t re-chainings.*

The idea underlying the new re-chaining algorithm is as follows. A $\langle$SUSPECT$\rangle$ message (with p_x and p_y being the accuser and accused, respectively) is triggered, either because p_x fails to receive the $\langle$ACK$\rangle$ message from p_y (due to, e.g., a timing failure or a omission failure), or because a replica p_x maliciously accused p_y, regardless of the correctness of p_y. For either case, one re-chaining can move at least one faulty replica to set $\mathcal{B}$. Note that every re-chaining might introduce some faulty replicas originally in set $\mathcal{B}$ to set $\mathcal{A}$. Thus, it is not necessarily the case that every re-chaining can reduce the number of faulty replicas in set $\mathcal{A}$ by at least one. (Note it is possible that the number of faulty ones might even increase by one.) However, we claim that after at most f re-chainings, all the f failures can be moved to set $\mathcal{B}$. This is further due to the fact that faulty replicas that have been moved through re-chainings shall not have a chance to set $\mathcal{A}$ again, since the cardinality of set $\mathcal{B}$ is exactly $2f$. The theorem easily follows from the discussion above.

B.6 BChain-5 Safety and Liveness

Theorem 6. *BChain-5 achieves safety in the asynchronous environment and achieves liveness in the partially synchronous environment.*

The proofs for the safety and liveness properties of BChain-5 are simpler than those of BChain-3, as BChain-5 avoids the reconfiguration mechanism. The main lemma to be proven for its safety is that if a request m commits at a correct replica then at least $3f + 1$ replicas accept the $\langle$CHAIN$\rangle$ message with the same m and sequence number.

C BChain-3 for Persistent Failures

We also discuss a variant of BChain-3 that handles persistent failures [33], providing an efficient algorithm for systems that exhibit this type of failure. *Persistent failures* (or

permanent failures) are failures such that replicas constantly violate the specification of the predetermined protocols. Accordingly, failures other than persistent ones include *transient failures* and *intermittent failures*, which do not manifest themselves all the time and occur at irregular times.

We now discuss a re-chaining algorithm for BChain-3 that allows more efficient handling of an important, but general class of Byzantine failures, namely *persistent failures*. Replicas exhibiting persistent failures will *constantly* violate their specification in an *arbitrary* way. This includes *timing failures*, where correct results are obtained, but delivered too late, conventional *omission failures*, and *permanent failures* where a replica cannot recover to a correct state after having been faulty. Persistent failures also captures a large class of Byzantine adversaries such as "advanced, persistent threats" to subvert the system.

Algorithm 5 shows the re-chaining algorithm used with BChain-3, which is suitable for applications where there are only persistent adversaries. As in BChain-3, the head handles only one $\langle \text{SUSPECT} \rangle$ message in each re-chaining and only the $\langle \text{SUSPECT} \rangle$ message sent from the replica which is the closest to the current proxy tail.

Algorithm 5 PBChain-3 Re-chaining

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

Theorem 7. *PBChain-3 re-chaining algorithm incorporates the benefits of Algorithms 2 and 3. First, at least one faulty replica can be moved to set $\mathcal{B}$ with only two re-chainings. Second, the rate of the reconfiguration process required is the same as that of Algorithm 2. Furthermore, in the presence of f faulty replicas, the number of replicas to be reconfigured is f instead of $2f$.*

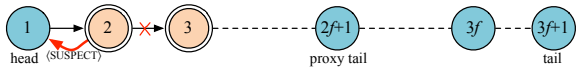
Proof: We assume that the correct head currently handles a $\langle \text{SUSPECT} \rangle$ message sent from p_x to accuse its successor p_y. This implies that p_x is the replica who sent a $\langle \text{SUSPECT} \rangle$ message and is the closest to the proxy tail.

We address four cases: (1) p_x and p_y are both correct; (2) p_x is correct and p_y is faulty; (3) p_x is faulty and p_y is correct; and (4) p_x and p_y are both faulty.

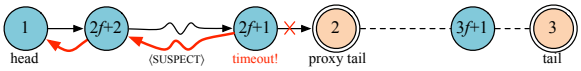
$\triangleright$ Case (1): Since we now consider the case in a synchronous environment, the situation where p_x and p_y are both correct and p_x generates a $\langle \text{SUSPECT} \rangle$ message to accuse its successor p_y is not possible. It is in fact easy to show that any other failures would not cause a $\langle \text{SUSPECT} \rangle$ message sent from p_x to be handled.

$\triangleright$ Case (2): In this case, replica p_x is correct and accuses its faulty successor p_y. Applying our re-chaining algorithm, p_y can be moved to the end of the chain with only one re-chaining. As an example in Figure 3, replica p_4 has a timing failure. This causes p_3 to send a $\langle \text{SUSPECT} \rangle$ message up the chain to accuse p_4. According to our re-chaining

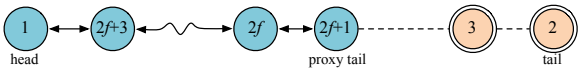
algorithm, p_3 is moved to the $(2f+1)^{\text{th}}$ position and becomes the proxy tail, and p_4 is moved to the end of the chain and becomes the tail.



(a) p_2 generates a ⟨SUSPECT⟩ message to accuse p_3



(b) p_{2f+2} generates a ⟨SUSPECT⟩ message to accuse p_2



(c) p_2 is moved to the tail

Fig. 6. Replica p_2 and replica p_3 are both faulty. Replica p_2 generates a ⟨SUSPECT⟩ message to accuse p_3, and p_2 becomes the proxy tail and p_3 is moved to the end of the chain. Replica p_{2f+2} becomes the predecessor of p_2, as captured in Figure 6(b). If p_2 later behaves incorrectly, p_{2f+2} generates a ⟨SUSPECT⟩ message to accuse p_2. Replica p_2 is moved to the end of the chain and p_{2f+2} becomes the proxy tail, as captured in Figure 6(c). Finally, faulty replicas p_2 and p_3 are moved to set $\mathcal{B}$.

▷ Case (3): We now consider the case where faulty replica p_x accuses its successor p_y which is actually correct. According to our re-chaining algorithm, p_y is moved to the tail and p_x becomes the proxy tail. Note now that p_x does not have a successor to accuse. Since p_x is a persistent failure, the only way that p_x can continue misbehaving is to cause its predecessor to fail to receive the corresponding ⟨ACK⟩ message on time, which would cause a ⟨SUSPECT⟩ message from its predecessor. Also recall that the head only handles the ⟨SUSPECT⟩ message sent from the replica the closest to the proxy tail even if there might be multiple ⟨SUSPECT⟩ messages at the same time. Therefore, p_x will be moved to the tail with another re-chaining. In this case, a faulty replica can be moved to the tail with only two re-chainings. Of course, its predecessor might be faulty as well and may not send any ⟨SUSPECT⟩ messages, in which case this predecessor will be removed with another re-chaining according to our algorithm. An example is illustrated in Figure 4, where 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.

$\triangleright$ Case (4): It is possible that a faulty replica p_x happens to accuse a faulty replica p_y, in which case each re-chaining can move one faulty replica. This can be justified as follows. When the head receives the $\langle$SUSPECT$\rangle$ message sent from p_x, p_y can be then moved to the end of the chain while p_x becomes the proxy tail in one re-chaining. Since p_x is a persistent failure, it will be moved to the tail with another re-chaining, just as in Case (3). Therefore, in this case, one faulty replica can be moved to the tail with only one re-chaining. We provide an example in Figure 6, where replicas p_2 and p_3 are both faulty and p_2 issues a $\langle$SUSPECT$\rangle$ message to accuse p_3.

D Optimizations and Extensions

Replacing *most* signatures with MACs. As shown in previous work [21, 6, 14, 28], it is possible to replace most signatures with MACs to reduce the computational overhead. This is also possible for BChain. In particular, it turns out that signatures for $\langle$REQUEST$\rangle$, $\langle$ACK$\rangle$, and $\langle$CHECKPOINT$\rangle$ can be replaced with a vector of MACs. However, in general, signatures on $\langle$CHAIN$\rangle$ messages cannot be replaced with MACs. Thus, we call this variant Most-MAC-BChain.

In our re-chaining protocol, a replica suspects its successor if it does not receive the $\langle$ACK$\rangle$ message in time. If a replica accepts and forwards a $\langle$CHAIN$\rangle$ message to its successor, it is trying to convince its successor that the message is correct. Meanwhile, the successor is able to verify if all its preceding replicas indeed honestly authenticated themselves. This requires transferability for verification, a property that signatures enjoys, while MACs do not.

We briefly describe an attack where a single replica can "frame" any honest replica—a scenario that our failure detection mechanism cannot handle, e.g., when $\langle$CHAIN$\rangle$ messages use MACs instead of signatures. Consider the following example, where there is only one faulty replica p_i, and $\vec{p}_i = p_j$ and $\vec{p}_j = p_k$. The faulty replica p_i simply generates a valid MAC for p_j and an invalid MAC for p_k. Replica p_j will accept it since the corresponding MAC is valid. It then adds its own MAC-based signature, and forwards the message to p_k. Since p_k receives the message with an invalid MAC produced by p_i, it aborts. Replica p_j will suspect p_k according to our algorithm, while p_i is the faulty one. Generalizing the result, a faulty replica can frame any honest replica without being suspected.

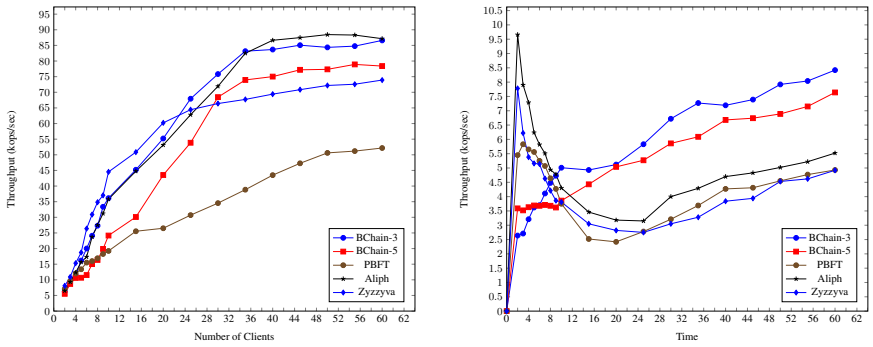
Replacing *all* signatures with MACs. We now discuss a variant of BChain, called All-MAC-BChain, in which all signatures are replaced with a vector of MACs, even for $\langle$CHAIN$\rangle$ messages in $\mathcal{A}$. As we discussed above however, these $\langle$CHAIN$\rangle$ messages must use signatures. However, if the head does not receive the $\langle$ACK$\rangle$ message on time, we can simply switch to Most-MAC-BChain to start the re-chaining protocol. Once the system regains liveness or faulty replicas have been reconfigured, we can switch back to All-MAC-BChain. This leads to the most efficient implementation of BChain. The performance in gracious executions will be that of All-MAC-BChain. In case of failures, the performance will be that of Most-MAC-BChain, with most signatures replaced with MACs and taking advantage of pipelining. The combined protocol is fundamentally different from the ones described in [21] such as Aliph, which does not perform well in

the presence of even a single faulty replica. Note that we evaluate our BChain protocols in Table 1 using this protocol variant.

BChain-3 with $n = 4$. We now consider BChain-3 configured with $(n = 4, f = 1)$, and show that this allows two interesting optimizations: BChain-3 without reconfiguration and All-MAC-BChain-3. This configuration of BChain is quite attractive, since its replication costs are reasonable for many applications, such as Google's file system [20].

BChain-3 without reconfiguration. We show that, with a slight refinement of the re-chaining algorithm, BChain-3 can also avoid reconfiguration: Upon receiving a $\langle$SUSPECT$\rangle$ from an accuser among the first two replicas in the chain, the head starts re-chaining. If the head is the accuser, then the accused is moved to the end of the chain. Otherwise, the accuser becomes the proxy tail, while the accused becomes the tail. It no longer needs to run the reconfiguration algorithm. In any future runs of BChain, if the head does not receive a correct $\langle$ACK$\rangle$ message, it simply *switches* the proxy tail (i.e., the third replica) and the tail (i.e., the last replica). A faulty replica can be identified with at most two re-chainings in case of synchrony. The view change algorithm is still the same as for BChain-3, which guarantees that eventually it achieves liveness with a bounded number of re-chainings in the partially synchronous environment.

All-MAC-BChain-3 via All MAC-based signatures. We now show that, contrary to the general case, BChain-3 with a $(n = 4, f = 1)$ configuration, can be implemented using only MACs. The reason we can do this is that the second replica in the chain can no longer frame its successor replica, while the behavior of the head is restricted by view changes. Thus, a total of twelve MACs are needed for communication between replicas and between replicas and clients. Recall also that a faulty replica can be identified with at most two re-chainings, and no reconfiguration is required.



(a) Throughput for the $0/0$ benchmark as the number of clients varies.

(b) Latency for the $0/0$ benchmark as the number of clients varies.

Fig. 7. Performance in gracious execution

Table 2. Throughput and latency improvement of BChain-3, comparing with PBFT and Zyzzyva, when f differs. Values with parenthesis in red represent negative improvement.

Number of Clients	Compared Protocol	$f = 1$		$f = 2$		$f = 3$	
		throughput	latency	throughput	latency	throughput	latency
20	PBFT [6]	48.61%	27.14%	36.95%	25.50%	1.69%	(1.36%)
20	Zyzzyva [28]	17.65%	5.44%	2.50%	5.79%	(1.93%)	(2.57%)
60	PBFT [6]	41.54%	33.72%	37.12%	30.50%	36.86%	26.03%
60	Zyzzyva [28]	22.59%	26.96%	15.67%	23.85%	14.04%	15.14%

E Evaluation

E.1 Gracious Execution Evaluation

Throughput. We discuss the throughput of BChain-3 and BChain-5 with different workloads *under contention*, where there are multiple clients issuing requests. We evaluate two configurations of BChain with $f = 1$: BChain-3 with $n = 4$ and BChain-5 with $n = 6$, both using All-MAC-BChain. As shown in Figure 7(a), all the other protocols outperform PBFT by a wide margin. With fewer than 20 clients, Zyzzyva achieves slightly higher throughput than the rest. But as the number of clients increases, Aliph-Chain, BChain-3, and BChain-5 gain an advantage over Zyzzyva. While BChain-3 and Aliph-Chain have comparable performance, they both outperform BChain-5. For both Aliph-Chain and BChain-3, peak throughput observed is 22% and 41% higher than that of Zyzzyva and PBFT, respectively. Note that the pipelining execution of our protocol explains why BChain-3 does not perform as well when the number of clients is small and why it scales increasingly better as the number grows larger.

Latency. We examine and compare the latency, as depicted in Figure 7(b). As expected, we can see that when the number of clients is smaller than 10, all the chain-based BFT protocols experience significantly higher latency than both Zyzzyva and PBFT. As the number of clients increases however, BChain achieves around 30% lower latency than Zyzzyva. Indeed, BChain-3, for instance, takes $4f$ message exchanges to complete a single request, which makes its latency higher in case of small number of clients. However, as the number of clients increases, the pipeline is leveraged to compensate for latency inflicted by the increased number of exchanges.

Scalability. We tested the performance of BChain-3 varying the maximum number of faulty replica, as summarized in Table 2, with both 20 and 60 clients. We observe that, the advantage of BChain-3 over other protocols decreases as f grows. When f grows to 3 and the number of clients is 20, BChain achieves lower performance than both PBFT and Zyzzyva. However, when the number of clients is large, BChain still achieves better performance. In contrast to many other BFT protocols with a constant number of one-way message exchanges in the critical path (c.f. Table 1), the number of exchanges in BChain-3 is proportional to f. In BChain-3, a client needs to wait for $2f + 2$ exchanges and the head needs to wait for $4f$ exchanges to commit a request. This intuitively explains why the performance benefits of BChain-3 becomes smaller as f increases. As

the pipeline is saturated with clients requests and large request batching is used, BChain can achieve better peak performance.

E.2 A BFT Network File System

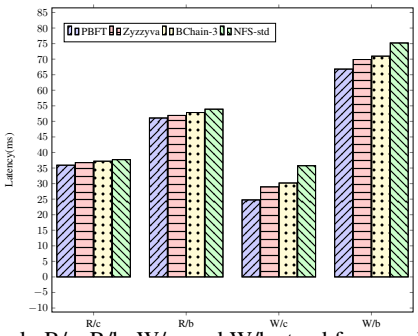


Fig. 8. Bonnie++ Benchmark. R/c, R/b, W/c, and W/b stand for per-character file reading, block file reading, per-character file writing, and block file writing, respectively.

This section describes our evaluation of a BFT-NFS service implemented using PBFT [6], Zyzzyva [28], 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.

We use the Bonnie++ benchmark [12] to compare our three implementations with NFS-std, an unreplicated NFS V3 implementation, using an I/O intensive workload. We first evaluate the performance on sequential input (including per-character and block file reading) and sequential output (including per-character and block file writing). Figure 8 shows that the performance of sequential input for all three implementations only degrades the performance by less than 5% w.r.t. NFS-std. However, for the write operations, PBFT, Zyzzyva, and BChain-3, respectively, achieves in average of 35%, 20%, and 15% lower processing speed than NFD-std.

In addition, we also 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 3, the latency achieved by BChain-3 is 1.10% lower than NFS-std, in contrast to BFS and Zyzzyva.

Finally, we evaluate the performance using the Bonnie++ benchmark when a failure occurs at time zero, as detailed in Figure 9. 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.

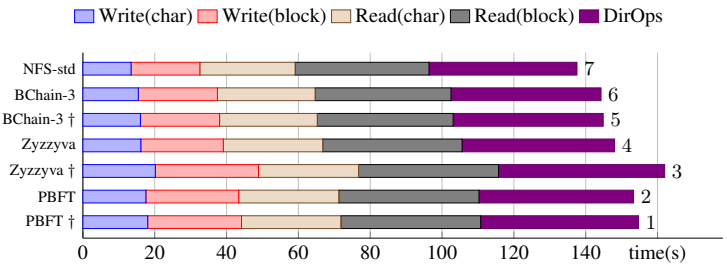


Fig. 9. NFS Evaluation with the Bonnie++ benchmark. The † symbol marks experiments with failure.

Table 3. 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 have shown in each case that every two re-chainings can move at least one faulty replica to the tail of the chain. With a similar argument, in the presence of at most f failures, as long as the first replica moved to set $\mathcal{B}$ can be reconfigured within the period of f re-chainings, there are no faulty replicas in set $\mathcal{A}$. ∎

F Further Related Work

Failure detectors were introduced by Chandra and Toueg [9] 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 [31, 15, 16, 4] 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 [16]. Therefore, for instance, consensus protocols from a muteness detector [15] have to handle Byzantine failures at the algorithmic level.

Fault diagnosis [33, 2, 34, 36, 41, 37, 25] 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 [22] 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 [24] 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.