Unstick Yourself: Recoverable Byzantine Fault Tolerant Services

Anonymized for Review

***Abstract*—Byzantine fault tolerant (BFT) state machine replication (SMR) protocols that can tolerate up to f failures in a configuration of $n = 3f + 1$ replicas cannot make any liveness guarantee once the number of faults surpasses f, even if some of these faults are benign crash faults. We argue that this weakness makes BFT protocols impractical in real-world deployments where faults accumulate over time. In this paper, we present a new reconfiguration mechanism, Phoenix, that builds on the pre-existing fault detection and reconfiguration mechanisms of BFT protocols to remove faulty replicas proactively using a trusted (but limited) configuration manager. We show that Phoenix can recover from f_B Byzantine faults and f_C crash faults, where $f_C \leq f_B$, if the system deploys $n = 3f_B + f_C + 1$ replicas. If a synchronous network connection is guaranteed between replicas and the configuration manager during reconfiguration, a synchronous variant of Phoenix needs only $n = 3f_B + 1$ replicas to achieve the same recoverability. To validate our approach, we implement Phoenix as an extension of the BFT-SMaRT library.**

1. Introduction

In recent years, blockchain technology has renewed interest in classical BFT, with much research done in how to adapt these protocols as the transaction ordering layer for permissioned blockchain systems [2], [5], [38]. BFT SMR, or simply BFT, protocols allow for implementations of highly available services that can tolerate Byzantine faults including bugs in the software or hardware, as well as malicious attacks. While crash failures are much more frequent in the real world, non-crash faults have been shown to cause issues [3], [37], which signifies the need for algorithms that can tolerate such faults. A permissioned blockchain system that employs a BFT protocol consists of a set of replicas, known a priori, each containing a copy of the application state. The replicas use Byzantine consensus to ensure enough correct replicas have the same state to prevent inconsistency.

Protocols like PBFT [8] require a configuration of $n = 3f + 1$ replicas [4], [12], up to f of which can fail simultaneously without affecting the liveness or safety guarantees. Since the inception of PBFT, we have seen an abundance of research into these types of protocols, including directions that explore optimizing the communication pattern [22], [26], [27] and resource requirements [20], [36], and relaxing network assumptions [24].

There are several limitations to systems implementing these protocols. First, these systems achieve fault tolerance by masking faults, and faulty behavior exhibited by one replica to another are simply ignored. The limited use of fault detection means that faulty replicas remain in the system indefinitely. Second, most BFT protocols are configured with a static set of replicas and do not explicitly support the addition or removal of replicas, meaning that even if faulty nodes are detected, no mechanism is available to remove them. If left unchecked, this would lead to an accumulation of faults that exceed the assumed upper bound (f) the system is capable of tolerating. Finally, if accumulated faults do eventually exceed f, Byzantine replicas completely control the availability of the system since there are now insufficient replicas to commit values or elect a new leader. Worse, an opportunistic Byzantine node can simply behave honestly until f faults have accumulated and only reveal itself when it assured control of the system.

Figure 1 highlights this scenario for $f = 1$ (thus $n = 4$). In this configuration, commiting a request or electing a new leader requires a quorum of $n - f = 3$ replicas to reach consensus. All replicas start at the same (orange) state, and a client issues a request to the current leader, $R0$. $R0$, which is Byzantine, excludes one of the replicas, $R3$, from the consensus round that processes the request and commits the request using only the remaining replicas. When all three have executed the request, replica $R1$ crashes, but replies from the two remaining replicas are sufficient to assure the client the request was successful. Once $R0$ suspects that $R1$ has crashed, however, $R0$ can choose to halt the system by not processing any additional requests. A new leader cannot be elected because only two other replicas are functional. $R0$ can block any quorum attempting to make progress, so the system is stuck.

Typically, the fault allowance f in traditional BFT protocols accounts for both crash and Byzantine replicas. In this work, we differentiate between crash faults and arbitrary Byzantine faults, and bound these faults by f_C and f_B, respectively. To avoid confusion we will refer to the fault bounds of prior BFT protocols as f_B instead of f since Byzantine faults subsume crash faults in these protocols. Furthermore, we distinguish *commit quorums*, quorums that commit new entries to the log, from *view change quorums*, quorums that elect a new leader. To recover the system, we must first be able to remove faulty replicas and replace them with new ones, then preserve the most recent state from the correct replicas across reconfiguration. In order to be practical for long term deployments, modern SMR implementations include a reconfiguration mechanism that can be invoked by an administrator to add or replace replicas. However, these reconfiguration protocols require the

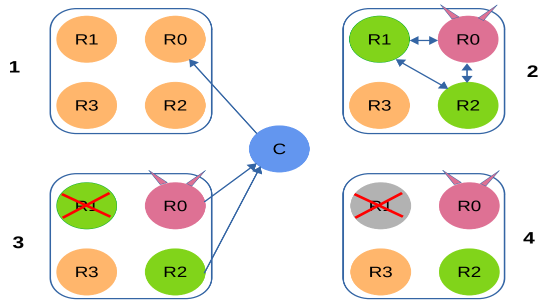

Figure 1: Client C issues a request to be executed, Byzantine leader $R0$ excludes $R3$ from the consensus instance to replicate the request, $R1$ crashes, and $R0$ stops sending messages and halts the system.

administrator to have prior knowledge of which replicas in the current configuration are faulty, which is not always possible.

Even if we assume a priori knowledge of faulty replicas, reconfiguration still might not be possible. In order to safely reconfigure a set of replicas without risking safety violations, the remaining replicas must reach consensus on when (and thus in what state) the reconfiguration takes place. For $n = 3f_B + 1$, reaching consensus requires a quorum of size $n - f_B$, which ensures that at least $f_B + 1$ members of this quorum participated in the previous quorum. Since this number exceeds f_B, at least one correct node in the reconfiguration quorum is fully up to date and no log entries will be truncated. The log entries from a correct node are authenticated by other nodes using *commit certificates* containing $n - f_B$ digital signatures from previous quorums. However, if there are $f_B + f_C$ faulty nodes, fewer than $n - f_B$ correct nodes remain, so reconfiguration will fail.

In this paper, we present Phoenix, a reconfiguration protocol that complements BFT protocols, allowing them to replace faulty replicas from the active replica set. Phoenix uses built-in fault detection mechanisms of BFT protocols to facilitate a voting scheme where replicas send votes to a configuration manager indicating which replicas they consider to be faulty. In order to tolerate an additional f_C crash faults over f_B Byzantine faults (with $f_C \leq f_B$), Phoenix requires $n = 3f_B + f_C + 1$ total replicas, commit quorums of size $n - f_B$, and view change quorums of size $n - f_B - f_C$. The configuration manager can then safely reconfigure the system as long as $n - f_B - f_C$ correct replicas remain (enough for a view change quorum). Note that tolerating the same number of faults in a traditional BFT protocol would require $n = 3(f_B + f_C) + 1$ replicas and quorum sizes of $2(f_B + f_C) + 1$.

Under stronger network assumptions, we can reduce the required number of replicas as well as the quorum size. If synchronous communication during reconfiguration is guaranteed between the replicas and the configuration manager, our synchronous variant, Sync Phoenix, tolerates f_B Byzantine and f_C crash faults ($f_C \leq f_B$) with only $n = 3f_B + 1$ replicas.

Finally, we present an evaluation of two design choices regarding the *reply quorum* size, or the number of replies a client must wait for before considering a request committed. Clients of protocols such as PBFT wait for a minimum of $f_B + 1$ replies to so that at least one correct node has received $n - f_B$ commit certificates (which is required before a replica responds to the client) so that the committed request persists even in the event of a view change. In Phoenix, we must guarantee that clients do not consider a request committed until it is ensured to be preserved across reconfigurations. We evaluate two alternative design choices that meet this requirement. One design preserves the reply quorum size of $f_B + 1$, but requires replicas to digitally sign their *write* messages and create certificates analagous to commit certificates, but for the *write* phase (the second phase of the commit protocol). These certificates allow requests to be replayed in the event all correct nodes in a $f_B + 1$ reply quorum crash after responding to the client. The other design simply requires the client to wait for $n - f_B$ replies. Since $f_C \leq f_B$, this means that at least one correct replica will survive across configurations. Our results show that under reasonable latency assumptions, the benefits of the smaller reply quorum are significantly outweighed by the overhead of signing and verifying *write* messages.

2. Background and System Model

Fault Model. We consider a distributed system consisting of a set of $n = 3f_B + 1$ replicas and a separate set of client machines, all connected by an asynchronous network. Both clients and replicas can exhibit Byzantine faults, and at point in time, there can be at most $f_B = \left\lfloor \frac{n-1}{3} \right\rfloor$ Byzantine-faulty replicas and an additional $f_C \leq f_B$ crash-faulty replicas.

The Configuration Manager. The configuration manager (CM) is an entity that helps the system recover by replacing faulty replicas, and it has connections with all the replicas in the current replica set as well as an additional set of spare replicas that are used as replacements. This CM serves three purposes. First, it can be used to modify the degree of replication by adding or removing replicas. This is useful for when replicas need to be taken offline for maintenance or added to increase f_B. Second, it can serve as an arbiter for faults in the system, taking in votes from replicas against other replicas they think are faulty; when there are enough votes against a replica, it can then remove the replica. We will describe in a later section how to leverage this functionality to serve as an imperfect fault detector to remove faulty replicas. By ”imperfect”, we mean that replicas can use the voting protocol to conclude that another replica is faulty, but occasionally this conclusion could be false if the network partitions the correct replicas for prolonged periods of time. Lastly, in the synchronized version of our protocol described in Section 6, the CM can help the system recover recover even when there are only $f_B + 1$ correct replicas remaining. Messages that are sent from the CM are signed, so replicas can trust these messages if the signature is valid.

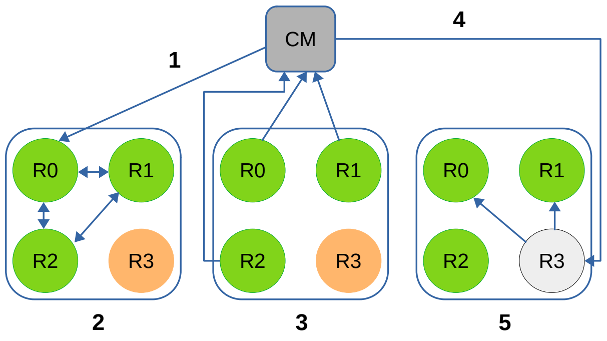

Figure 2: Reconfiguration in BFT-SMaRt. The configuration manager issues a reconfiguration request to replace $R3$.

BFT typically consists of two subprotocols: a Byzantine commit protocol for normal-case execution, and a view-change protocol to replace the leader when requests fail to move through the commit protocol. When a leader receives requests from clients, it uses *propose* messages to order these requests in a sequence, and broadcasts these messages to replicas. All replicas, including the leader, send *write* messages that acknowledge the validity of the *propose* to each other. A replica waits for a quorum of $n - f$ of these *write* messages to commit locally; a local commit at a replica ensures a total order of requests that it has seen. Replicas then exchange *accept* messages. We call a quorum of $n - f$ of these *accept* messages a *Commit Quorum* (CQ), which lets a replica know that the request has been committed at enough correct replicas for it to execute and send a *reply* message to the client. Clients wait to receive $f + 1$ of these reply messages for a request, which we call a *Reply Quorum* (RQ), to be sure that at least one is from a correct replica before it considers the request as finished and move on to the next request.

3. Reconfiguring a Byzantine State Machine

The standard approach to reconfigure the current active replica set is to place the configuration change inside a request, and have it executed by the replicas [4], [33]. Figure 2 shows the reconfiguration process in BFT-SMaRt. An administrator node (CM), constructs the new configuration and issues it as a client request (1) that will be included in a *propose* message, after which it will go through a consensus round (2). Once the replicas have executed this request, they will reply to the CM (3), who will wait to receive $f_B + 1$ replies. If a new replica was added, the CM will notify the new replica with a message containing the new view (4). A new replica joining will first initiate a state transfer request from the other replicas (5) to catch up to the latest consensus decision.

3.1. Unsafe Reconfiguration

By having replicas reach consensus on the CM's reconfiguration request, we ensure that eventually all correct replicas will transition to the new view. However, we can run into situations where safety is violated if the CM is not careful when replacing replicas. Consider the example in Figure 2, but the replica that is being replaced is $R2$ instead of $R3$. That is, $R2$ is participating in a consensus decision that leads to the removal of itself from the active replica set. This situation might seem strange, but it could be that $R2$ is the only one partitioned away from the CM, so the CM sees it as faulty. Suppose that $R0$ is Byzantine and it is the leader. $R0$ could truncate its log to match that of $R3$, and $R4$ would then possess the old (orange) state at the end of the state transfer protocol if it receives state from only $R0$ and $R3$. Then, $R0$ could propose a different value for the most recent consensus instance and, together with $R3$ and $R4$, they would be make up a sufficiently large quorum to decide on that value. In order to avoid this situation and remove $R2$ safely, we must prevent reconfigurations in which a correct replica participates in a decision that leads to its removal, or allow slow replica $R3$ to catch up before removing $R2$.

4. Phoenix

Notice that when the CM constructs a new view to replace a replica, it needs to know which replica to remove (i.e., which replica is faulty) before sending the reconfiguration request. Knowing which replicas to remove is very difficult in a system with Byzantine faults, because a Byzantine replica can behave correctly to the CM but misbehaves when interacting with other replicas. One example of this type of Byzantine behavior are mute replicas [14], where a mute process simply stops sending consensus messages to one or more processes but could still respond to heartbeat messages. To detect such failures, a CM must be able to gather information about suspected faulty behavior from the replicas themselves.

Our goal in building Phoenix is to build a bridge between subjective fault detection and dynamic reconfiguration, both of which already exists individually in state-of-the-art BFT systems like BFT-SMaRt. By themselves, these mechanisms are limited in their ability to remove subjectively faulty replcas since no replica can provide irrefutable proof against a faulty replica. In many cases, meaningful reconfiguration choices cannot be made without witnessing replica-to-replica communication. However, if replicas can use their local fault detection mechanism to inform a configuration manager of other replicas they suspect of being faulty, the configuration manager can then rely on such information to make informed reconfiguration decisions.

4.1. Voting: Subjective Fault Detection

Using Phoenix, we would like to remove replicas that consistently exhibit subjectively faulty behavior. We define a subjective fault to be one that cannot be proven to a third party; examples of these include delayed or missing consensus messages and messages with an invalid MAC. Over time, a replica that experiences faulty behavior from another replica might suspect it to be faulty, after which it will ask the CM to remove the suspected replica. To do this, a replica will construct a message $\langle$VOTE, v, n, j$\rangle_{\sigma_i}$, where v indicates the current view number, n is the sequence number of the last finished consensus instance, and j is the replica that is being voted against. After constructing the message, a replica signs it and then sends the signed message to the CM as well as the other replicas.

A replica k that receives a *vote* message from another replica will first validate the message before checking to see whether the replica being voted against j has received $f_B + 1$ votes. If replica k receives $f_B + 1$ *vote* messages against replica j, k will prepare its own *vote* message against j, and send it to the other replicas and the CM. This step ensures that if $f_B + 1$ replicas have committed to voting out j, then all correct replicas will eventually commit to voting out j. View changes rely on a similar approach to allow all correct replicas to send their own view change messages once they believe that a correct replica has committed to a view change (after receiving $f_B + 1$ *view-change* messages) [8], [22]. When the CM receives a *vote* message, it will also perform the same verification on the message and tally the vote against j. The CM will wait to receive $n - f_B - f_C$ votes against j, then construct a new configuration with this replica replaced with a backup and send it to the replicas in a client request.

To handle the case of unsafe reconfiguration in Section 3.1, a correct replica will not send a vote against itself. That is, a replica j will not send a *vote* message, even if it has received $f_B + 1$ votes against itself. There are two possible scenarios depending on whether the Byzantine replica votes against j. If the Byzantine replica participates in the vote, then eventually the CM will receive enough votes to remove j. However, a slow replica that receives *vote* messages with a higher sequence number than the one it knows about will first request a state transfer before sending its own vote. Thereby, any slow replicas will be caught up before the CM has enough votes, and removing j will be safe. If, on the other hand, the Byzantine replica does not send its vote and stop sending protocol messages altogether, then the CM will not receive enough votes with j's nonparticipation, causing j to remain in the replica set and the system will be stuck until the partition that caused j to look faulty heals.

4.2. Reaching Agreement in Reconfiguration

The difficulty in reconfiguration is to make sure enough correct replicas commit to the new configuration. If we do not consider the existence of additional f_C crashed replicas (i.e., $f_C = 0$), reconfiguration requests from the CM would eventually finish even if the Byzantine replicas do not participate, since the remaining replicas form a sufficiently large CQ. Now we must consider the case of $f_C > 0$ and how these additional crashes affect the decision of the reconfiguration request. If we were to set CQ at $n - f_B - f_C$ $(2f_B + 1)$, it is easy to see that safety can be violated. Two CQs can intersect at only a Byzantine leader, which can equivocate, causing two groups of correct replicas to decide on different values for a single log entry. If, instead, we have CQ be $n - f_B$ $(2f_B + f_C + 1)$, then the Byzantine replicas can withhold messages when there are crashed replicas, and the reconfiguration request can never be completed. Thus, since we cannot achieve both safety and liveness, replicas have to be able to reconfigure through a different path other than the commit path. This separate path is the view change protocol.

In PBFT, the view change protocol is used to rotate leaders, preventing a Byzantine leader from halting the system indefinitely. It also serves another purpose in that it allows correct replicas to converge on the latest consensus instance before entering the new view. This is done by having replicas exchange their logs and verify them individually before they transition to a new view. Phoenix utilizes the view change path to have replicas synchronize their logs and perform reconfiguration.

To initiate a reconfiguration, the CM sends a *reconfig* message to the replicas, causing them to issue a *stop* message and halting processing of further requests. A replica receiving at least $2f_B + 1$ *stop* messages will then send a *stop-data* message to the leader of the next view, and include within this message the signed request from the CM containing the new configuration. The new leader will wait for $2f_B + 1$ valid *stop-data* messages before issuing a *sync* message to the replicas with the information from the *stop-data* messages. We call the set of $2f_B + 1$ of these *stop-data* messages a *View change Quorum* (VQ). Each replica will verify the *sync* message, and check for the request from the CM. If the *sync* message contains the CM's request, replicas will process the configuration change before replying to the CM. Replicas will preserve this *sync* message so that they can provide it to any replicas that were not part of the reconfiguration change.

Because of the smaller view change quorum, it is possible that a view change succeeds with just the slow and Byzantine replicas. However, without the up-to-date correct replicas, there won't be a sufficiently large enough quorum to commit new requests and the slow replicas will eventually be caught up once they receive a state transfer from the up-to-date replicas.

5. Tolerating f_B and f_C with $n = 3f_B + 1$

Asynchronous BFT protocols cannot tolerate more than $\lfloor \frac{n-1}{3} \rfloor$ simultaneous faults out of $n = 3f_B + 1$ replicas [24]. Phoenix as described in Section 4 would fail since $f_C > 0$ crashed replicas allow the Byzantine replicas to prevent any reconfiguration attempts by simply withholding *view change* messages. Fault detection in Phoenix relies on at least $f_B + 1$ *vote* messages to convince a correct replica to also send their *vote*, so a CM could wait for only $f_B + 1$ *vote* messages instead of $n - f_B - f_C$ to initiate a reconfiguration. However, replicas can't just adopt the new configuration without the view change since that step is necessary for replicas to synchronize their states.

Without performing a view change or committing the CM's request, reconfiguration would be possible only if the $f_B + 1$ vote messages that the CM receives contains at least one message from a replica with the latest state and the replicas going into the new configuration can synchronize their state with this replica. But this smaller set of *vote* messages that the CM waits for could be from the slow replicas and the Byzantine replicas who truncated their logs to match that of the slow replicas. If the CM instead waits for $2f_B+1$ *vote* messages, it can be left waiting forever if the Byzantine replicas do not send in their vote. For this reason, if there are f_B Byzantine faults and f_C crash faults, reconfiguration is safe with $n = 3f_B+1$ replicas only under the assumption that system has synchrony during reconfiguration, and that the *vote* message from the correct replica with the latest state makes it to the CM and subsequently all the correct replicas in the new configuration.

5.1. Safety in reconfiguration

Phoenix must preserve the following properties.

Property 1. A reconfiguration cannot be triggered by the faulty replicas only, and must be requested by at least one correct replica.

Property 2. All correct replicas in the new configuration must start with the same state, and this state reflects the latest consensus decision from the previous configuration.

Property 3. Requests acknowledged by $n-f_B$ replicas must be preserved across reconfigurations.

Reconfiguration, like view changes, is costly and results in a temporary halt to the service as replicas stop processing messages in order to reconfigure. Property 1 prevents Byzantine replicas from being able to force the CM to initiate a reconfiguration, even if they collude. This is done by having the CM waits for at least $n-f_B-f_C$ *vote* messages before it replaces a replica.

Property 2 and a weaker variation of property 3 (f_B+1 replies instead of $n-f_B$ replies) must be guaranteed by a standard view change algorithm [16], therefore they also must hold for reconfigurations. To guarantee Property 2, the replica(s) with the latest state in the old configuration must be able to relay this state to the CM. This is done by having replicas send in their decision logs, with the longest log, one that contains no gaps and a valid proof ($n-f_B$ signed *accept* messages) for each decision, being the one selected by the CM. The CM can then relay this state to all the replicas in the new configuration, requiring that correct replicas install this state so that they can all start at the same consensus instance.

Property 3 allows clients to have a consistent view of the replica state. Clients must know when to consider their request as successful in order to move on to subsequent requests, and replies from correct replicas to the client must be consistent across reconfigurations. Previous designs of BFT allows a client to consider a request as successful after f_B+1 replies from different replicas, but this is not sufficient when there are f_B and f_C simultaneous faults. These properties are maintined even if a non-faulty replica is removed.

5.2. The Role and Cost of Digital Signatures

We have shown that in order to survive f_B and $f_C > 0$ faults, at least one correct replica with the latest state must be able to relay their decision log to the CM. It is also mentioned that this log must be valid in that each entry in the log contains a proof that allows verifications by the CM and other replicas. To allow consensus decisions to be verified in a posterior view, we can have replicas sign their consensus messages [34]. There is, however, a tradeoff between which consensus message to sign and the number of replies a client waits for before moving on. In BFT-SMaRt, the *accept* messages are signed, which allows replicas to build certificates that can be verified. With this approach of signing messages in the last phase of consensus, clients have to wait for $n-f_B$ replies before they can move on. PBFT-PK (PBFT with signatures [6]) relies on signatures on the *write* messages to prove consensus decisions in a view change. If signatures are used on these consensus messages, clients can wait for a smaller number of replies ($n-2f_B$) between requests. Although the reply quorum is smaller, there is at least one correct replica that has gathered $n-f_B$ *accept* messages before replying, meaning that at least $n-f_B$ replicas got enough signed *write* messages for a certificate. Based on our evaluation in section 7.1, adding signatures to the *write* phase would have a bigger impact on performance than waiting for more replies, so in Phoenix we followed the latter approach.

6. Sync Phoenix

We now present a version of Phoenix, called Sync Phoenix, that allows a system to reconfigure the replica set when there are up to f_B Byzantine faults and f_C crash faults with just $3f_B+1$ replicas. Sync Phoenix relies on synchrony to provide a stronger recoverability guarantee during reconfiguration, that is, a BFT system using Sync Phoenix can recover if at least f_B+1 correct replicas are still alive. Technically, we don't require that communication is always synchronous, only during reconfiguration. Therefore, Sync Phoenix can also be used in weaker synchrony models, such as one where there is always a subset of synchronous replicas [1], [24] or a model where this synchronous subset can change with each round [9], provided the CM is part of the synchronous subset.

6.1. Changes to Phoenix

Voting. The first change is to require that replicas also include their decision log in their *vote* message when sending it to the replicas and the CM. A replica's decision log contains all the decided consensus instances, each with a proof [34]. The *vote* message from a replica will now be $\langle$VOTE, v, L_i, j$\rangle_{\sigma_i}$, where L_i is the latest decision log of the replica i.

When the CM or a replica receives a *vote* message, it checks that the decision log L_i has no gaps and that each decided value in the log has a valid proof. The checking

	n	CQ	VQ	RQ	Reconfiguration
BFT-SMaRt [4]	$3f_B + 1$	$n - f_B$	$n - f_B$	$f_B + 1$	$n - f_B$
Phoenix	$3f_B + f_C + 1$	$n - f_B$	$n - f_B - f_C$	$n - f_B$	$n - f_B - f_C$
Sync Phoenix	$3f_B + 1$	$n - f_B$	$n - f_B$	$n - f_B$	$f_B + 1$

TABLE 1: The differences in sizes of the Commit Quorum (CQ), View-change Quorum (VQ), Reply Quorum (RQ), and Reconfiguration Quorum (RQ) for BFT-SMaRt, Phoenix, and Sync Phoenix.

of the decision log for gaps and valid consensus proofs is a standard check that is a part of the BFT-SMaRt view change algorithm. Once the contents have been validated, the CM will increment the vote count for the replica being voted against in the message, then records the sequence number n along with the proof. When the vote count against a replica j surpasses f_B+1 (i.e., at least one honest replica has voted against j), the CM then finalizes the voting round.

Reconfiguration. To finalize a voting round, the CM sets a timer and sends a $\langle$VOTE-REQUEST$\rangle_{\sigma_{CM}}$ to all replicas, requesting they send their vote. Upon receiving this message, replicas will stop processing requests and broadcast their vote. The request from the CM not only allows replicas that have not voted to have their vote counted, but also allows replicas that have voted to update their log. This accounts for replicas that have voted but a reconfiguration happens after it has processed further consensus instances; however, a replica updating its vote will not have the vote counted twice.

The CM waits until the timer has expired before processing the *vote* messages. After processing all the *vote* messages, the CM determines the longest log it has seen where n is the highest sequence number seen and that there is a valid proof for consensus instance n along with the corresponding view v. The CM then sends a $\langle$RECONFIGURE, v+1, S, $L_k\rangle_{\sigma_{CM}}$ to all the replicas in S. Here, L_k is the log from replica k that was chosen as the longest valid log. When a replica receives the *reconfigure* message from the CM, it will adopt the log L_k, and sends an acknowledgement to the CM to signal that it has successfully transitioned into the new configuration. At this point, all the replicas in the new configuration will initiate a view change to elect the leader for view $v + 1$, after which the replicas can resume processing requests.

6.2. Unsticking the System

Let us revisit the example shown in Figure 1. The leader $R0$ stops sending messages after suspecting (correctly) that $R1$ has crashed, and the only two replicas with the most up-to-date state are the leader $R0$ and replica $R2$. If clients issue subsequent requests, they will not be able to go through the Byzantine commit protocol and get executed by the replicas because the leader will not include them in a *propose* message. Furthermore, the request timers and the view change algorithm that would normally replace the faulty leader would not succeed because $R0$ would withhold the view change messages.

In this case, the correct replicas at the very least can suspect that $R0$ is faulty, having caused the request timers to expire and not sending its view change messages. After some number of unsucessful view changes, the correct replicas can send their votes to remove $R0$. When the CM receives f_B+1 votes from the correct replicas, it will finalize the voting round to remove $R0$. Once the replicas adopt the new configuration in the reconfigure message sent by the CM, they will elect a leader for the new view, and resume processing client requests.

6.3. Quorum Sizes

Table 1 shows the differences in the quorum sizes and how they influence reconfiguration. BFT-SMaRt changes to a new configuration by committing it as a request, therefore reconfiguration requires a quorum the size of the commit quorum, CQ. Committing configurations as requests means that the clients can keep the reply quorum, RQ, at $f_B + 1$, and the system guarantees that finished requests will persist across reconfigurations. A system using Phoenix requires a higher number of replicas provisioned ($n = 3f_B + f_C + 1$), but it can tolerate f_C crash faults in addition to f_B Byzantine faults. Since new configurations cannot be committed if there are f_B and f_C faulty replicas, reconfiguration is done through the view change with a VQ of $n - f_B - f_C$. Clients must wait for $n - f_B$ replies, and maintaining CQ at $n - f_B$ prevents a Byzantine replica from successfully overwriting committed values. Sync Phoenix is able to tolerate f_B and f_C faulty replicas whilst keeping n at $3f_B + 1$ by relying on synchrony during reconfiguration to maintain safety. Similar to BFT-SMaRt, CQ and VQ are kept at $n - f_B$, and reconfiguration only requires $f_B + 1$ in the worst case, but clients must also wait for $n - f_B$ replies.

7. Evaluation

In this section, we present our evaluation of Sync Phoenix, which consists of two sets of experiments. The goal of the first experiment is to measure the performance overhead of the two design choices against the baseline off-the-shelf implementation of BFT-SMaRt. In the second experiment, we look into the latency of reconfiguration to see how fast a system is able to recover when there are crash and Byzantine faults.

7.1. Large Quorum vs. Write Signatures

In this first experiment, we provisioned a cluster of four replica machines and 2 clusters of four client machines on Chameleon [13]. Each machine contains two Intel Xeon Gold 6242 "Cascade Lake R" processors (16

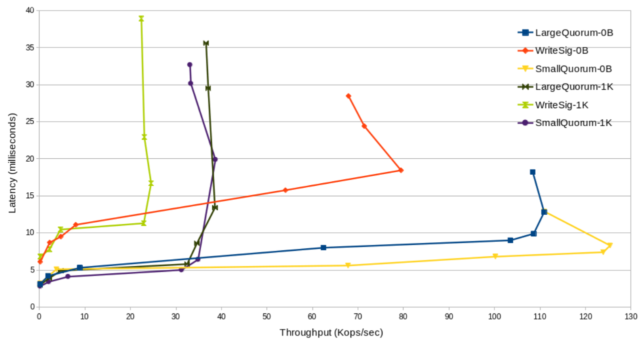

Figure 3: Throughput and latency for 0/0 and 1K/1K operations.

cores @ 2.8GHz, 32 threads, 192GB RAM), all running Ubuntu18.04. A total of 2400 ”closed-loop” client processes are launched across the eight client machines, all continuously issuing requests.

Figure 3 shows the throughput and latency of the three protocols (baseline, large reply quorum, *write* signatures). For each of these protocols, we evaluated their performance using two benchmarks. In the first benchmark, clients send requests of size 0 (empty), and replicas will also send back empty replies. Similarly, in the second benchmark, clients will send 1kB-sized requests and replicas will send 1kB-sized replies. We call these benchmarks the 0/0 benchmark (introduced by Casto et al. [8] to measure the maximum possible throughput) and the 1K/1K benchmark, respectively.

For the 0/0 benchmark, increasing the quorum size for the replies causes clients to wait longer between requests, which increases the end-to-end latency and slightly decreasing the throughput. As the payload and reply size increase to 1K, we see that the performance drops drastically due to the overhead of message transmission. This overhead is also the reason why the performance of the large quorum and small quorum are similar with the large payload, since the time to transmit the message is greater than the time it takes for clients to wait for the extra reply. For both benchmarks, adding signatures to the *write* phase greatly impacts the performance with respect to the other experiments. The reason is that in BFT-SMaRt, signatures in the commit are done speculatively; that is, when a replica receives a *write*, it spawns a separate thread to create and sign the *accept* before it even receives a quorum of the *write* message. This parallelism cannot be applied to the *write* message itself, as the replica has to wait to receive the *propose* before creating and signing the *write* message.

7.2. Reconfiguration

For the second experiment, we provisioned a cluster of four replicas and a total of 200 client processes distributed across four client machines. Here we reduced the number of client machines and client processes, because we are interested in reconfiguration latency but want the system saturated enough to visualize the drop in throughput. We configure the replicas so that each replica will vote to remove another replica if it has experienced faulty behavior from the other replica more than once. Figure 4 shows a timeline of the execution of the system with respect to its throughput.

In the time period **T1**, the replicas start up, establishes connections with each other. Once done, the replicas are then able to accept connections from the clients, and begin processing their requests. After 25 seconds, the Byzantine leader briefly halts, causing request timers on the other replicas to expire and the system to go into a view change. Time period **T2** shows the total time between the leader halting and operations resuming after a successful election, with 2 seconds of the time for the request timer to expire and another 2 seconds for the replicas to perform the view change. Because the Byzantine replica has caused the view change to occur, all the correct replicas will give this replica a mark.

During time period **T3**, the Byzantine replica behaves correctly again, then the system goes through a period of 30 seconds of normal operations, after which the new leader crashes. When the leader crashes, the communication channels that it has with the other replicas will be severed, and all will repeatedly try to reestablish the connection with the leader. The request timers will again expire after 2 seconds, causing another view change. However, the Byzantine replica now does not send any further messages, because it suspects that the old leader has crashed after failing to reestablish the communication channel multiple times. This leads to an insufficient number of view-change messages and an unsuccessful view change in **T4**. Normally, this is the point where BFT-SMaRt without Phoenix would stop and perform these unsuccessful view changes, one after another.

Since the correct replicas only received view-change messages from each other, they each mark the other two (Byzantine and crash) replicas and, with the Byzantine replica having two marks total, the correct replicas votes against the Byzantine replica. In **T5**, the CM receives the votes from the correct replicas, initiate a reconfiguration round, and subsequently removes the Byzantine replica. Finally, after a reconfiguration to replace the Byzantine replica, the new configuration has enough correct replicas to resume processing requests in time **T6**.

8. Related Work

Fault Detection. A failure detector can either be decoupled from the underlying distributed protocol to exist as an oracle that can be consulted, or coupled to allow protocol-specific actions to influence fault detection. Chandra et al. [10] introduced the concept of such oracles, but even the weakest failure detector cannot work in a fully asynchronous network due to FLP [15]. However, the assumption of partial synchrony can circumvent this result and allow such oracles to exist [11]. Malkhi and Reiter [25] defined a failure detector $\Diamond S$ that can detect *quiet* processes that do not participate in a protocol. Subsequent work by Doudou et

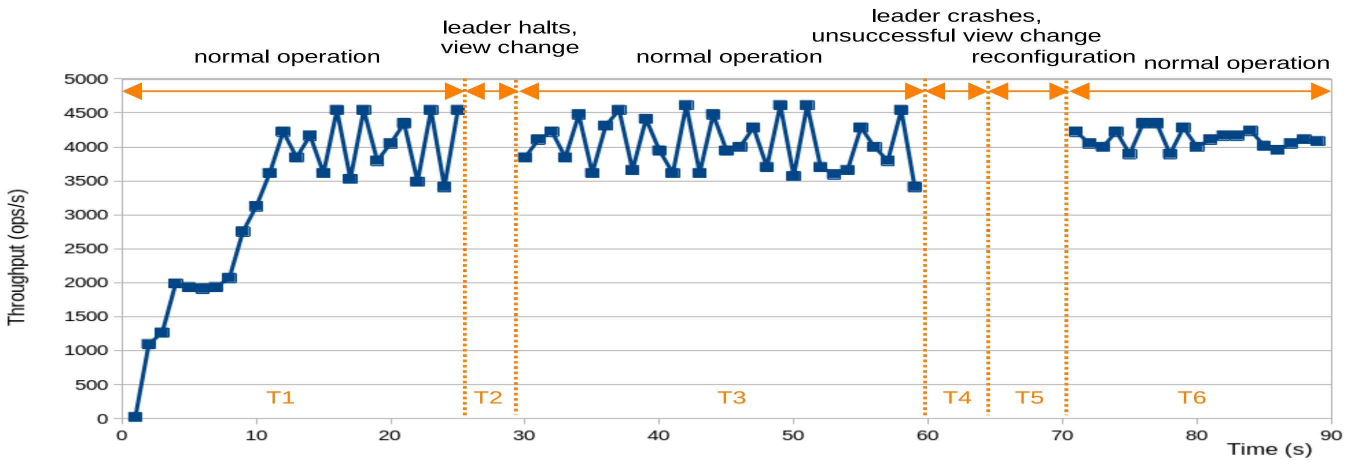

Figure 4: A system with $n = 4$ replicas going into reconfiguration after experiencing f_B and f_C faults.

al. [14] introduces a failure detector of the same class that can detect *mute* processes, the equivalent of *quiet* processes.

In BFT, failures detectors are typically coupled with the underlying consensus protocol. Haeberlen et al. [17] envisioned a fault detector that would allow replicas to determine if message omissions by a replica are malicious. Their approach requires replicas to frequently publish a signed digest of their log and to sign all messages it sends as well as to acknowledge all messages it receives. In addition to being inefficient, this approach to fault detection can be abused by malicious replicas that constantly issue challenges that requires other replicas to respond. Replicas in Aardvark [12] can detect subjective faults other than omissions, such as message flooding and invalid messages, and use these detections to blacklist faulty replicas. Regardless of whether fault detection is coupled or decoupled from the underlying consensus protocol, Phoenix can leverage the fault detector to make better reconfiguration choices.

Fault Recovery. Prior works on recovering faulty replicas in replicated systems fall into one of three categories: reconfiguration, proactive recovery, and reactive recovery. For CFT systems, protocols like Vertical Paxos [23] and Raft [28] reconfigure by having a special reconfiguration request committed by a quorum of the old active replica set as well as a quorum of the new active replica set. A faulty replica can also be recovered reactively by means of rebooting once it is detected [18], and there has been research looking into reducing the recovery time to increase availability [29] or by integrating diagnosis with recovery to support application-specific recovery actions [19]. All of these protocols work as long as there are only benign faults in the system, and would not work in the presence of Byzantine faults.

To reconfigure a replica set to remove a replica that could be Byzantine faulty, BFT-SMaRt [4] relies on a special process that sends a reconfiguration request to either add or remove a replica, which only needs to be committed by a quorum of the old active replica set. IA-CCF [33] allow a consortium of members in a permissioned ledger system to vote on referendums that updates the current configuration. When there are enough votes, the new configuration can be carried out as part of a transaction that must be committed and executed. Earlier systems such as Rampart [31] and SecureRing [21] provide group membership protocols that can be used to reconfigure the replica set, but subsequent work have shown that these protocols are not ideal for the Byzantine fault model [7].

Various works have looked into proactively recovering replicas that could be Byzantine faulty [7], [30]. A process can rely on a timer at each replica to initiate a recovery process which, in addition to rebooting, includes the replica discarding all the old keys used for encrypting communication that it had with the other replicas as well as the clients. Byzantine faulty replica can also be recovered reactively with the help of a failure detector oracle [35] .

Phoenix takes the approach of reactive reconfiguration to recover replicas because there are situations where faulty replicas cannot recover by a reboot [32], and also because it reduces downtime of the system. Proactive and reactive recovery requires rebooting of replicas and possibly re-establishing keys with clients and replicas, which can be expensive during runtime. However, this approach to recovery can complement reconfiguration. For example, a CM can decide to reintroduce removed replicas back into the system if they are still alive and have gone through a reboot and rekeying process.

9. Conclusion

BFT protocols typically come with fault detection mechanisms that allows replicas to detect faults, and a reconfiguration mechanism that allows an administrator to replace replicas. Phoenix integrates these two mechanisms to allow replicas to inform the administrator about suspected faults in order for the administrator to make better replacement choices. We also present Sync Phoenix, a version that works in the synchronous network model, allowing a system to recover when there are f_B Byzantine faults in addition to $f_C < f_B$ crash faults with a deployment of $3f_B+1$ replicas.

References

[1] Abraham, I., Malkhi, D., Nayak, K., Ren, L., and Yin, M. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 106–118.

[2] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference* (2018), pp. 1–15.

[3] AWS. Aws s3 availability event. https://status.aws.amazon.com/s3-20080720.html. Accessed: 2021-09-20.

[4] Bessani, A., Sousa, J., and Alchieri, E. E. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), IEEE, pp. 355–362.

[5] Buchman, E. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[6] Castro, M. *thesis-mcasto*, 2020 (accessed October 25, 2020).

[7] Castro, M., and Liskov, B. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* (2000).

[8] Castro, M., Liskov, B., et al. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[9] Chan, T. H., Pass, R., and Shi, E. Pili: An extremely simple synchronous blockchain. *Cryptology ePrint Archive* (2018).

[10] Chandra, T. D., Hadzilacos, V., and Toueg, S. The weakest failure detector for solving consensus. *Journal of the ACM (JACM) 43*, 4 (1996), 685–722.

[11] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM) 43*, 2 (1996), 225–267.

[12] Clement, A., Wong, E. L., Alvisi, L., Dahlin, M., and Marchetti, M. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI* (2009), vol. 9, pp. 153–168.

[13] Cloud, C. *A configurable experimental environment for large-scale edge to cloud research*, 2021 (accessed September 2, 2021).

[14] Doudou, A., Garbinato, B., Guerraoui, R., and Schiper, A. Muteness failure detectors: Specification and implementation. In *European Dependable Computing Conference* (1999), Springer, pp. 71–87.

[15] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[16] Gupta, S., Hellings, J., and Sadoghi, M. Fault-tolerant distributed transactions on blockchain. *Synthesis Lectures on Data Management 16*, 1 (2021), 1–268.

[17] Haeberlen, A., Kouznetsov, P., and Druschel, P. The case for byzantine fault detection. In *HotDep* (2006).

[18] Huang, Y., and Kintala, C. Software implemented fault tolerance: Technologies and experience. In *FTCS* (1993), vol. 23, IEEE Computer Society Press, pp. 2–9.

[19] Joshi, K. R., Hiltunen, M. A., Sanders, W. H., and Schlichting, R. D. Automatic model-driven recovery in distributed systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)* (2005), IEEE, pp. 25–36.

[20] Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S. V., Schröder-Preikschat, W., and Stengel, K. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), pp. 295–308.

[21] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. The securering protocols for securing group communication. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences* (1998), vol. 3, IEEE, pp. 317–326.

[22] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (2007), pp. 45–58.

[23] Lamport, L., Malkhi, D., and Zhou, L. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing* (2009), pp. 312–313.

[24] Liu, S., Viotti, P., Cachin, C., Quéma, V., and Vukolić, M. {XFT}: Practical fault tolerance beyond crashes. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 485–500.

[25] Malkhi, D., and Reiter, M. Unreliable intrusion detection in distributed computations. In *Proceedings 10th Computer Security Foundations Workshop* (1997), IEEE, pp. 116–124.

[26] Martin, J.-P., and Alvisi, L. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing 3*, 3 (2006), 202–215.

[27] Nawab, F., and Sadoghi, M. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), IEEE, pp. 124–135.

[28] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)* (2014), pp. 305–319.

[29] Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., et al. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Tech. rep., Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[30] Reiser, H. P., and Kapitza, R. Hypervisor-based efficient proactive recovery. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)* (2007), IEEE, pp. 83–92.

[31] Reiter, M. K. The rampart toolkit for building high-integrity services. In *Theory and practice in distributed systems*. Springer, 1995, pp. 99–110.

[32] Rodrigues, R., and Liskov, B. Byzantine fault tolerance in long-lived systems.

[33] Shamis, A., Pietzuch, P., Canakci, B., Castro, M., Fournet, C., Ashton, E., Chamayou, A., Clebsch, S., Delignat-Lavaud, A., Kerner, M., et al. {IA-CCF}: Individual accountability for permissioned ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 467–491.

[34] Sousa, J., and Bessani, A. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference* (2012), IEEE, pp. 37–48.

[35] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems 21*, 4 (2009), 452–465.

[36] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., and Verissimo, P. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers 62*, 1 (2011), 16–30.

[37] Wang, P., Dean, D. J., and Gu, X. Understanding real world data corruptions in cloud systems. In *2015 IEEE International Conference on Cloud Engineering* (2015), IEEE, pp. 116–125.

[38] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 347–356.