PrestigeBFT: Revolutionizing View Changes in BFT Consensus Algorithms with Reputation Mechanisms

Anonymous Author(s)

Abstract

This paper proposes PrestigeBFT, a novel leader-based BFT consensus algorithm that addresses the weaknesses of passive view-change protocols. Passive protocols blindly rotate leadership among servers on a predefined schedule, potentially selecting unavailable or slow servers as leaders. PrestigeBFT proposes an active view-change protocol using reputation mechanisms that calculate a server's potential correctness based on historic behavior. The active protocol enables servers to campaign for leadership by performing reputation-associated work. As such, up-to-date and correct servers with good reputations are more likely to be elected as leaders as they perform less work, whereas faulty servers with bad reputations are suppressed from becoming leaders by being required to perform more work. Under normal operation, PrestigeBFT achieves 5× higher throughput than the baseline that uses passive view-change protocols. In addition, PrestigeBFT remains unaffected under benign faults and experiences only a 24% drop in throughput under a variety of Byzantine faults, while the baseline throughput drops by 62% and 69%, respectively.

1 Introduction

The rapid development of distributed systems has spurred extensive research on Byzantine fault-tolerant (BFT) consensus algorithms. Among them, leader-based BFT algorithms have been favored by practical applications due to their high performance. These algorithms operate state machine replication to produce deterministic results using two distinct protocols: the view-change and replication protocol. The view-change protocol selects a leader (primary) for each view, while the replication protocol enables the leader to initiate consensus with followers (backups). While prior research has primarily concentrated on the replication protocol to optimize system performance, the importance of the view-change protocol has often been overlooked. However, today, the view-change protocol has become a critical factor in system performance as view changes occur more frequently than previously assumed due to various factors associated with fault tolerance, performance criteria, and decentralization with fairness imperatives [3, 6, 9, 22, 29, 40, 46]. For example, ① network problems, bursty workloads, operator errors, and software bugs can result in leader failure[6], resulting in more frequent view changes as applications scale up [40]. ② Since faulty leaders can intentionally slow down processing without triggering timeouts [3], some approaches monitor a leader's performance and invoke a view change if the performance falls below a set threshold [9, 22]. Furthermore, ③ faulty leaders can unfairly handle client requests [46, 81], resulting in some approaches frequently changing leadership to mitigate unfairness [29]. Evidently, the view-change protocol is vital for system performance, especially in blockchain applications where frequent view changes are becoming the norm [17, 28].

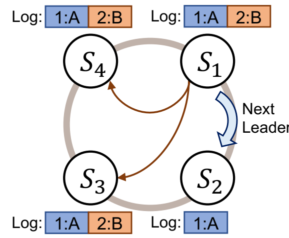


Figure 1. The passive view-change protocol follows a fixed schedule to rotate leadership; it cannot avoid an already crashed server or a slow server to become a leader (S_2).

Numerous state-of-the-art BFT algorithms (e.g., [27, 38, 45, 72–74, 78]), despite optimizing the replication protocol extensively, all rely on similar view-change mechanisms introduced by PBFT [18]. The passive protocol follows a predefined schedule to rotate leadership among servers: for a total of n servers, a leader server (L) for a view V is decided such that L = V mod n. For example, in Figure 1, S_1 is the leader in $V1$; when a view change occurs, S_2 becomes the leader in $V2$, and S_3 in $V3$, and so on.

Unfortunately, the passive protocol lacks robustness and efficiency. During a view change, since all servers blindly follow a predefined schedule and rotate leadership, the passive protocol cannot skip a scheduled server that is already unavailable, which leads to weak robustness. In addition, the passive protocol can result in inefficiency in replication because it cannot ensure *optimistic responsiveness* (OR) [64], which requires a non-faulty leader to be up-to-date and make immediate progress after being assigned [7, 78]. However, when a slow server is rotated to be the leader by a passive protocol, it must sync to become up-to-date first and then starts operating consensus. Thus, BFT algorithms using the passive view-change protocol have to add a sync-up phase after each value is committed to obtain OR (e.g., from two-phase to three-phase in HotStuff [78]), but this comes at the cost of reduced throughput and increased latency due to additional messages and rounds.

Figure 1 shows an n = 4 system where S_1 is the leader in $V1$, and value B has been replicated at sequence number (SN) 2 among S_1, S_3, and S_4. Next, we assume that S_1 fails, causing a view change to take place. ① If S_2 has already crashed, S_2 will still blindly be assigned to be the leader in

$V2$, and the system must wait for timeouts from $f+1$ servers to realize that S_2 has failed before moving on to S_3 in $V3$. On the other hand, ② if S_2 is alive, it cannot make immediate progress because it must first sync to become up-to-date (i.e., know the highest SN). Therefore, in a circle of leadership rotations among n=3f+1 servers (i.e., from S_1 to S_n), the probability of encountering an unavailable or slow leader is $f/(3f+1) \approx 33\%$ in the worst case.

In order to improve the robustness and efficiency of view changes, we set out to investigate active view changes where servers no longer follow a predefined schedule. Raft's leader election mechanism ushered a way for designing active view-change protocols under non-Byzantine (benign) failures [63]. It allows servers to actively campaign for leadership upon detecting a leader's failure and vote for an up-to-date server to become a new leader. Consequently, it can prevent unavailable and slow servers from becoming leaders. However, in the context of BFT, this approach alone is insufficient. While servers are empowered to campaign for leadership, it also opens the door for Byzantine servers to repeatedly initiate new view changes to seize control of leadership and neglect replication (repeated view change attacks). Therefore, resolving this issue is essential for active view changes to be deployed under BFT.

To tackle this challenge, we propose PrestigeBFT, a new BFT consensus algorithm with an active view-change protocol featuring reputation mechanisms. **Our reputation mechanism utilizes a server's behavior history to generate a reputation value that reflects the likelihood of the server's correctness. This reputation value then determines the probability of the server being selected as a new leader during the active view-change protocol.**

PrestigeBFT penalizes suspiciously faulty behavior with worsening reputations, while it rewards protocol-obedient behavior with improving reputations. A server's reputation value is utilized to assess its likelihood of becoming a new leader. During view changes, the active view-change protocol imposes computational work on each leadership campaigner, where the difficulty of the computation is determined by the campaigner's reputation value. Correct servers, who exhibit protocol-obedient behavior and thus maintain a "good" reputation, perform negligible computational work. On the other hand, faulty servers, whose behavior history has led to a "bad" reputation, perform more time-consuming computational work. Thus, correct and up-do-date servers are more likely to be elected than faulty servers over time.

Equipped with its reputation-embedded active view-change protocol, PrestigeBFT demonstrates both robustness and efficiency. During normal operation, its replication achieves optimistic responsiveness with 5× higher throughput than HotStuff [78]. Under benign faults, passive view-change protocols suffer from an approximate 65% drop in throughput, whereas PrestigeBFT's performance remains unaffected. Furthermore, under a variety of Byzantine faults, PrestigeBFT's reputation engine swiftly suppresses faulty servers from attaining leadership, resulting in only a 24% drop in throughput compared to that under normal operation.

PrestigeBFT makes the following key contributions:

- Its view-change protocol is the first active protocol operating under BFT. By enabling servers to proactively campaign for leadership, it prevents the election of unavailable or slow servers, thereby achieving optimistic responsiveness.
- Its reputation mechanisms effectively convert a server's behavior history during replication and view changes into a reputation value that indicates the server's likelihood of being correct. The reputation value is crucial in determining a server's eligibility for leadership in the view-change protocol.
- It demonstrates a unique combination of robustness and efficiency, with improved performance even under Byzantine failures. Faulty servers are quickly suppressed during view changes, and their probability of being elected rapidly decreases after they perform attacks that relegate their reputations.
- It is fully open-source, including a wide range of examples, discussions, visualized proofs, and videos that improve understandability at: /r/prestigebft-1FE0/ [51].

2 PrestigeBFT overview

PrestigeBFT, similar to other state-of-the-art leader-based BFT consensus algorithms (e.g., PBFT [18] and HotStuff [78]), moves through a succession of system configurations called views. Views are integers that increase monotonically. Each view starts with a *view-change period* conducted by the view-change protocol that decides a leader and may follow with a *replication period* conducted by the replication protocol that achieves consensus for client requests.

PrestigeBFT architecture. Besides the view-change and replication protocols, PrestigeBFT establishes a unique reputation mechanism (shown in Figure 2). At any given time, a server operates in one of four states: *follower*, *redeemer*, *candidate*, and *leader*. Under normal operation, there is one leader and the other servers are followers, where all servers operate under the replication protocol, which conducts consensus for committing client requests, producing `txBlocks` (shown in Figure 3) that record quorum certificates (QCs) [36]. When a view change is invoked, the view-change protocol produces `vcBlocks` that record leadership and servers' reputation information in the new view. Note that `txBlocks` and `vcBlocks` are the deterministic consensus results of replication and view change.

Each server has a reputation engine that has predefined rules to calculate reputation values. The reputation engine is utilized during view changes, wherein it retrieves the necessary states of `txBlocks` and `vcBlocks` stored in the state machine through read operations. Based on the retrieved

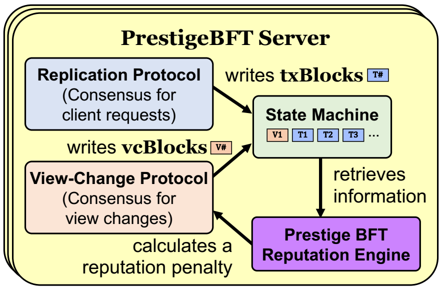


Figure 2. PrestigeBFT architecture.

Attributes in Transaction Blocks (TX Consensus)	
`header`	addresses of this block and the previous `txBlock`
// block agreement fragment	
`v`	view number
`n`	block index (sequence number)
`ordering_QC`	QC collected for confirming the ordering action
`commit_QC`	QC collected for confirming the commit action
// transaction fragment	
`tx[]`	a list storing transactions contained in this block, where the size of `tx[]` is the batch size
`status[]`	a list storing the consensus result of each transaction in `tx[]`; the result is a Boolean value

Attributes in View-Change Blocks (VC Consensus)	
`header`	addresses of this block and the previous `vcBlock`
// leader election fragment	
`v`	view number
`leaderId`	leader server's ID
`conf_QC`	QC collected for confirming leader failures
`vc_QC`	QC collected for confirming leadership legitimacy
// reputation fragment	
`rp map[Id]int`	a map storing the reputation penalty (rp) for each server (Id) in this view
`ci map[Id]int`	a map storing the compensation index (ci) for each server, tracking `txBlocks` used in calculation

Figure 3. Attributes in `vcBlocks` and `txBlocks`.

information, it calculates a **reputation penalty** (an integer). The penalty reflects the server's likelihood of correctness and determines the work that the server performs to become a candidate. The process for a server, initially as a follower, to become a leader works at a high level as follows:

1. Each server is initially a follower. If a follower triggers a view change confirmed by f+1 servers in view V, it will campaign for leadership and become a redeemer.
2. The redeemer increments its view to $V + 1$ and gets its reputation penalty from its reputation engine. It then performs computation determined by the reputation penalty; once completed, it transitions to a candidate.
3. The candidate starts a leader election by collecting votes from $2f + 1$ servers; if it succeeds in time, it becomes the leader in view $V + 1$.
4. The leader prepares a `vcBlock` including the election result and updated reputation information and then broadcasts the `vcBlock` to others.

The reputation penalty is critical in successful elections. Servers with higher penalties are more suspected of being faulty and will perform more computational work, making the election process substantially more difficult for them. Next, we show how the reputation mechanism translates a server's past behavior to a reputation penalty that serves as an indicator of the server's correctness.

3 PrestigeBFT's reputation mechanism

The reputation mechanism analyzes a server's behavior in past replication and view changes and produces a reputation penalty (rp), represented as an integer. A higher penalty corresponds to a worse reputation and indicates a higher level of suspicion that the server may be malicious.

The calculation of rp involves two steps: penalization and compensation. It increases rp for a server that seeks to become the leader (penalization) and reduces rp if the server exhibits good historic behavior (compensation). **Therefore, a server's reputation penalty may either increase, decrease, or remain unchanged from its current value after being assessed by the reputation engine.** Algo. CalcRP shows the calculation workflow.

Algorithm 1 Calculate-Reputation-Penalty (CalcRP)

Require: V', vcBlock, txBlock, Id ▷ V' is the new view
V, rp = vcBlock.v, vcBlock.rp[Id] ▷ get current view and penalty
new_rp_temp = Eq.1(V', V, rp)
ti, ci = `txBlock.n`, vcBlock.ci[Id] ▷ get ti and ci from stored blocks
P = [ ]int{rp} ▷ init penalty set with rp
while vcBlock.header.preVcBlock != nil **do**
  vcBlock = vcBlock.header.preVcBlock ▷ iterates to the first block
  P.add(vcBlock.rp[Id]) ▷ P contains all past penalties
delta_tx, delat_vc = Eq.2(ti, ci), Eq.3(P, rp)
new_rp = Eq.4(new_rp_temp, delta_tx, delat_vc)
return new_rp, ci

Init. The initial view and penalty can be defined differently. For simplicity, we set initial view to $V1$ and $rp^{(1)} = 1$.

Step 1: Penalization. A server's rp is increased for campaigning for leadership for the new view (V') following Eq. 1. The increase in rp is the increase in view numbers, which prevents Byzantine servers from overloading the view data structure (an integer). The higher the increase in views, the higher the penalty will be. Correct servers will always increase their views by one.

$$rp_{temp}^{(V')} = rp^{(V)} + (V' - V) \tag{1}$$

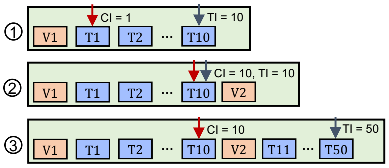
After applying penalization, the calculation proceeds to the second step, which involves deducting from the increased rp if the server's behavior history meets certain criteria.

$$\delta_{tx} = \frac{\mathtt{ti} - \mathtt{ci}}{\mathtt{ti}} \quad (2) \qquad \delta_{vc} = 1 - Sigmoid(\frac{rp^{(V)} - \mu_{\mathcal{P}}}{\sigma_{\mathcal{P}}}) \quad (3)$$

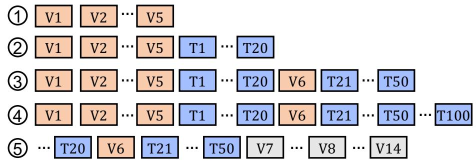
$$\delta = rp_{temp}^{(V')} C_{\delta} \delta_{tx} \delta_{vc}, \quad rp^{(V')} = rp_{temp}^{(V')} - \lfloor \delta \rfloor \tag{4}$$

Step 2: Compensating good behavior history. The compensation has two criteria: *incremental log responsiveness* (δ_{tx} in Eq. 2), which is calculated using `txBlocks`, and *leadership zealousness* (δ_{vc} in Eq. 3), which is calculated using `vcBlocks`. The final rp is calculated in Eq. 4 with a possible deduction considering both criteria.

The first criterion (δ_{tx}) considers good behavior to be up-to-date replication. In Eq. 2, `ti` is the number of `txBlocks` this server has committed, which is the sequence number of its latest `txBlock`, and `ci` is the compensation



(a) Examples of calculating incremental log responsiveness (δ_{tx}).



(b) Examples of historic behavior. The server is the leader in highlighted `V#` but not in gray `V#`.

	`ci`	`ti`	δ_{tx}	$\mathcal{P}$	δ_{vc}	δ	$rp^{(V')}$
①	1	1	**0**	{1, 2, 3, 4, 5}	0.19	0	**6**
②	1	20	**1**	{1, 2, 3, 4, 5}	0.19	1.14	**5**
③	20	50	**0.6**	{1, 2, 3, 4, 5, 5}	0.25	0.89	**6**
④	20	100	**0.8**	{1, 2, 3, 4, 5, 5}	0.25	1.2	**5**
⑤	20	50	**0.6**	$\mathcal{P}^5$	0.36	1.29	**5**

$\mathcal{P}^5$={1, 2, 3, 4, 5, 5, ..., 5} //5 appears 10 times

(c) Breakdown of calculating compensation in 4b; $rp^{(V')}$ will be the new penalty in V'.

Figure 4. Examples of calculating reputation penalty (rp), where "`V#`" are `vcBlocks` and "`T#`" are `txBlocks`.

index representing the number of `txBlocks` this server has used for past compensation stored in the current `vcBlock` (Line 3). Initially, `ti`=1 and `ci`=1, so $0 \le \delta_{tx} \le 1$.

Figure 4a shows three examples of calculating δ_{tx} for a server (say S_a). ① S_a has replicated 10 `txBlocks` in $V1$, so its `ci`=1 and `ti`=10. ② If S_a campaigns for leadership for $V2$, after applying Eq. 2, its $\delta_{tx} = 0.9$ with 10 `txBlocks` used for compensation. If S_a is elected, `ci`=10. ③ If S_a then replicates 50 `txBlocks` in total and campaigns for $V3$, its $\delta_{tx} = 0.8$. Therefore, to receive a higher δ_{tx}, the reputation mechanism entices servers to replicate more `txBlocks`.

Considering log responsiveness is a common aspect of reputation-based approaches (e.g., DiemBFT [29]), where the more transactions a server replicates, the more reliable it is perceived to be. Nevertheless, since PrestigeBFT is an active view-change protocol, relying solely on behavior in replication is inadequate. *It is crucial to consider cases where Byzantine servers repeatedly acquire leadership but make limited progress in replication.* Therefore, in addition to log responsiveness, our reputation mechanism considers a server's historic penalties in previous view changes as well.

The second criterion (δ_{vc}) considers good behavior to be having gradually increasing penalties during past view changes. δ_{vc} first computes the z-score of a server's current penalty in relation to its past penalties, taking into account the rate of change of the current penalty over past penalties. It retrieves the server's historic penalties stored in `vcBlocks` and adds them into a set $\mathcal{P}$ (Line 5 to 7); then, it calculates the mean ($\mu_{\mathcal{P}}$) and standard deviation ($\sigma_{\mathcal{P}}$) of $\mathcal{P}$. Finally, the `Sigmoid` function normalizes the z-score between 0 and 1. Thus, $0 < \delta_{vc} < 1$.

Therefore, a higher δ_{vc} value indicates a slower increase in penalties, which is more towards the behavior of correct servers. Since correct servers adhere to the protocol for triggering view changes, they are unlikely to be penalized while regaining leadership in each view; doing so would require proactively performing significantly increasing computation to "fight against" the penalty increase (discussed in §4.2).

Finally, after obtaining δ_{tx} and δ_{vc}, the new rp is calculated by Eq. 4, where C_δ is a constant that may be used by different applications to adjust the effect of δ_{tx} and δ_{vc}. For simplicity, we set $C_\delta = 1$. Since $0 \le \delta_{tx} \le 1$ and $0 < \delta_{vc} < 1$, the deduction (δ) is a portion of the increased penalty of Eq. 1; i.e., $0 \le \delta < rp^{(V')}_{temp}$. As such, the reputation mechanism will raise suspicion of malicious behavior by increasing rp if it observes a pattern of penalized leadership repossession with limited replication. Conversely, it will decrease rp if it observes historic behavior that implies high log responsiveness and gradually increased or unchanged historic penalties.

More examples. Figure 4b and 4c provide five examples of calculating rp for different server behavior. They show how the reputation mechanism responds to suspicious malicious behavior and to protocol-obedient behavior.

① Server S_a has been the leader from $V1$ to $V5$ without replication. Thus, its δ_{tx} remains 0, resulting in no compensation in Eq. 4 with its rp only increasing. If S_a campaigns for leadership for the next view ($V6$), its rp will increase to $rp^{(6)}$=6.

② If S_a conducts consensus for 20 `txBlocks` in $V5$ and then campaigns for leadership for the next view, its δ_{tx}=1. It will receive a compensation of 1 with unchanged rp ($rp^{(6)}$=$rp^{(5)}$=5).

Analysis. S_a's behavior in ① is extremely suspicious to be malicious, as it keeps repossessing leadership without making progress in replication. The reputation mechanism captures this pattern and keeps increasing its reputation penalty. Compared to ①, S_a's behavior in ② reduces suspicion as it starts to replicate transactions; the reputation mechanism encourages this behavior and grants compensation.

③ In $V6$, S_a replicates 20 more `txBlocks`. If S_a starts to campaign for $V7$, its δ_{tx}=0.6 by Eq. 2 as `ci`=20 and `ti`=50. Thus, S_a receives no compensation with its penalty increasing to $rp^{(7)} = 6$.

④ If S_a replicates more `TxBlocks` for a total of 100 in $V6$, its $\delta_{tx} = 0.8$. In this case, if S_a campaigns for leadership in $V7$, it will receive a compensation of 1 with its rp remaining unchanged ($rp^{(7)}$=5).

Analysis. Incremental log responsiveness (δ_{tx}) expects an increasing number of `txBlocks` after each compensation (e.g., S_a cannot get compensated in ③ but can be compensated in ④); this prevents a server from frequently occupying leadership but only making limited progress in replication.

⑤ Continuing in ③, assume S_a is no longer eager to leadership after $V6$, staying as a follower from $V7$ to $V14$ (the gray `vcBlocks` where other servers are leaders). Its δ_{vc} kept increasing as its penalty remains unchanged as 5 from $V7$ to $V14$. If S_a campaigns for leadership in $V15$, it will be compensated by 1.

Analysis. S_a in ⑤ can be compensated as ④ with the same level of replication in ③ when it stops repossessing leadership with a suspicious history. It shows that δ_{vc} incentivizes servers with a history of increasing penalties to become indifferent to leadership.

The examples show the effectiveness, efficiency, and simplicity of the reputation mechanism in transforming a server's prior actions into an integer value (rp) that is indicative of its correctness. In addition, the calculation schema is highly adaptable and can be customized for specific use cases. For example, users can define the criteria for useful `txBlocks` in δ_{tx} and alter the impact of $\delta_{tx}\delta_{vc}$ by modifying C_δ. More examples can be found in [51].

Features. *The reputation mechanism does not incur additional cost on replication.* The reputation engine is independent (shown in Figure 2) and involved only when a server has become a redeemer in a view change; i.e., the replication has already stopped.

Moreover, *the reputation mechanism never writes to the state machine.* The reputation engine operates as a "consultant" who calculates an rp when called. It never changes a server's rp in the current view; i.e., in a given view V, a server's rp remains unchanged throughout V. The calculated rp will become a server's new rp in the next view only if it is elected as the new leader through VC consensus.

4 The PrestigeBFT consensus algorithm

This section introduces PrestigeBFT's system model, active view-change protocol, and replication protocol.

4.1 System model

We adopt the partial synchrony network model introduced by Dwork et al. [33], where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all transmissions between two non-faulty servers arrive within time Δ. PrestigeBFT does not require network synchrony to provide safety, but it requires partial synchrony to provide liveness.

We use a Byzantine failure model, meaning that faulty servers may behave arbitrarily. We assume independent server failures where each server represents an independent entity. PrestigeBFT tolerates up to f Byzantine servers out of n=$3f + 1$ servers (i.e., $f = \lfloor \frac{n-1}{3} \rfloor$) with no limit on the number of faulty clients. Similar to other BFT algorithms [18, 27, 38, 47, 78, 81], we do not consider DDoS attacks (e.g., buffer overflow attacks), which are often handled by lower-level mechanisms, such as rate limiting and admission control, outside of the scope of consensus algorithms.

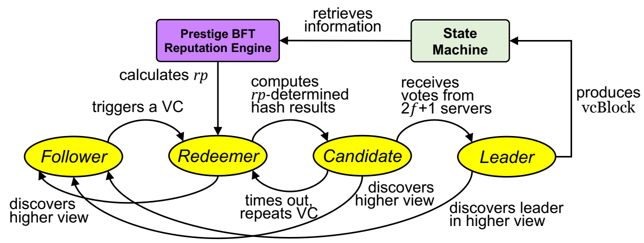


Figure 5. Server states transition in view changes.

Cryptographic primitives and quorums. PrestigeBFT applies (t, n) threshold signatures where t out of n servers collectively sign a message [69]. Threshold signatures can convert t individually signed messages (of size $O(n)$) into one fully signed message (of size $O(1)$), which can then be verified by all n servers, proving that t servers have signed it. PrestigeBFT uses threshold signatures to form quorum certificates (QCs) in consensus by setting t=$2f + 1$.

Attack vector and threat model. We allow for a faulty server to collude with the other $f-1$ faulty servers as well as an unlimited number of faulty clients. The set of faulty servers can change dynamically, with servers transitioning between correct and faulty states, as long as the total number of faulty servers does not exceed f. Faulty servers can behave arbitrarily and maliciously, but we assume that faulty servers cannot intervene to prevent the state changes of non-faulty servers specified by the algorithm, which is a fundamental assumption of Byzantine fault tolerance [8]. For example, faulty servers cannot prevent a non-faulty server from delivering messages to other non-faulty servers after GST. We also assume that a faulty server (and its colluding faulty servers) are computationally bound. They cannot produce a valid signature of a non-faulty server. It is worth noting that our system model is the same as other standard partially synchronous BFT algorithms [15, 18, 38, 78], without additional assumptions.

4.2 The active view-change protocol

The view-change (VC) protocol achieves VC consensus for deciding on a new leader and updating its reputation penalty and compensation index (shown in Figure 3). The VC protocol attains the following properties:

- **P1** At most one leader can be elected in a view.
- **P2** An elected leader has the most up-to-date replication (ensuring optimistic responsiveness).
- **P3** An elected leader's reputation penalty and the correspondingly performed computation can be verified by all non-faulty servers.

Next, we describe the VC protocol with server state transitions (illustrated in Figure 5) in Algo. State-Transition and demonstrate how these properties are achieved.

Algorithm 2 State-Transition

```
    ▽ As a follower
1:  upon receiving a COMPT do
2:      if COMPT.δ_c is valid then send COMPT to Leader
3:      new timer for COMPT.tx
4:      while timer do
5:          if COMPT.tx is committed then return        ▷ Leader is correct
6:      broadcast CONFVC ← ⟨V, COMPT, σ_{S_i}⟩ and reset timer
7:      while timer do
8:          upon receiving f + 1 REVC messages do
9:              convert REVCs to conf_QC and transition to Redeemer
10:             return                     ▷ Leader is faulty; start a new VC
11:     tag client COMPT.c                             ▷ Client can be faulty
12: upon receiving a CONFVC do
13:     if CONFVC.COMPT has been received from COMPT.c then
14:         send REVC ← ⟨V, σ_{S_i}⟩
15: upon receiving a CAMPVC do                  ▷ voting criteria in §4.2.3
16:     if CAMPVC.V' < myVcBlock.v then return
17:     if !C1 then return                         ▷ vote only once in a view
18:     if !C2 then return                     ▷ verify if this VC is confirmed
19:     if CAMPVC.V > myVcBlock.v then
20:         SYNCUP(myVcBlock, CAMPVC.V)                ▷ sync-up view changes
21:     ti = CAMPVC.txBlock.n
22:     if ti < myTxBlock.n then return            ▷ C3 verifying replication
23:     if ti > myTxBlock.n then
24:         SYNCUP(myTxBlock, ti)                        ▷ sync-up replication
25:     rp', ci' = CALCRP(CAMPVC.V', myVcBlock, txBlock, CandID)
26:     if ci'! = CAMPVC.ci then return
27:     if rp'! = CAMPVC.rp then return                    ▷ C4 verifying rp
28:     hr' = HASH(CAMPVC.txBlock, CAMPVC.nc)
29:     if !PREFIX(hr', rp') then return           ▷ C5 verifying computation
30:     send VOTECP ← ⟨CAMPVC.V', σ_{S_i}⟩
    ▽ As a redeemer
31: retrieve vcBlock, txBlock, V
32: V' = V + 1                                       ▷ increments current view
33: rp, ci = CALCRP(V', vcBlock, txBlock, myID)
34: stop replication in V
35: newThread (upon discovering higher view transition to Follower)
36: while 1 do
37:     nc = GEN-NONCE(rand)                      ▷ Randomly generates a string
38:     hr = HASH(txBlock, nc)
39:     if PREFIX(hr, rp) then                ▷ Check if hr has a prefix of rp 0s
40:         transition to Candidate
41:         return V, V', rp, nc, hr, ci, txBlock
    ▽ As a candidate
42: reset timer                                          ▷ starts an election
43: broadcast CAMP ← ⟨conf_QC, V, V', rp, nc, hr, ci, txBlock, σ_{S_i}⟩
44: newThread (upon discovering higher view transition to Follower)
45: while timer do
46:     upon receiving 2f + 1 VOTECP messages do
47:         convert VOTECPs to vc_QC and transition to Leader
48: transition to Redeemer
    ▽ As a leader
49: new vcBlock{V', myID, conf_QC, vc_QC, vcBlock.rp, vcBlock.cp}
50: vcBlock.rp[myID], vcBlock.ci[myID] = rp, ci
51: broadcast vcBlock
52: newThread (upon discovering higher view transition to Follower)
53: upon receiving 2f+1 vcYES messages do
54:     store vcBlock and start replication in V' as Leader
```

4.2.1 As a Follower.

Each server is initially a follower and has a timer with a random timeout. The timeout should be sufficiently greater than network latency (Δ), allowing ample time for a correct leader to complete consensus (e.g., timeout range = [300, 600 ms] for a Δ=30 ms).

View changes can be invoked by policy-defined criteria and failure detection. The former can be implemented differently according to application specifications. For example, a throughput-threshold policy that changes a view if a leader fails to operate at an expected throughput (e.g., Aardvark [22]) or a timing policy that changes a view every 5 minutes. The detection of a leader's failure involves both clients and servers. If a client (`c`) cannot confirm its proposed transaction (`tx`) in time through the replication protocol, it broadcasts a complaint (COMPT) message with the proposal message (PROP in §4.3) it sent to the leader, including `tx`, `c`, and the client's signature σ_c, suspecting a leader's failure.

Assume a server S_i operates as a follower; after receiving a COMPT message, S_i verifies and relays it to the leader and then waits for consensus to be completed (Line 2). If `tx` is committed before the timer expires (Line 5), it shows that the leader is still correct. Otherwise, S_i suspects that the leader or client may be faulty and starts an inspection by broadcasting a CONFVC message (Line 6), where V is the current view number and σ_{S_i} is the signature that S_i's signs this message. The other followers, after receiving a CONFVC from S_i, check if they have received the same CMPT from client c (Line 12). If so, they reply with a REVC message.

If S_i receives f+1 replies in time (including itself), it converts them to a threshold signature with t=f+1, which forms a quorum certificate (`conf_QC`). S_i considers the leader faulty and starts a view change by transitioning to a redeemer. On the other hand, if f+1 replies cannot be collected in time, S_i will tag client `c` as faulty.

This failure detection mechanism prevents faulty clients, faulty servers, and their collusion from inflicting unnecessary view changes on correct followers. Since a correct client is required to broadcast its complaint to all servers, at least 2f+1 correct servers can relay the complaint to the leader. Thus, a view change is confirmed by at least a correct server (Line 8). Formal proofs are available in [51]. Note that we do not assume DDoS attacks (in §4.1), such as faulty clients overwhelming servers by pouring complaints. This can be handled by rate control or blacklisting tagged clients.

4.2.2 As a Redeemer.

After becoming a redeemer, S_i retrieves the view number (V), the `vcBlock` of the current view, and the latest committed `txBlock` (`txBlock.n` is the highest among all `txBlocks`). S_i first increments its view to V' and calls the reputation engine to get its rp and ci for view V'. Next, it computes a hash puzzle (similar to Proof-of-Work [59]): it generates a random string (nc) and hashes the combination of nc and `txBlock` until the hash result (hr) has a prefix of rp zero bytes (e.g., hr="0000966sv0d3..." under

rp=4). Thus, the higher the rp is, the more iterations it takes to find a prefix with rp leading zeros. Servers with a higher rp will spend more time and energy to "redeem themselves" from the imposed work, while servers with a lower rp can complete the computation and transition to a candidate more quickly.

In addition, a redeemer transitions back to follower when it discovers a leader of a higher view. If the leader's `vcBlock` is valid (see §4.2.4), it indicates that the redeemer is out-of-sync. The redeemer will abort ongoing computation and operate as a follower in the higher view.

4.2.3 As a Candidate.

After becoming a candidate, S_i broadcasts a campaign message (Line 43), where `conf_QC` was collected when confirming this view change (Line 9), V to `txBlock` are the results after the redeemer state, and σ_{S_i} is the signature that S_i signs this message with. Then, S_i waits for votes from the other servers. The other servers, operating as followers, vote for S_i (Line 15) with the following criteria:

- **C1** The follower has not voted in this view (Camp.V').
- **C2** The threshold of Camp.`conf_QC` is f+1.
- **C3** The candidate's replication is at least as up-to-date as the follower's.
- **C4** The candidate's rp can be recalculated and verified.
- **C5** The candidate's performed computational work aligns with its rp; i.e., Camp.hr has rp leading zeros.

C1 enforces that a server votes at most once in a view, and thus guarantees Property **P1** that at most one leader can be elected in a given view. **C2** guarantees that the current view change is necessary and was confirmed by at least one correct server.

Since BFT consensus operates in QCs of size $2f + 1$, up to f servers can be correct but stale (fallen behind in their logs including `txBlocks` and `vcBlocks`). For example, if S_1 fails in Figure 1 and S_3 becomes a candidate, S_4 and S_3 have identical logs. However, while S_2 is correct, it has stale logs at the time of S_1's crash. Therefore, stale servers must sync to update their logs before verifying requests from candidates. To achieve this, the SyncUp function acquires needed blocks from the candidate:

```
function SyncUp(btype, end)            ▷ btype is a block interface
    start = btype.id                   ▷ id = view/n in vcBlocks/txBlocks
    acquire blocks[] from start to end from remote
    validate all blocks in blocks[] through their QCs
    btype = blocks[:-1]                ▷ set myBlock to the latest block
```

If followers fall behind in view changes, they call SyncUp to acquire missing `vcBlocks` (Line 20). Then, they check if the candidate's replication is at least up-to-date as themselves (Line 22 for **C3**), and sync up replication if falling behind (Line 24). **C3** enforces the election of a candidate that has the most up-to-date log, which ensures Property **P2**.

After syncing up, followers can verify the candidate's rp and associated computation. They use the same calculation scheme by calling Algo. CalcRP with the candidate's ID, where rp and ci should be reproduced (Line 25 to 27). If so, **C4** is satisfied. Then, followers verify the candidate's computational work. They reproduce the hash result and check if the result has a prefix of rp leading zero bytes (Line 28 to 29). If so, **C5** is satisfied, thereby ensuring Property **P3**. Note that followers only hash once ($O(1)$) to verify the computation. Finally, followers send a vote back to the candidate. Formal proofs of **P1**, **P2**, and **P3** can be founded in [51].

The candidate becomes the new leader if it can collect $2f$+1 VoteCPs in time. It then coverts the votes to a `vc_QC` with a threshold of $2f$+1 and declares leadership. During this process, the candidate may find itself out-of-sync if it discovers a leader operating in a higher view; it will abort the election and transition back to a follower (Line 44).

On the other hand, if the candidate neither becomes a leader nor transitions back to a follower when its timer expires, split votes may have occurred, where multiple candidates campaigning for the same view and collecting partial votes (because of **C1**), similar to Raft's split votes [63]. In this case, the candidate transitions back to a redeemer with incremented view V'+1 (Line 31) and starts a new campaign. It is worth noting that this situation is extremely rare, especially with randomized timers (in §4.2.1). Our evaluation in §6 shows that split votes never occurred in 10,000 view changes with just 50ms of randomization.

4.2.4 As a Leader.

After becoming a leader, S_i prepares a new `VcBlock` with the parameters shown in Figure 3. It inherits the old view's `vcBlock` (view V) with its updated rp and ci (Line 50) and broadcasts the new `vcBlock`. **Note that only the elected leader may have a change in its rp in view changes; unsuccessful view change attempts will not result in an rp change.**

```
procedure Receiving(newVcBlock)        ▷ vcBlock sent by the leader
    validate newVcBlock and compare with myVcBlock
    send vcYes ← ⟨newVcBlock.v, σ_{S_i}⟩
    stop myVcBlock.v and start operating in newVcBlock.v as follower
```

The other non-leader servers (i.e., followers, redeemers, and candidates), follow the above steps to verify `newVcBlock`. They validate the QCs and compare the reputation segment of `newVcBlock` with that of `myVcBlock` (i.e., the current `vcBlock` of V). If the only change is the leader's rp and ci, servers adopt `newVcBlock` and send a vcYes message to the leader. When the leader collects $2f$+1 vcYess, the consensus for ① *deciding a new leader* for the new view and ② *updating the new leader's reputation penalty* has been achieved; then, normal operation resumes under the new leadership.

A note on using Proof-of-Work (PoW). In PrestigeBFT, **PoW is never involved in replication.** It is only used as an implementation of the reputation penalty to reduce the probability of electing suspected faulty servers in view changes. Alternatively, Verifiable Delay Functions (VDF) [14] can also be used to delay high penalty servers for participating view changes. Our use of PoW imposes a computational cost on attackers without overburdening correct servers ("let bad guys pay"). Our evaluation shows that the cost for correct servers is negligible (less than 20 ms for rp<5) but becomes significantly higher for attackers (hours for rp>8). While VDF is more environmentally friendly, PoW is an efficient and economic deterrent to malicious attacks.

4.2.5 Refresh penalties. In the partial synchrony model (§ 4.1), when GST is sufficiently long, it may trigger timeouts on non-faulty servers. This may cause non-faulty leaders to get penalized in the long run. PrestigeBFT allows a refresh on imposed rp when at least f+1 non-faulty servers get penalized with their rp exceeding a *threshold* (π). The refresh process for a server S_i is as follows.

1. S_i broadcasts a Ref message: $\langle \text{Ref}, V, \sigma_{S_i} \rangle$.
2. Upon receiving $2f+1$ Refs from different servers (including itself), S_i converts them to a `rs_QC` and set its rp and ci to the initial values. It then broadcasts a Rdone message: $\langle \text{Rdone}, \texttt{rs\_QC}, V, rp, ci, \sigma_{S_i} \rangle$.
3. After receiving a Rdone message from S_i, the other servers verify `rs_QC` and update S_i's rp and ci in the current `VcBlock`.

This refresh mechanism ensures that when a server refreshes its penalty, there have been at least $f+1$ correct servers (in `rs_QC`) whose rp has exceeded π. The refresh sets both rp and ci to their initial values, relieving the imposed potential computational work as well as refreshing the compensation of δ_{tx} and δ_{vc} for future rp calculations.

4.3 The replication protocol

With Properties **P1 P2** and **P3** in the view-change protocol, PrestigeBFT can achieve consensus with optimistic responsiveness [64] using a standard two-phase replication protocol, which are the ordering phase and commit phase. In replication, *servers never respond to a leader that has a lower view.* A replication consensus instance works as follows.

Invoking a consensus service (1 round): A client broadcasts a proposal $\langle \text{Prop}, t, d, c, \sigma_c, tx \rangle$ to all servers, including a unique timestamp (t), a transaction (tx), the digest of the transaction (d), its ID (c), and its signature (σ_c) that signs t, d, and c. It then waits for this proposal to be committed.

Phase 1: constructing `ordering_QC` (2 rounds):

- The leader starts a consensus instance for tx when ① it receives a Prop message from a client, or ② $f+1$ Compt messages from different servers. The leader then assigns a unique sequence number n to Prop and broadcasts an ordering message: $\langle \text{Ord}, \langle \text{Prop} \rangle, n, V, \sigma_{S_i} \rangle$.
- Followers verify the received Ord message by checking that n has not been used. Then, they send a reply to the leader with their signatures.
- The leader waits for $2f+1$ replies and converts them (of size $O(n)$ in total) to a threshold signature (of size $O(1)$), which forms the QC of `ordering_QC`.

Phase 2: constructing `commit_QC` (3 rounds)

- The leader then broadcasts a Cmt message with the obtained QC: $\langle \text{Cmt}, \texttt{ordering\_QC}, V, \sigma_{S_i} \rangle$
- Followers verify `ordering_QC`'s threshold and then send replies to the leader with their signatures.
- The leader waits for $2f$+1 replies to form `commit_QC` and prepares a `txBlock` (shown in Figure 3) by setting the block agreement and transaction fragments accordingly. Then, it broadcasts `txBlock` and sends a Notif message to the client.

Terminating consensus instance (1 round).

- Followers verify the received `txBlock` and then send a Notif message to the client.
- If the client can receive $f+1$ Notifs before its timer expires, it considers tx committed. Otherwise, it complains to the servers (in §4.2).

The replication protocol has a message complexity of $O(n)$ and a time complexity of 7 (rounds). PrestigeBFT achieves optimistic responsiveness (OR) using a two-phase protocol, which is more efficient than HotStuff [78]'s three-phase protocol. HotStuff's additional phase is to sync up non-faulty servers about the commit result, as its passive VC protocol blindly rotates leadership. In contrast, PrestigeBFT's active VC protocol allows servers to elect the most up-to-date candidate, leading to high performance due to the reduced messages and rounds needed (see §6).

5 Correctness argument

This section sketches the correctness arguments including safety and liveness. We denote the assumed computation bound of faulty servers in our system model as γ (§4.1).

Theorem 1 (Validity). *In each consensus instance, if all servers have received the same tx, then any tx committed by a non-faulty server must be that common tx.*

Proof. Each committed tx must have been endorsed by a `commit_QC`. A server that signs in `commit_QC` must have verified a corresponding `ordering_QC`. Since `ordering_QC` is signed by $2f$+1 servers, a committed tx must be the common value that has been seen by at least $2f$+1 servers. □

Lemma 1. *At least $f+1$ non-faulty servers are up-to-date.*

Proof. Lemma 1 is straightforward. Both `vc_QC` in `vcBlocks` and `commit_QC` in `txBlocks` have a threshold of $2f$+1, which can include up to f faulty servers. Therefore, there are at least $f+1$ non-faulty servers included in these QCs and thus are up-to-date in replication in the highest view. □

Lemma 2. *In any given view change, at least $f+1$ non-faulty servers are eligible for being elected as the leader.*

Proof. We denote the set of non-faulty and up-to-date servers in Lemma 1 as $\mathcal{S}_1$, and non-faulty but stale servers as $\mathcal{S}_2$ ($|\mathcal{S}_1| + |\mathcal{S}_2| = 2f+1$). In the worst case, $|\mathcal{S}_1| = f+1$ and

$|\mathcal{S}_2| = f$. Since $\mathcal{S}_2$ can always sync up to a more up-to-date candidate (in §4.2.3), $\forall S_i \in \mathcal{S}_2$ can vote for $\forall S_j \in \mathcal{S}_1$. Thus, $\forall S_j \in \mathcal{S}_1$ are eligible for an election. □

The major difference between the passive and active VC protocols is that faulty servers can actively campaign for leadership and replace a correct leader. In the passive protocol, f faulty servers cannot usurp leadership from a correct leader when they are not the scheduled leader. In the active protocol, all servers can campaign for leadership, which gives faulty servers the opportunity to replace a correct leader. The orchestration of PrestigeBFT's voting-based leader election and reputation mechanisms makes great effort to suppress Byzantine servers from being elected and reduces the possibility of Byzantine leaders over time.

Lemma 3. *Faulty servers cannot repossess leadership indefinitely without making progress in replication.*

Proof. If a faulty server does not make progress in replication, it cannot get compensated as its δ_{tx} remains 0. Thus, its rp keeps increasing. In addition, if a faulty server makes limited replication and stops, after it gets compensated, its $ci = ti$. If it no longer make replication progress, its $\delta_{tx} = 0$ and its rp keeps increasing. When the required computational work exceeds the faulty server's computation capability γ, the faulty server cannot transition to a candidate and thus will never be elected. □

Theorem 2 (Liveness). *After GST, a non-faulty server eventually commits a proposed client request.*

Proof (sketch). In any given time, leadership is in one of the following two conditions: ① leadership is controlled by f faulty servers, or ② leadership is released by f faulty servers.

In ①, with Lemma 3, faulty leaders must at some point start to conduct replication. Otherwise, they cannot control the leadership indefinitely. When they start to conduct replication, they become temporary non-faulty leaders.

In ②, with Lemma 2, a leader will eventually be elected from up-to-date and non-faulty servers. Thus, after GST, a client request will eventually be committed by all non-faulty servers. Therefore, in both cases, PrestigeBFT ensures that a client eventually receives replies to its request after GST. □

Theorem 3 (Safety). *Non-faulty servers do not decide on conflicting blocks. That is, non-faulty servers do not commit two* `txBlock`*s at the same sequence number n.*

Proof (sketch). With Property **P1**, no view has more than one leader. Next, we prove this theorem by contradiction. We use the partition of servers in Lemma 2 and denote faulty servers as $\mathcal{S}_f$. We claim there are `txBlock` and `txBlock`$_\diamond$, both committed with sequence number n.

In this case, `commit_QC` and `commit_QC`$_\diamond$ are both signed by $2f+1$ servers. Say `commit_QC` is signed by servers in $\mathcal{S}_1 \cup \mathcal{S}_f$. Then, servers in $\mathcal{S}_1$ cannot sign `commit_QC`$_\diamond$ with n. Although faulty servers in $\mathcal{S}_f$ can double commit, `commit_QC`$_\diamond$ can only find servers in $\mathcal{S}_f \cup \mathcal{S}_2$ ($|\mathcal{S}_f|$+$|\mathcal{S}_2|$=$2f$) to sign it, which is not sufficient to form a QC of size $2f + 1$. Therefore, `commit_QC`$_\diamond$ cannot be formed, which contradicts our claim. □

Due to space limitations, here we provided only proof sketches and refer reader to the extended version of our paper [51]. We strongly encourage readers to consult with the extended version, where we provide the complete proofs, visualized analysis, various examples, and collected Q&A to aid understanding.

6 Evaluation

We implemented PrestigeBFT in Golang and deployed it on 4, 16, 31, 61, and 100 VM instances on a popular cloud platform. Each instance includes a machine with 4 vCPUs supported by 2.40 GHz Intel Core processors (Skylake) with a cache size of 16 MB, 15 GB of RAM, and 75 GB of disk space running on Ubuntu 18.04.1 LTS. The TCP/IP bandwidth measured by `iperf` is around 400 MB/s with a raw network latency between two instances less than 2 ms. We use the following notations to report on the results.

- n The number of VMs (scales)
- β The number of transactions in a batch (batch size)
- d Emulated network delays (ms) using `Netem`
- m The message size (e.g., m = 32 bytes)

6.1 Performance under normal operation

Performance is reported in terms of throughput and latency. Since linear BFT algorithms have shown significant performance advantage compared to non-linear algorithms (e.g., HotStuff [78] ($O(n)$) > PBFT [18]($O(n^2)$) > RBFT [9]($O(n^3)$)), we conducted end-to-end comparisons for PrestigeBFT (`pb`) against three linear BFT algorithms: SBFT [23] (`sb`), HotStuff [42] (`hs`), and Prosecutor [80] (`pr`) using their original implementations. Throughput was measured in transactions per second (TPS) on servers, i.e., the number of requests committed in one second. Latency was measured on clients from when a request is sent to when f+1 NOTIFs are received.

Peak performance. The peak performance was measured when n=4. Clients generated random requests of m=32 bytes and waited for one request to complete before sending the next one. We kept increasing batch sizes for each algorithm to find the batch size that resulted in the highest throughput while maintaining a low latency. Under each batch size, we kept deploying more clients until their generated workloads were sufficient (until an elbow of a curve occurs (Figure 6)).

PrestigeBFT outperforms its baselines with a peak performance at a throughput of 186,012 TPS and a latency of 166 ms at β=3000. Its peak performance is 5.2× higher than HotStuff's, which peaked at 35,428 TPS in 129 ms at β=1000. The high performance is attributed to the reduced phases in

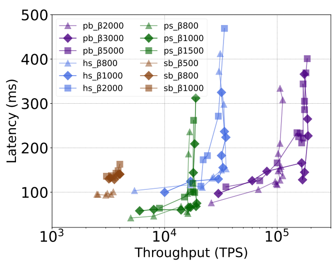


Figure 6. Performance under batching (n=4 and m=32).

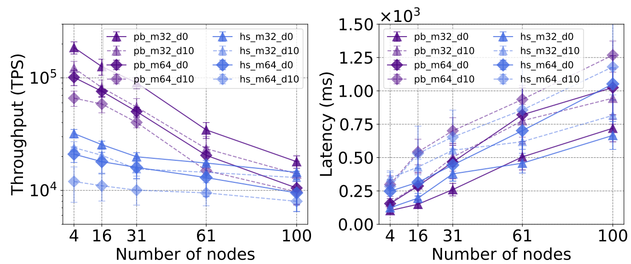


Figure 7. Throughput (left) and latency (right) under increasing system scales (m = 32 and 64, d = 0 and 10 ± 5 ms).

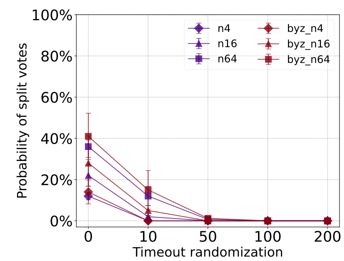


Figure 8. Split votes under different timeout randomization.

replication while achieving optimistic responsiveness. Prosecutor performed at a similar throughput compared to HotStuff with a lower latency, and SBFT peaks at 4, 872 TPS in 148 ms at β=800 (similar results are reproduced by [81]).

Scalability. We evaluated the performance of the two best-performing algorithms (i.e., pb and hs) at increasing system scales with two workloads (m=32 and 64 bytes), choosing their best batch sizes when n=4 (i.e., β=1000 for hs and β=3000 for pb). In addition to the raw network latency (d=0), we implemented additional network delays of d=10±5 ms at normal distribution using `netem` to emulate a higher network latency. The results show that the throughput of both algorithms decreases while their latencies increase with cluster sizes (shown in Figure 7). Under the emulated network delay, consensus latency increased significantly as delayed messaging prolongs packing requests into batches and also results in a high variance.

6.2 Performance under failures

We also conducted experiments to evaluate PrestigeBFT's performance under failures. Since we cannot simulate all types of Byzantine failures, we have considered the following four common attacks.

- **F1** (Timeout attacks) Faulty servers set their timeouts to the same of f randomly picked correct servers.
- **F2** (Quiet participants) Faulty servers do not respond to any request (similar to crash/send omission failures).
- **F3** (Equivocation) Faulty servers reply to a quest by sending back erroneous messages.
- **F4** (Repeated view-change attacks) Faulty servers campaign for leadership when they are not the leader.

Split votes under F1. The nature of active view changes allows servers to campaign for leadership. In theory, multiple servers can become candidates simultaneously and may cause split votes (discussed in §4.2.3) prolonging an election. Nevertheless, for this scenario to occur, competing candidates must detect a leader's failure + finish their computations + arrive their campaign requests at other servers in the same period, which is extremely rare when correct servers randomize their timeouts.

We accessed 10, 000 view changes under n=4, 16, and 64 with increasing amount of randomization (ϵ) and set timeouts from [800, 800+ϵ ms]. We observe that randomization significantly reduced the occurrence of split votes (shown in Figure 8). Without failures, split votes stopped occurring with just ϵ=50 ms while timeout attacks (**F1**) only slightly increased the occurrence and could not inflict split votes when ϵ>100 ms.

Performance under F2 and F3. HotStuff uses a passive view-change protocol inherited from PBFT [18], which is also used by numerous leader-based BFT algorithms, such as Zyzzyva [47], SBFT [38], Aardvark [22], and RBFT [9]. This makes HotStuff an ideal baseline for comparing the performance of passive protocols with PrestigeBFT's active protocol. The trends in performance changes can provide insights into the effectiveness of the passive vs. active comparison.

To show the performance under frequent view changes, we implemented a simple timing policy; i.e., each server starts a view change every x time in a view. Thus, a server in both algorithms triggers a view change if ① a leader failure is reported, or ② the current view has operated for x time. We set x=10 (more frequent, higher decentralization) and 30 seconds (less frequent, lower decentralization), denoted by (`r10`) and (`r30`), respectively. We set HotStuff's initial timeout to 1 second and our timeouts from [800, 1200 ms].

We arbitrarily chose f=1 and f = 1, 3, 5 servers when n=4 and n=16 to perform **F2** and **F3** and measured the throughput (shown in Figure 9). We ran each algorithm for 20 min[1]. Generally, **F3** caused a higher drop in throughput than **F2**, as it sends erroneous messages consuming the bandwidth.

HotStuff encountered f faulty servers being assigned as a leader by its passive VC protocol. Each faulty leader cost the system around 1.2 s (1 s timeout + 200 ms voting and switching leaders) with no transactions replicated. When n=4, HotStuff's throughput dropped by nearly 62%, from 32, 234 TPS (f=0) to 12, 051 TPS (f=1) at `hs_r30_quiet`. Additionally, more frequent rotations resulted in a higher decrease in throughput under both **F2** and **F3**.

[1]Under n = 16 and `r/30S`, to rotate leadership in a full circle in the passive view-change protocol, it requires at least 30 s × 16 = 8 min.

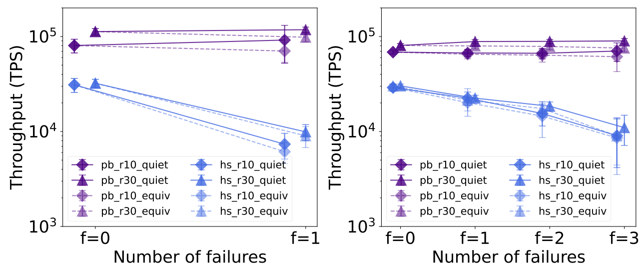


Figure 9. Throughput comparisons under quiet and equivocation attacks (F2+F3) in n=4 (left) and n=16 (right).

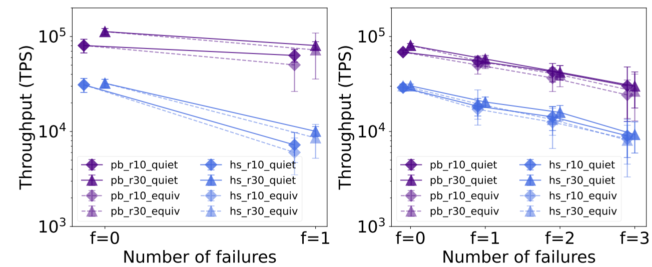


Figure 10. Throughput comparisons under repeated VC attacks (F4+F2 and F4+F3) in n=4 (left) and n=16 (right).

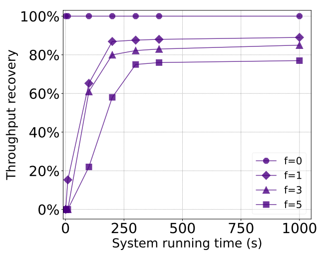


Figure 11. Improving TPS under F4+F2 in pb_r10_quiet.

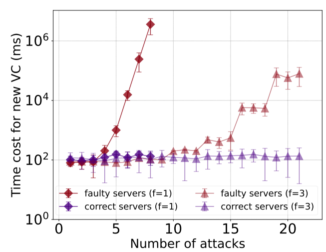


Figure 12. Time costs to start a view change under attacks.

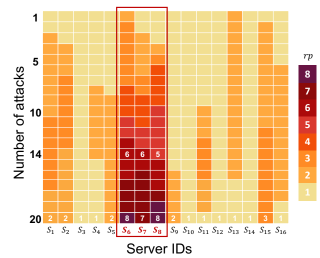


Figure 13. The change of server rp under f=3 in Fig. 12.

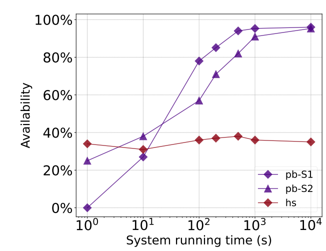


Figure 14. Availability under different types of attacks.

In contrast, PrestigeBFT's throughput was nearly unaffected. Interestingly, with more quiet servers (F2), its throughput saw an increase (e.g., throughput had a 14% from 80, 418 TPS under f=0 to 91, 765 TPS under f=1 at vb_r10_quiet). Since f quiet servers do not consume network bandwidth, more transactions can be exchanged by correct servers.

Performance under F4+F2 and F4+F3. We also evaluated the behavior of repeated view change attacks, which is the most detrimental to PrestigeBFT's active VC protocol. We arbitrarily chose f=1 when n=4 and f=1, 3, 5 when n=16 faulty servers to perform the two combined attacks: when a faulty server becomes a leader, it becomes quiet (F4+F2) or perform equivocation (F4+ F3). We allow faulty servers to collude to launch attacks when f>1 by performing joint computation and sharing logs.

As shown in Figure 10, HotStuff was hit with a similar sustained drop in throughput as in Figure 9. Its throughput saw a drop of 69% under n=4, from 32, 234 TPS (f=0) to 10, 051 TPS (f=1) at hs_r10_quiet. Since HotStuff follows a predefined schedule to rotate leadership, faulty servers cannot be selected when they are not scheduled despite they sent view change requests. Compared to the result in Figure 9, throughput drops slightly higher because of the erroneous messages faulty servers kept sending.

PrestigeBFT witnessed a moderate drop in overall throughput. Under n=4, its throughput dropped by 24%, from 80, 418 TPS (f=0) to 61, 208 TPS (f=1) at vb_r10_quiet. In addition, PrestigeBFT showed an improving performance over the experiment, as faulty servers are constantly penalized after launching attacks without replication. When faulty servers struggled to launch new attacks, PrestigeBFT gained failure-free views with correct leaders conducting replication. Figure 11 shows the trend of the improving throughput: at the beginning of the experiment, PrestigeBFT suffered from the attacks and could not make progress in replication (in the first 10s). However, after being repeatedly penalized by the reputation mechanism, faulty servers were quickly suppressed in view changes while correct servers regained leadership and resumed normal operation (staring from 100s). At time 1000s, PrestigeBFT's throughput recovered to 87% of its throughput under normal operation ($f = 0$).

We show the time costs for faulty servers launching repeated VC attacks (F4+F2). Our implementation uses SHA-256 as the hashing algorithm; thus, the probability of finding a hash that has a prefix of rp leading 0s is as follows.

$$Pr(rp) = \frac{2^{256-8rp}}{2^{256}} = 2^{-8rp}$$

$Pr(rp)$ requires exponentially increasing computation to find a required hash result, which resulted in the skyrocketing time cost for attackers (shown in Figure 12).

We show the change of server rps in Figure 13 throughout the attacks in Figure 12 when f=3, where S_6, S_7, and S_8 are the three faulty servers. After faulty servers' rp exceeded 5, they began to struggle to perform the required hash computation (Figure 12) and cannot prevent a correct leader from conducting replication. At this time, correct servers start to regain leadership and apply compensation with reduced rp (from the 14th attack in Figure 13). When the rp of faulty servers increased to 8, they were unable to launch new attacks and could not become a leader in future view changes.

Availability. In addition, faulty servers in PrestigeBFT can "smartly" launch repeated VC attacks: they can calculate their rp and launch attacks only when they can get compensated. We name this strategy as S2 and the previous strategy, where faulty servers launch attacks whenever they are not the leader, as S1. We kept both PrestigeBFT and HotStuff (f=3) running for 10^4s and reports their availability in Figure 14. To pursue S2, faulty servers must temporarily behave correctly and allow for replication, thereby giving PrestigeBFT failure-free time. With more transactions replicated, ti in Eq. 2 keeps increasing, and faulty servers must behave correctly increasingly longer to continuously get compensated. Overall, PrestigeBFT exhibits a significantly higher availability when faulty servers are penalized.

6.3 Summary of results

The evaluation results show that ① PrestigeBFT achieves high performance in replication in terms of throughput and latency. ② PrestigeBFT's performance is unaffected by faulty servers being quiet or equivocating, remaining at a high throughput. ③ Under repeated VC attacks, PrestigeBFT's reputation mechanism quickly suppresses faulty servers with improving performance and availability over time.

7 Related work

Consensus algorithms provide safety and liveness for state machine replication (SMR) [67] under different failure assumptions. With increasing software scales, Byzantine failures are becoming more common due to hardware glitches, operator errors, and worldwide anonymous collaboration, especially in blockchains where participants may intentionally break the protocol to gain more profit [26, 43, 50, 81].

Leader-based BFT algorithms have been favored by permissioned blockchains, such as HyperLedger Fabric [5] and Diem [28]. After PBFT [18] pioneered a practical BFT solution with an $O(n^2)$ message complexity using public-key signatures, numerous approaches have been proposed for optimizations in terms of replication and view changes.

Replication optimizations have been focusing on various aspects. They use speculative decisions and reduce workloads for a single leader [31, 39, 47], develop high performance implementations [13, 15, 34, 37, 61, 65, 66, 71], use sharding mechanisms to improve throughput [4], reduce messaging costs [30, 53, 54, 70, 77], offer confidentiality protection dealing with secret sharing [76], apply accountability for individual participants [21, 60, 68], limit faulty behavior using trusted hardware [11, 20, 44, 49], and utilize threshold signatures [52, 69] to achieve linear message complexity [38, 78, 80].

In addition, DAG-based protocols have achieved high performance by separating transaction distribution from consensus [27, 45, 72]. PrestigeBFT's view change protocol can also be applied in DAG for efficiently selecting leaders as well as in transaction pipelining [10, 35].

View changes detect leader failures and move the system to new views [2]. PBFT [18] developed a passive view-change mechanism where servers follow a predefined leader schedule to rotate leaders. Because of its simplicity, this mechanism has been adopted by numerous BFT algorithms [1, 9, 13, 15, 22, 29–31, 34, 37–39, 41, 47, 78]. Aardvark [22] imposes regular view changes when a leader slows down by a certain threshold, and HotStuff [78] suggests that views be rotated for each request. However, the frequent passive view changes will result in frequent faulty new leaders.

Reputation approaches with history. Learning from the past to predict the future is a common design philosophy in Computer Science, such as the multi-level feedback queue in CPU scheduling [24, 55], hardware branch predictors [56, 79], and caching algorithms [19, 62]. Reputation-based BFT algorithms also incorporate historic information and adaptively predict correctness. For example. DiemBFT [29] calculates reputation by tracking active servers' total log length. However, DiemBFT still uses a passive view-change protocol; it restricts the use of reputation only when correct servers are rotated as leaders. In contrast, PrestigeBFT takes a more proactive approach, fully utilizing reputation with an active view-change protocol to enhance system efficiency and robustness against failures.

Leaderless BFT algorithms do not use a designated leader to conduct replication, thereby mitigating the problem of single points of failures and single server bottlenecks [25, 32, 48, 57, 75]. Without a leader, leaderless BFT algorithms often utilize binary Byzantine agreement [58] to jointly form quorum certificates [12]. Compared with leader-based BFT algorithms, they are more robust and avoid leadership changes, but often suffer from high message and time costs for conflict resolutions, even after applying erasure coding (e.g., AVID broadcast [16]).

8 Conclusions

This paper introduces PrestigeBFT, a leader-based BFT consensus algorithm that enables active view changes with reputation mechanisms. The reputation mechanism learns from a server's history and ranks the server's correctness with a reputation penalty. The active view-change protocol allows servers to proactively campaign for leadership by performing reputation-determined work. Consequently, servers with good reputations are more likely to be elected as new leaders than servers with bad reputations. Our evaluation results show that PrestigeBFT is robust and efficient. It achieves 5× higher throughput than its best-performing baseline under normal operation. It exhibits robustness under failures: while its baseline suffered from a 69% drop in throughput under a variety of Byzantine failures, PrestigeBFT witnessed only a 24% drop with a vigorous recovery.

References

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync HotStuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 106–118.

[2] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. 2005. BAR fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 45–58.

[3] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. 2008. Byzantine replication under attack. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*. IEEE Computer Society, 197–206. https://doi.org/10.1109/DSN.2008.4630088

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*. 76–88.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30.

[6] Mina Aryal. 2019. Top 9 Causes Of A Server Down Or Server Crash. https://ictframe.com/top-9-causes-of-a-server-down-or-server-crash/

[7] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1994. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)* 41, 1 (1994), 122–152.

[8] Hagit Attiya and Jennifer Welch. 2004. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons.

[9] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 297–306.

[10] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* 34 (2016).

[11] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*. 222–237.

[12] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. 183–192.

[13] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. 2014. State machine replication for the masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 355–362.

[14] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I*. Springer, 757–788.

[15] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph. D. Dissertation.

[16] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 191–201.

[17] Christian Cachin and Marko Vukolić. 2017. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873* (2017).

[18] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[19] Marek Chrobak and John Noga. 1999. LRU is better than FIFO. *Algorithmica* 23, 2 (1999), 180–185.

[20] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 189–204.

[21] Pierre Civit, Seth Gilbert, and Vincent Gramoli. 2021. Polygraph: Accountable byzantine agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 403–413.

[22] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults.. In *NSDI*, Vol. 9. 153–168.

[23] Concord-bft. 2019. Concord-BFT: a Distributed Trust Infrastructure. https://github.com/vmware/concord-bft

[24] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. 1962. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*. 335–344.

[25] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. 2018. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 1–8.

[26] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 910–927.

[27] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.

[28] Diem. 2020. The Diem blockchain. https://developers.diem.com.

[29] Diem. 2021. DiemBFT v4: State Machine Replication in the Diem Blockchain. https://developers.diem.com/docs/technical-papers/state-machine-replication-paper.

[30] Tobias Distler and Rüdiger Kapitza. 2011. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the sixth conference on Computer systems*. 91–106.

[31] Sisi Duan, Sean Peisert, and Karl N Levitt. 2014. hBFT: speculative Byzantine fault tolerance with minimum cost. *IEEE Transactions on Dependable and Secure Computing* 12, 1 (2014), 58–70.

[32] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2028–2041.

[33] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[34] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1597–1609.

[35] Bryan Ford. 2019. Threshold logical clocks for asynchronous distributed coordination and consensus. *arXiv preprint arXiv:1907.07010* (2019).

[36] David K Gifford. 1979. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*. 150–162.

[37] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer systems*. 363–376.

[38] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.

[39] Lachlan J Gunn, Jian Liu, Bruno Vavala, and N Asokan. 2019. Making speculative BFT resilient with trusted monotonic counters. In *2019 38th*

Symposium on Reliable Distributed Systems (SRDS). IEEE, 133–13309.

[40] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, et al. 2013. Failure recovery: When the cure is worse than the disease. In *HotOS*.

[41] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment* 13, 6 (2020), 868–883.

[42] HotStuff. 2019. Libhotstuff: A general-puropse BFT state machine replication library with modularity and simplicity. https://github.com/hot-stuff/libhotstuff

[43] Yuming Huang, Jing Tang, Qianhao Cong, Andrew Lim, and Jianliang Xu. 2021. Do the rich get richer? Fairness analysis for blockchain incentives. In *Proceedings of the 2021 International Conference on Management of Data*. 790–803.

[44] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*. 295–308.

[45] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 165–175.

[46] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 583–598.

[47] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[48] Leslie Lamport. 2011. Brief announcement: Leaderless Byzantine paxos. In *International Symposium on Distributed Computing*. Springer, 141–142.

[49] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems.. In *NSDI*, Vol. 9. 1–14.

[50] Michael Lewis. 2014. *Flash boys: a Wall Street revolt.* WW Norton & Company.

[51] PrestigeBFT Lib. 2023. https://anonymous.4open.science/r/prestigebft-1FE0/

[52] Benoît Libert, Marc Joye, and Moti Yung. 2016. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science* 645 (2016), 1–24.

[53] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. 2016. XFT: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 485–500.

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

[55] Richard McDougall and Jim Mauro. 2006. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (paperback)*. Pearson Education.

[56] Scott McFarling. 1993. *Combining branch predictors*. Technical Report. Citeseer.

[57] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 31–42.

[58] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. 2014. Signature-free asynchronous Byzantine consensus with t< n/3 and O (n2) messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 2–9.

[59] Satoshi Nakamoto. 2019. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report. Manubot.

[60] Joachim Neu, Ertem Nusret Tas, and David Tse. 2021. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 446–465.

[61] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. 2020. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 583–598.

[62] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[63] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 305–319.

[64] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with optimistic instant confirmation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–33.

[65] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. FalconDB: Blockchain-based collaborative database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 637–652.

[66] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. 2020. Blockene: A high-throughput blockchain over mobile devices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 567–582.

[67] Fred B Schneider. 1990. The state machine approach: A tutorial. *Fault-tolerant distributed computing* (1990), 18–41.

[68] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, et al. 2022. IA-CCF: Individual Accountability for Permissioned Ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 467–491.

[69] Victor Shoup. 2000. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 207–220.

[70] Yee Jiun Song and Robbert van Renesse. 2008. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*. Springer, 438–450.

[71] Joao Sousa, Alysson Bessani, and Marko Vukolic. 2018. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 51–58.

[72] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag BFT protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.

[73] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-bft: High-throughput BFT for blockchains. *arXiv preprint arXiv:1906.05552* (2019), 92.

[74] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 17–33.

[75] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.

[76] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. 2022. COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1528–1528.

[77] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. 2021. DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks. *arXiv preprint arXiv:2110.04371* (2021).

[78] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.* ACM, 347–356.

[79] Cliff Young and Michael D Smith. 1994. Improving the accuracy of static branch prediction using branch correlation. *ACM SIGOPS Operating Systems Review* 28, 5 (1994), 232–241.

[80] Gengrui Zhang and Hans-Arno Jacobsen. 2021. Prosecutor: An efficient BFT consensus algorithm with behavior-aware penalization against Byzantine attacks. In *Proceedings of the 22nd International Middleware Conference.* 52–63.

[81] Yunhao Zhang, Srinath T. V. Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 633–649.

A Correctness Argument

In this section, we show the correctness argument of PrestigeBFT's view-change protocol and prove its safety and liveness. We continue to use the partition of servers as in §5, where $n = 3f + 1$ servers are divided into three sets: $\mathcal{S}_1$ ($|\mathcal{S}_1| = f + 1$) and $\mathcal{S}_2$ ($|\mathcal{S}_2| = f$) are non-faulty, and $\mathcal{S}_f$ ($|\mathcal{S}_f| = f$) are faulty servers.

A.1 View change correctness

We first show the correctness argument of the client interaction of the view-change protocol that attains two key correctness properties:

1. Under a correct leader, no view change will be initiated (leadership robustness).
2. Under a faulty leader, a view change must be initiated when the faulty leader cannot achieve consensus for client requests (leadership completeness).

To prove the above two key properties, we first define two types of view changes as follows.

Definition 1 (Unnecessary view changes)**.** *Under a non-faulty leader, any view change initiated by other non-faulty servers is an unnecessary view change.*

Definition 2 (Necessary view changes)**.** *Under a faulty leader, a view change initiated by a non-faulty server is necessary.*

PrestigeBFT requires a non-faulty client to broadcast its complaint to all servers. As such, all non-faulty servers ($\mathcal{S}_1 \cup \mathcal{S}_2$ where $|\mathcal{S}_1 \cup \mathcal{S}_2| = 2f + 1$) are able to receive the complaint (illustrated in Figure 15a) and start the procedure of handling a client complaint (Line 1). Next, we show that faulty clients cannot trigger an unnecessary view change (Definition. 1) with or without faulty servers.

Lemma 4. *Under a non-faulty leader, faulty clients and non-faulty servers cannot trigger a view change.*

Proof. Lemma 4 is straightforward. A faulty client can behave in one of two ways: ① it sends no complaint, or ② it sends its complaint to at least one server (illustrated in Figure 15b).

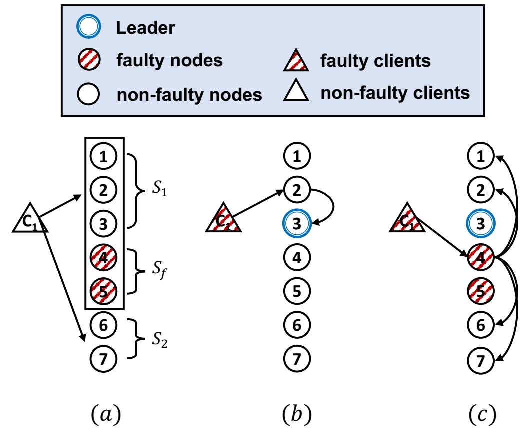


Figure 15. Leadership robustness analysis in PrestigeBFT's active view-change protocol.

Scenario ① simply does not affect our system, which can be disregarded. In Scenario ②, when a non-faulty server receives a complaint, it relays it to the leader (Line 2). Then, the leader will achieve the consensus for the transaction piggybacked in the complaint. Note that since our failure assumption does not assume DDOS attacks, non-faulty servers are able to handle every proposed request; e.g., if a faulty client sends different transactions to different servers, non-faulty servers will relay every complaint, and the leader will receive them in time. Therefore, all non-faulty servers will terminate this procedure (Line 5) with no view change triggered. □

Next, we show that colluding faulty clients and faulty servers cannot trigger a view change when the current leader is non-faulty.

Lemma 5. *Under a non-faulty leader, faulty clients and faulty servers cannot trigger a view change.*

Proof. A faulty client in Lemma 5 can behave in one of two ways: ① it sends its complaint to at least a non-faulty server, and ② it does not send its complaint to any non-faulty servers. In Scenario ①, with Lemma 4, any non-faulty server ($\forall S_i \in \mathcal{S}_1 \cup \mathcal{S}_2$) that receives a complaint will terminate the procedure under a correct leader. In Scenario ②, the procedure will not be invoked on non-faulty servers.

In the worst case, all faulty servers collude and try to invoke a view change. To make a successful candidate, they have to construct a `conf_QC` of size $f + 1$ (illustrated in Figure 15c). However, since all non-faulty servers will terminate the procedure, they cannot collect a ReVC from a non-faulty server; i.e., no server in $\forall S_i \in \mathcal{S}_1 \cup \mathcal{S}_2$ will be included in `conf_QC`. Consequently, even if a faulty server becomes a candidate, non-faulty servers will not vote for it according to **C2**, and thus a faulty candidate cannot be elected.

Therefore, under a non-faulty leader, no view change will be triggered under f faulty servers and unlimited faulty clients. □

Theorem 4 (Leadership robustness)**.** *In any given view, under a non-faulty leader, no view change will be initiated.*

Proof. Without failures, PrestigeBFT operates under the replication protocol, and the view-change protocol will not be invoked. With faulty clients and servers, Lemma 4 and 5 have shown that no view change will be invoked under either condition. Therefore, under a non-faulty leader, no view change will be invoked, which proves this theorem. □

Theorem 4 is critical for system availability. It shows that **PrestigeBFT's active view change protocol will have a stable view under a correct leader regardless of the behavior of faulty clients, faulty servers, and their collusion**.

In addition, with Theorem 4, faulty servers can only intervene in the view-change process when the current leader becomes faulty or a view change is invoked by policy-defined criteria, such as timing policies and throughput-threshold policies (discussed in §4.2.1). Next, we show that the interference of faulty servers cannot prevent view changes from being initiated (Definition 2).

Lemma 6. *A faulty leader cannot prevent a necessary view change for a higher view.*

Proof. When a faulty leader stops committing a non-faulty client's transaction, all non-faulty servers ($\mathcal{S}_1 \cup \mathcal{S}_2$) will receive a complaint from the client. Then, at least a non-faulty server S_i ($S_i \in \mathcal{S}_1 \cup \mathcal{S}_2$) will broadcast a ConfVC message. In this case, S_i can receive at least $f + 1$ RecV replies from $\mathcal{S}_1 \cup \mathcal{S}_2$ and construct a conf_VC (Line 9) regardless of servers in $\mathcal{S}_f$, starting a new view change with an incremented view. Thus, a faulty leader cannot prevent a necessary view change for a higher view. □

With Lemma 6, a faulty leader cannot prevent a necessary view change from being initiated by non-faulty servers. However, due to the nature of active view changes, other faulty servers can compete with non-faulty servers in the initiated view change; they may win the election and repossess the leadership. Next, we show the completeness of leadership in PrestigeBFT; that is, faulty servers cannot indefinitely prevent the election of a non-faulty leader.

Theorem 5 (Leadership completeness)**.** *In a given view V, faulty servers cannot indefinitely prevent a non-faulty leader from being elected in a higher view V' ($V' > V$).*

Proof. Faulty servers can delay the appearance of a non-faulty leader by repossessing leadership. With Lemma 3 in §5, faulty servers cannot repossess leadership indefinitely without making progress in replication. Thus, faulty servers can behave in one of two ways: ① they launch attacks with the rise of their reputation penalties, or ② they launch attacks only when they can remain their reputation penalties unchanged by receiving compensation.

Pursuing ①, after faulty servers exhaust their computation capability, they can no longer be elected as future leaders. Pursuing ②, in order to get compensated, faulty servers must temporarily give up leadership to non-faulty servers (δ_{vc}) or behave temporarily correctly in replication (δ_{tx}). Therefore, in both ways, faulty servers cannot indefinitely prevent a non-faulty leader from being elected in a higher view. □

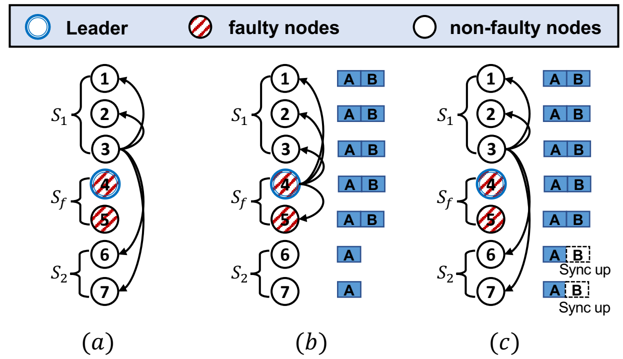


Figure 16. Examples of replication and view changes under a faulty leader.

A.2 Liveness

Next, we prove liveness. We first prove the three properties of PrestigeBFT's view-change protocol (discussed in §4.2).

Lemma 7 (Property **P1**)**.** *At most one leader can be elected in a given view.*

Proof. We prove this Lemma by contradiction. We claim that there are two legitimate leaders S_i and S_j in a given view V. For this claim to be true, S_i and S_j must both have constructed their vc_QCs of size $2f + 1$ when they were candidates, denoted by $\texttt{vc\_QC}_{S_i}$ and $\texttt{vc\_QC}_{S_j}$, respectively.

In the worst case, $\texttt{vc\_QC}_{S_i}$ is constructed by $\mathcal{S}_1 \cup \mathcal{S}_f$. Since each server votes only once in a view (**C1**), $\mathcal{S}_1$ will not vote for another server in view V. However, $\texttt{vc\_QC}_{S_j}$ must also have a size of $2f + 1$; it can be composed of $\mathcal{S}_f \cup \mathcal{S}_2$, as $\mathcal{S}_f$ are faulty servers. Nevertheless, since $|\mathcal{S}_f \cup \mathcal{S}_2| = 2f$, it must contain at least one server S_k such that $S_k \in \mathcal{S}_1$. However, no server in $\mathcal{S}_1$ will vote again in view V; thus, $\texttt{vc\_QC}_{S_j}$ cannot be formed, which contradicts our claim and proves this Lemma. □

Lemma 8 (Property **P2**)**.** *An elected non-faulty leader has the most up-to-date replication.*

Proof. The replication protocol (discussed in §4.3) requires that a txBlock be committed with a commit_QC of size $2f+1$. In the worst case, under a faulty leader, the commit_QC is formed by $\mathcal{S}_1 \cup \mathcal{S}_f$ (illustrated in Figure 16b). Similarly, the view-change protocol requires that a vcBlock be committed with a vc_QC of size $2f+1$. Therefore, at least $f+1$ non-faulty servers have the most up-to-date logs, including txBlocks in replication and vcBlocks in view changes. According to **C3**, a non-faulty leader will be elected at least from the $f+1$ most up-to-date servers, which proves this Lemma. ◻

Lemma 9 (Property **P3**)**.** *An elected leader's reputation penalty and the correspondingly performed computation can be verified by all non-faulty servers.*

Proof. Since vcBlocks are the result of view-change consensus, they are replicated among at least $2f+1$ servers. For the sake of simplicity, we assume that all up-to-date non-faulty servers are in $\mathcal{S}_1$ and stale non-faulty servers are in $\mathcal{S}_2$. In the worst case, vcBlocks are replicated among $\mathcal{S}_1 \cup \mathcal{S}_f$.

From Lemma 8, a leader is elected among up-to-date servers (i.e., $\mathcal{S}_1$). When servers in $\mathcal{S}_2$ receive a VoteCP from a candidate from $\mathcal{S}_1$ (illustrated in Figure 16c), they can verify any more advanced txBlocks and vcBlocks by checking their *QC*s. Thus, they will sync to up-to-date (Line 20 to 24), obtaining logs as least as up-to-date as the candidate. After the sync up, $\mathcal{S}_2$ invokes Algo. CalcRP using the same input as the candidate. Therefore, the candidate's rp can be reproduced, which can be used to verify its corresponding hash computation result. ◻

We have shown that PrestigeBFT's view-change protocol guarantees the election of an up-to-date leader. Next, we show that it also guarantees that stale servers will not be penalized in unsuccessful elections.

Lemma 10. *The reputation penalties of non-faulty but stale servers will not be increased in view changes.*

Proof. Although a stale server's leader election will not be successful, the stale server can still invoke the view-change protocol and transition to the candidate state. It will not receive sufficient votes because up-to-date servers in $\mathcal{S}_1$ will never vote for it. In this case, its calculated reputation penalty will not be recorded in the vcBlock of the new view. Note that in each view change, only the elected leader's reputation penalty and compensation index are updated (discussed in §4.2.4). Therefore, unsuccessful attempts of leader election will not change a server's reputation penalty. ◻

Now we show the proof of liveness; we repeat the theorem of liveness below:

Theorem 6 (Liveness)**.** *(Same as Theorem 2) After GST, a non-faulty server eventually commits a proposed client request.*

Proof. At any given time, leadership is in one of the following two conditions: ① leadership is controlled by f faulty servers, or ② leadership is released by f faulty servers.

In ①, with Lemma 3, faulty leaders must at some point start to conduct replication. Otherwise, they cannot control the leadership indefinitely. When they start to conduct replication, they become temporary non-faulty leaders.

In ②, with Lemma 2, a leader (e.g., S_i) will eventually be elected from up-to-date and non-faulty servers (i.e., $S_i \in \mathcal{S}_1$). With Lemma 10, the reputation penalties of stable servers do not increase in view changes. Note that in case of rising reputation penalties incurred by GST, all non-faulty servers can apply refresh penalties introduced in §4.2.5.

In addition, we show that all non-faulty servers are able to move to a new view. Assume a server S_i in a view V, which has one of three possible scenarios in a view change: ① S_i initiates a leader election campaign for a higher view V' ($V' > V$) and is elected as the new leader; ② S_i initiates a leader election campaign for a higher view V' ($V' > V$) but is not elected as the new leader; and ③ S_i does not initiate a leader election campaign and remained as a follower in view V.

Scenario ① is straightforward. When S_i is elected, S_i moves to the new view it initiated. In Scenario ②, if S_i did not win an election, it can be in the redeemer state or the candidate state. In both states, its operating view is V (from the current vcBlock of view V), and the view it is campaigning for is V' ($V' > V$). Once S_i receives a legit vcBlock of view V^* ($V^* > V$), it aborts its campaign activity for view V', moving to view V^* by transitioning back to the follower state in accordance with the procedure of receiving a new vcBlock defined in §4.2.4. In Scenario ③, S_i simply follows the same procedure moving to the new view when it receives a vcBlock of a higher view.

To conclude, after GST, all servers are able to move to a new view, and a non-faulty leader is eventually elected in the new view. Therefore, a client request will eventually be committed by all non-faulty servers; i.e., PrestigeBFT ensures that a client eventually receives replies to its request after GST. ◻

A.3 Safety

After a leader is elected in a view, PrestigeBFT uses a standard two-phase replication protocol to conduct consensus for transactions proposed by clients. We now prove that PrestigeBFT ensures safety.

Theorem 7 (Safety)**.** *(Same as Theorem 3) Non-faulty servers do not decide on conflicting blocks. That is, non-faulty servers do not commit two* txBlock*s at the same sequence number n.*

Proof. With Lemma 2, PrestigeBFT ensures that each view has at most one leader. When a view has a non-faulty leader, the replication protocol is invoked to conduct consensus for transactions proposed by clients and produces the consensus result as a txBlock with a unique sequence number n. Note that PrestigeBFT does not allow the consensus process of a txBlock to operate across views, as servers never respond to a message from a lower view (discussed in §4.3). The `ordering_QC` and `commit_QC` must be constructed in the same view. We claim that there are `txBlock` and $\texttt{txBlock}_\diamond$, both committed with sequence number n.

In this case, `commit_QC` and $\texttt{commit\_QC}_\diamond$ are both signed by $2f+1$ servers. Say `commit_QC` is signed by servers in $\mathcal{S}_1 \cup \mathcal{S}_f$. Then, servers in $\mathcal{S}_1$ cannot sign $\texttt{commit\_QC}_\diamond$ with n. Although faulty servers in $\mathcal{S}_f$ can double commit, $\texttt{commit\_QC}_\diamond$ can only find servers in $\mathcal{S}_f \cup \mathcal{S}_2$ ($|\mathcal{S}_f|+|\mathcal{S}_2|=2f$) to sign it, which is not sufficient to form a QC of size $2f+1$. Therefore, $\texttt{commit\_QC}_\diamond$ cannot be formed, which contradicts our claim.

In addition, with Lemma 8 and Theorem 5, a non-faulty server that has the most up-to-date log will be elected as a leader for normal operation. Thus, a non-faulty leader is always aware of the highest sequence number and will not reassign a used sequence number for a `txBlock`.

Therefore, the combination of PrestigeBFT's view-change protocol and the standard two-phase replication protocol ensures safety, with no non-faulty servers deciding on conflicting blocks. ◻

A.4 Leadership fairness

Since the passive view-change protocol rotates leadership according to a predefined leader schedule, it intrinsically achieves leadership fairness as each server becomes a leader once in a circle of rotations. However, its leadership fairness is shared among all servers including faulty ones, which can always result in regular faulty views with unavailable leaders, especially under frequent view changes.

In contrast, PrestigeBFT's active view-change protocol achieves a stronger form of leadership fairness. Since faulty servers are penalized with worsening reputation penalties after showing a pattern of launching attacks, leadership will be eventually shared among non-faulty servers over the long run.

Theorem 8 (Strong leadership fairness). *PrestigeBFT eventually achieves leadership fairness among all non-faulty servers.*

Proof. With Lemma 3, faulty servers may ① become faulty leaders with increasing reputation penalties or ② become temporary non-faulty servers and launch attacks when they can get compensated.

In Scenario ①, after faulty servers exhaust their computation capability, leadership will be campaigned by only non-faulty servers. In Scenario ②, during the period when faulty servers behave correctly, leadership will also be campaigned by only non-faulty servers. Therefore, PrestigeBFT eventually achieves leadership fairness among all non-faulty servers. ◻

B Collected questions

In this section, we show questions that were collected during presentations, lectures, and conversions from various groups including ECE/CS graduate students, professors, and distributed system developers. Questions are arranged according to their related sections.

Question 1 (Motivation). *The passive view-change protocol indeed suffers from performance degradation, but the good thing about passive VC is that it can decide on a leader regardless of whether the f failures are crash failures or Byzantine failures. How does PrestigeBFT perform under a variety of attacks compared to the passive protocol?*

Answer. Compared to the simple passive VC protocol, PrestigeBFT has a more advanced and sophisticated VC protocol. As shown in the evaluation section, PrestigeBFT outperforms the passive VC protocol both under crash and Byzantine failures. When it comes to tolerating crash failures, PrestigeBFT shows a significant advantage. Since PrestigeBFT allows servers to actively campaign for leadership upon detecting a leader's failure, it never assigns an unavailable or a stale server as a leader. Additionally, the evaluation of quiet attacks (**F2**), similar to crash failures, demonstrates that PrestigeBFT remains unaffected while the passive VC protocol is severely impacted (see Figure 9).

Regarding tolerating Byzantine failures, PrestigeBFT has the capability to mitigate the impact of arbitrary faults and progressively improve its availability over time. Despite the fact that faulty servers can launch attacks that come with computational costs, PrestigeBFT may experience a brief period of low availability while increasing faulty servers' reputation penalties. However, PrestigeBFT surpasses the passive VC protocol as soon as its reputation mechanism responds appropriately to accumulated historical data in view changes and replication (see Figure 14). ◻

Question 2 (Motivation). *Why a speculative approach? Can we kick faulty servers out when some servers fail and reconfigure the system?*

Answer. Excluding faulty servers can be a temporary solution to deal with failures, but it does not represent a fault-tolerant approach. The focus of fault tolerance is to ensure that the system continues to function correctly even in the presence of failures.

Furthermore, in the context of Byzantine fault tolerance, distinguishing between benign and malicious behavior can be difficult. It is often impossible to determine whether a server is intentionally dropping a request or if the network is responsible for the failure. If we continuously exclude servers every time they exhibit a failure, we may soon find ourselves running out of servers, leading to frequent and manual configuration changes. ◻

Question 3 (Reputation mechanism). *What if bad clients collude with faulty leaders and send bad requests to the system to let the faulty leader gain some reputation and in turn let faulty servers enjoy penalty deductions?*

Answer. PrestigeBFT leaves the judgment of good and bad requests to the applications. As discussed in §3, users can define the criteria of useful `txBlocks` and the impact factor C_δ based on specific use cases. For example, in a financial application, a `txBlock` can be considered useful if its transactions are worth more than $\$1,000$, while transactions below this amount will not be counted in `ti` to receive compensation. This strategy can prevent frequent small transactions from impacting the calculation of reputation penalties.

PrestigeBFT proposes a general and versatile architecture incorporating a behavior-aware reputation mechanism, providing flexibility to its applications. This architecture enables user-defined information to convert behavior into a reputation penalty, which can be tailored to each application's unique requirements. □

Question 4 (Reputation mechanism)**.** *Will the increasing value of* `ti` *in the incremental log responsiveness make it more challenging for servers to receive compensation over time?*

Answer. The criterion of incremental log responsiveness is intended to reward servers that make increasing progress in replication, which prevents faulty servers from receiving compensation for making only limited progress. When faulty servers temporarily pretend to be correct in order to receive compensation, this criterion forces them to keep replicating more transactions after each time they receive compensation (e.g., examples ③ vs. ④ in Figure 4). This design has resulted in an improvement in availability when faulty servers choose to launch attacks only when they can receive compensation. In the long run, when the reputation penalties of at least $f+1$ non-faulty servers exceed the predefined threshold, the refresh mechanism will reset rp and ci to the initial value for these servers (discussed in 4.2.5). Consequently, the refresh will "rejuvenate" the calculation of δ_{tx}. □

Question 5 (Reputation mechanism)**.** *The reputation design is interesting. Your current approach seems to only reduce the interference of faulty servers in view changes. Can the reputation mechanism be adapted to also reduce the interference of faulty servers in replication? If so, will this increase the performance even more?*

Answer. The primary focus of PrestigeBFT is on view changes, as faulty leaders have the most harmful impact on leader-based BFT algorithms. While we have considered the possibility of introducing penalization in replication, we have two major concerns that have hindered us from implementing this feature. Firstly, under a faulty leader, f correct servers can always be blacked out in replication. Thus, it is not possible to judge reputation based on states, as a QC can always be constructed by $f+1$ correct servers and f faulty servers. Secondly, the reputation mechanism is currently only activated during view changes, which does not impose any additional overhead on replication. Penalizing wrongdoing during replication may require additional message passing among servers, which could introduce overhead.

However, we remain open to the idea of introducing penalization in replication in the future, as we continue to explore ways to build up more efficient and more robust fault tolerance algorithms. □

Question 6 (View changes)**.** *You mentioned that VDF is an alternative way of using PoW to implement the effect of reputation penalties. How would incorporating VDF to implement the effect of reputation penalties change the overall architecture?*

Answer. Changing PoW to VDF will not change PrestigeBFT's overall architecture. In fact, the reputation mechanism does not need to change at all. To use VDF, we first change the hash computation process of a redeemer (Line 36 to 39) to a delay function where the delay time is determined by the reputation penalty. Then, we change the verification of PoW computational work (**C5** in §4.2.3) to the verification of delayed time. □

Question 7 (View changes)**.** *What if a faulty candidate colludes with a faulty leader and tricks correct servers by sending them a block that is not the latest one, since stale servers cannot know what the latest transaction block is?*

Answer. This is a possible scenario, but it will not affect the correctness. In replication, each QC has a size of $2f+1$, there must be at least $f+1$ up-to-date and non-faulty servers knowing the latest `txBlock`. The faulty candidate cannot receive sufficient votes because all up-to-date and non-faulty servers will never vote for them. Our view-change protocol ensures that an elected leader must have the most up-to-date logs (Property **P2**). In addition, when stale servers receive a CampVC message from an up-to-date and non-faulty candidate, they will sync to up-to-date, regardless of any tricks from faulty candidates. □

Question 8 (View changes)**.** *In addition to leadership fairness, how can your algorithm support fairness in handling client requests?*

Answer. PrestigeBFT's active VC protocol supports the fairness problem of handling client requests by making frequent view changes more efficient and robust. As discussed in the introduction, faulty leaders can unfairly handle client requests. For example, faulty leaders can choose to handle the requests from selected clients first and intentionally slow down the consensus process for targeted clients. Some approaches such as Aardvark [22] and Diem [28] have proposed approaches to frequently change leadership through view changes in order to mitigate the unfair handling problem. PrestigeBFT's active VC protocol can be applied to replace the passive VC protocol used in these approaches with enhanced efficiency and robustness, leading to high performance in terms of throughput and latency under frequent view changes. □

C Examples

In this section, we show the step-by-step calculations that the reputation mechanism converts a server's behavior history into a reputation penalty. By walking through these calculations, we aim to provide a clear and comprehensive understanding of how the reputation mechanism operates and the results are used in view changes.

We assume a 4-server system including servers S_1, S_2, S_3, and S_4. The initial view is $V1$ where $rp = 1$ and $ci = 1$ for each server. The reputation segment of the initial `vcBlock` of view $V1$ (denoted by `vcBlock[V1]`) is presented below:

$$\texttt{vcBlock[V1]}.rp = \begin{cases} < ID:1, rp:1 > \\ < ID:2, rp:1 > \\ < ID:3, rp:1 > \\ < ID:4, rp:1 > \end{cases} \quad \texttt{vcBlock[V1]}.ci = \begin{cases} < ID:1, ci:1 > \\ < ID:2, ci:1 > \\ < ID:3, ci:1 > \\ < ID:4, ci:1 > \end{cases}$$

We show how the reputation penalty (rp) and compensation index (ci) of server S_1 are calculated based on its different behavior in view changes and replication.

① V1 V2 ··· V5

The blocks in ① show that S_1 has repeatedly possessed the leadership without making progress in replication. After $V1$, S_1 campaigned for view $V2$. It first goes through Eq. 1 (penalization), where $V = 1$, $V' = 2$, and $rp^{(1)} = 1$:

$$rp^{(2)}_{temp} = rp^{(1)} + V' - V = 2$$

After this, S_1 is not eligible to receive compensation because it has not replicated any transaction, resulting in its $\delta_{tx} = 0$. Thus, $rp^{(2)} = rp^{(2)}_{temp} = 2$, and `vcBlock[V2]` that S_1 prepares for the view $V2$ is as follows:

$$\texttt{vcBlock[V2]}.rp = \begin{cases} < ID:1, rp:2 > \\ < ID:2, rp:1 > \\ < ID:3, rp:1 > \\ < ID:4, rp:1 > \end{cases} \quad \texttt{vcBlock[V2]}.ci = \begin{cases} < ID:1, ci:1 > \\ < ID:2, ci:1 > \\ < ID:3, ci:1 > \\ < ID:4, ci:1 > \end{cases}$$

S_1 repeats this behavior to view $V5$, and `vcBlock[V5]` is as follows:

$$\texttt{vcBlock[V5]}.rp = \begin{cases} < ID:1, rp:5 > \\ < ID:2, rp:1 > \\ < ID:3, rp:1 > \\ < ID:4, rp:1 > \end{cases} \quad \texttt{vcBlock[V5]}.ci = \begin{cases} < ID:1, ci:1 > \\ < ID:2, ci:1 > \\ < ID:3, ci:1 > \\ < ID:4, ci:1 > \end{cases}$$

② V1 V2 ··· V5 T1 ··· T20 V6

In view $V5$, S_1 can not afford the increasing rp and decides to temporarily behave like a correct leader. The blocks in ② indicates that S_1 has replicated 20 transactions (i.e., `txBlock[T1]` to txBlock[T20]). Then, at the end of $V5$, when S_1 campaigns for $V6$, it first gets penalized by Eq. 1, where $V = 5$, $V' = 6$, and $rp^{(5)} = 5$:

$$rp^{(6)}_{temp} = rp^{(5)} + V' - V = 6$$

Then, S_1 can receive compensation, as it has replicated 20 `txBlocks` ($\texttt{ti} = 20$).

$$\delta_{tx} = \frac{\texttt{ti} - \texttt{ci}}{\texttt{ti}} = 1$$

Since $\mathcal{P} = \{1, 2, 3, 4, 5\}$, $\mu_{\mathcal{P}} = 3$ and $\sigma_{\mathcal{P}} = 1.41$; thus,

$$\delta_{vc} = 1 - Sigmoid(\frac{rp^{(5)} - \mu_{\mathcal{P}}}{\sigma_{\mathcal{P}}}) = 0.19$$

$$\delta = \delta_{tx} \times \delta_{vc} \times rp^{(6)}_{temp} = 1.14$$

$$rp^{(6)} = rp^{(6)}_{temp} - \lfloor \delta \rfloor = 5$$

Consequently, the reputation segment of `vcBlock[V6]` that S_1 sends is as follows:

$$\texttt{vcBlock[V6]}.rp = \begin{cases} < ID:1, rp:5 > \\ < ID:2, rp:1 > \\ < ID:3, rp:1 > \\ < ID:4, rp:1 > \end{cases} \quad \texttt{vcBlock[V6]}.ci = \begin{cases} < ID:1, ci:20 > \\ < ID:2, ci:1 > \\ < ID:3, ci:1 > \\ < ID:4, ci:1 > \end{cases}$$

③ V1 V2 ··· V5 T1 ··· T20 V6 T21 ··· T50 V7

In view $V6$, S_1 wants to get compensated again, so as shown in ③, it temporarily behaves correctly again by replicating another 30 `txBlocks` (i.e., 50 `txBlocks` in total). At the end of $V6$, when S_1 campaigns for view $V7$. It first gets penalized by Eq. 1, where $V = 6$, $V' = 7$, $rp^{(6)} = 5$:

$$rp^{(7)}_{temp} = rp^{(6)} + V' - V = 6$$

Then, the calculation moves to compensation. Since S_1 has used 20 `txBlocks` for the last compensation calculation, its $\texttt{ci} = 20$ and $\texttt{ti} = 50$.

$$\delta_{tx} = \frac{\texttt{ti} - \texttt{ci}}{\texttt{ti}} = 0.6$$

Since $\mathcal{P} = \{1, 2, 3, 4, 5, 5\}$, $\mu_{\mathcal{P}} = 3.33$ and $\sigma_{\mathcal{P}} = 1.49$; thus,

$$\delta_{vc} = 1 - Sigmoid(\frac{rp^{(5)} - \mu_{\mathcal{P}}}{\sigma_{\mathcal{P}}}) = 0.25$$

$$\delta = \delta_{tx} \times \delta_{vc} \times rp^{(7)}_{temp} = 0.89$$

$$rp^{(7)} = rp^{(7)}_{temp} - \lfloor \delta \rfloor = 6$$

In ③, S_1 cannot receive any compensation, and its reputation penalty increases to $rp = 6$. If S_1 wants to receive compensation in this view, it must get a higher δ_{tx}, as δ_{vc} remains unchanged in a view. For example, if S_1 replicates 80 more `txBlocks` in this view with $\texttt{ti} = 100$ (shown in ④), its δ_{tx} will result in compensation.

④ V1 V2 ··· V5 T1 ··· T20 V6 T21 ··· T50 ··· T100 V7

$$\delta_{tx} = \frac{\texttt{ti} - \texttt{ci}}{\texttt{ti}} = 0.8$$

In this case, with $\delta_{vc} = 0.25$,

$$\delta = \delta_{tx} \times \delta_{vc} \times rp^{(7)}_{temp} = 1.2$$

Thus, S_1 will be compensated.

$$rp^{(7)} = rp^{(7)}_{temp} - \lfloor \delta \rfloor = 5$$

Consequently, the reputation segment of `vcBlock[V7]` that S_1 sends is as follows:

$$\texttt{vcBlock[V7]}.rp = \begin{cases} < ID: 1, rp: 5 > \\ < ID: 2, rp: 1 > \\ < ID: 3, rp: 1 > \\ < ID: 4, rp: 1 > \end{cases} \quad \texttt{vcBlock[V7]}.ci = \begin{cases} < ID: 1, ci: 100 > \\ < ID: 2, ci: 1 > \\ < ID: 3, ci: 1 > \\ < ID: 4, ci: 1 > \end{cases}$$

The comparison of ③ and ④ shows that **the criterion of replication (δ_{tx}) entices servers to behave correctly by incentivizing a more up-to-date replication log.** In order to continuously receive compensation, S_1 must make incrementally growing progress in replication. If S_1 is a faulty server and only temporarily behaves correctly to receive compensation, the temporary period increases significantly after each time S_1 receives compensation.

In addition to enticing servers to have a more up-to-date replication, the **reputation mechanism also incentives "heavily penalized servers" to give up leadership** and stay as a follower for a while (e.g., until they can receive compensation again).

⑤ ··· T20 V6 T21 ··· T50 V7 ··· V8 ··· V14 V15

For example, in ⑤, if S_1 gives up leadership and does not campaign for view $V7$, its rp and ci remain unchanged. If S_1 operates as a follower through view $V7$ to $V14$ (gray `vcBlocks`), then at the end of $V14$, its $rp = 5$.

$$\mathcal{P} = \{1, 2, 3, 4, 5, 5, ..., 5\} \text{ //5 appears 10 times}$$

If S_1 campaigns for view $V15$, after penalization,

$$rp^{(15)}_{temp} = rp^{(14)} + V' - V = 6$$

Then, it goes through compensation with $\mu_{\mathcal{P}} = 4.28$ and $\sigma_{\mathcal{P}} = 1.27$.

$$\delta_{tx} = \frac{\texttt{ti} - \texttt{ci}}{\texttt{ti}} = 0.6$$

$$\delta_{vc} = 1 - Sigmoid(\frac{rp^{(14)} - \mu_{\mathcal{P}}}{\sigma_{\mathcal{P}}}) = 0.36$$

$$\delta = \delta_{tx} \times \delta_{vc} \times rp^{(15)}_{temp} = 1.29$$

Thus, S_1 will be compensated by a deduction of 1 with its rp unchanged.

$$rp^{(15)} = rp^{(15)}_{temp} - \lfloor \delta \rfloor = 5$$

In addition, if S_1 has replicated 400 `txBlocks` throughout the 14 views with its $\texttt{ti} = 400$ (as shown in ⑥), it will receive higher compensation for the better behavior from both sides.

⑥ ··· T20 V6 T21 ··· T50 V7 T51 ··· V8 ··· T400 V14 V15

$$\delta_{tx} = \frac{\texttt{ti} - \texttt{ci}}{\texttt{ti}} = 0.95$$

$$\delta = \delta_{tx} \times \delta_{vc} \times rp^{(15)}_{temp} = 2.05$$

Then, S_i will be compensated by a deduction of 2 with its rp decreased to 4.

$$rp^{(15)} = rp^{(15)}_{temp} - \lfloor \delta \rfloor = 4$$

In this section, we have demonstrated how the reputation mechanism calculates a server's reputation penalty and compensation index during view changes based on its behavior history through various examples. We have shown how the reputation mechanism incentivizes servers to maintain up-to-date replication and avoid frequent leadership repossession.