FnF-BFT: A BFT protocol with provable performance under attack

Zeta Avarikioti[1], Lioba Heimbach[2], Roland Schmid[2], Laurent Vanbever[2], Roger Wattenhofer[2], and Patrick Wintermeyer[2]

[1] TU Wien, Austria
[2] ETH Zürich, Switzerland

Abstract. We introduce FnF-BFT, the first partially synchronous BFT protocol with performance guarantees under truly byzantine attacks during stable networking conditions. At its core, FnF-BFT parallelizes the execution of requests by allowing all replicas to act as leaders independently. Leader parallelization distributes the load over all replicas. Consequently, FnF-BFT fully utilizes all correct replicas' processing power and increases throughput by overcoming the single-leader bottleneck.
We prove lower bounds on FnF-BFT's efficiency and performance in synchrony: the amortized communication complexity is linear in the number of replicas and thus competitive with state-of-the-art protocols; FnF-BFT's amortized throughput with less than $\frac{1}{3}$ byzantine replicas is at least $\frac{16}{27}$th of its best-case throughput. We also provide a proof-of-concept implementation and preliminary evaluation of FnF-BFT.

Keywords: BFT, SMR, parallel leaders, byzantine-resilient performance

1 Introduction

Byzantine fault tolerance has been the gold standard for making distributed systems more robust. Instead of modeling every single failure scenario, the byzantine failure model considers *arbitrarily malicious* actors that may infiltrate the system, thus covering many unforeseeable failure scenarios. This failure model has been broadly applied to *state machine replication (SMR)*. In SMR, a set of distributed replicas aims to agree on a unique ordering of client requests, even though a subset of the replicas, the byzantine failures, tries to disrupt the protocol. Therefore, the primary objectives of a protocol are the system's correctness (*safety*) and continuous progress (*liveness*). SMR protocols that offer these guarantees, i.e., are resilient against byzantine failures while continuing system operation, are known as *byzantine fault-tolerant (BFT)* protocols.

The first practical BFT system, PBFT [9], was introduced more than two decades ago and has inspired numerous other BFT systems, e.g., [22,18,38]. However, even today, BFT protocols do not scale well with an increasing number of replicas, making large-scale deployment of BFT systems a challenge. Often, the origin of this issue stems from the *single-leader bottleneck*: most BFT protocols rest the responsibility of uniquely ordering client requests on a single leader instead of distributing this task amongst the replicas [35]. More recently, protocols

tackling the single-leader bottleneck through *parallelization* emerged, demonstrating a staggering performance increase over state-of-the-art sequential-leader protocols [26,35,19,36,34,11,20]. Similar to most of their single-leader counterparts, these works only consider non-malicious faults for their performance analysis. However, malicious attacks may lead to significant performance losses that are not evaluated. While these systems exhibit promising performance with simple faults, they *fail to provide lower bounds on their performance under attack.*

Our contribution. In this work, we introduce the first parallel-leader BFT protocol *with a provable performance guarantee under truly byzantine attack in stable network conditions*, which we term Fast'n'Fair-BFT (FnF-BFT). To formally capture this performance guarantee, we define the *byzantine-resilient performance* of a BFT protocol as the ratio between its worst-case and its best-case throughput, i.e., the effective utilization. For FnF-BFT, we bound this ratio to be constant, meaning that the throughput of our protocol under byzantine faults is lower-bounded by a constant fraction of its best-case throughput where no faults are present. Concretely, we show that FnF-BFT achieves byzantine-resilient performance with a ratio of 16/27 while maintaining safety and liveness.

In short, FnF-BFT is the first BFT protocol that *provably* achieves all the following properties in the partially synchronous communication model, i.e., where after some unknown *global stabilization time (GST)*, messages are delivered within a known bound Δ.

- **Optimistic Performance:** After GST, the best-case throughput is $\Omega(n)$ times higher than the throughput of sequential-leader protocols.
- **Byzantine-Resilient Performance:** After GST, the worst-case throughput of the system is at least a constant fraction of its best-case throughput.
- **Efficiency:** After GST, the amortized authenticator complexity of reaching consensus is $\Theta(n)$.

FnF-BFT achieves these properties by combining two key insights: First, by enabling all replicas to continuously act as leaders in parallel – sharing the load of clients' requests. Second, FnF-BFT does not replace leaders upon failure but configures each leader's load based on the leader's past performance. With this combination, we guarantee a *fair* distribution of requests according to each replica's capacity, which in turn results in *fast* processing of requests.

The rest of this paper is structured as follows: First, we present our formal model, an overview of the protocol, and define the protocol goals (§2). We then explain the design of FnF-BFT (§3), and analyze its security and performance formally (§4). We conclude with a related work section (§5). Proofs omitted in Section 4 can be found in Appendix A, where we present the complete analysis of FnF-BFT. A description and preliminary evaluation of our proof-of-concept implementation of FnF-BFT based on HotStuff [38] can be found in Appendix B.

2 FnF-BFT Overview

Model. The system consists of $n = 3f + 1$ authenticated replicas and a set of clients. We index replicas by $i \in [n] = \{1, 2, \ldots, n\}$. Throughout an execution, at most f unique replicas in the system are *byzantine*, i.e., they are controlled by

an adversary with full information on their internal state. All other replicas are *correct*, i.e., following the protocol. The adversary cannot intercept the communication between two correct replicas. Any number of clients may be byzantine.

Communication model: We assume a *partially synchronous communication model*, i.e., a known bound Δ on message transmission will hold between any two correct replicas after some unknown global stabilization time (GST). We show that FnF-BFT is safe in asynchrony, that is, when messages between correct replicas may arrive in arbitrary order after any finite delay. We evaluate all other properties of the system after GST, thus assuming a synchronous network.

Cryptographic primitives: We make the usual cryptographic assumptions: the adversary is computationally bounded, and cryptographically-secure communication channels, computationally secure hash functions, (threshold) signatures, and encryption schemes exist. Similar to other BFT algorithms [5,38,18], FnF-BFT makes use of threshold signatures. In an (l, n) threshold signature scheme, there is a single public key held by all replicas and clients. Additionally, each replica u holds a distinct private key allowing the generation of a partial signature $\sigma_u(m)$ of any message m. Any set of l distinct partial signatures for the same message, $\{\sigma_u(m) \mid u \in U, |U| = k\}$ can be combined (by any replica) into a unique signature $\sigma(m)$. The combined signature can be verified using the public key. We assume that the scheme is *robust*, i.e., any verifier can easily filter out invalid signatures from malicious replicas. In this work, we set $l = 2f + 1$.

Authenticator complexity: Message complexity has long been considered the main throughput-limiting factor in BFT protocols [18,38]. In practice, however, the throughput of a BFT protocol is limited by both its computational footprint (mainly caused by cryptographic operations), as well as its message complexity. Hence, to assess the performance and efficiency of FnF-BFT, we adopt a complexity measure called authenticator complexity [38]. An *authenticator* is any (partial) signature. We define the *authenticator complexity* of a protocol as the average number of all computations or verifications of any authenticator by replicas during the protocol execution per request. Note that the authenticator complexity also captures the message complexity of a protocol if, like in FnF-BFT, each message can be assumed to contain at least one signature. Unlike [38], where only the number of received signatures is considered, our definition allows to capture the load handled by each individual replica more accurately. Note that authenticator complexities according to the two definitions only differ by a constant factor. We only analyze the authenticator complexity after GST, as it is impossible for a BFT protocol to ensure deterministic progress and safety at the same time in an asynchronous network [15].

Protocol Overview. The FnF-BFT protocol implements a state machine (cf. Section 2) that is replicated across all replicas in the system. Clients broadcast requests to the system. Given client requests, replicas decide on the order of request executions and deliver commit-certificates to the clients.

Our protocol moves forward in *epochs*. In an epoch, each replica u is responsible for ordering a set of up to C_u client requests that are independent of all requests ordered by other replicas in the epoch. Every replica in the system

simultaneously acts as both a leader and a backup to the other leaders. The number of assigned client requests C_u is based on u's past performance as a leader. The client space is rotated between replicas between epochs to guarantee liveness. More precisely, during the epoch-change, a designated replica acting as primary: (a) ensures that all replicas have a consistent view of the past leader and primary performance, (b) deduces non-overlapping sequence numbers for each leader, and (c) assigns parts of the client space to leaders.

An epoch-change occurs when requested by more than two-thirds of replicas. Replicas requesting an epoch-change immediately stop participating in the previous epoch. The primary in charge of the epoch-change is selected through periodic rotation based on performance history. Replicas request an epoch-change if: (a) all replicas have exhausted their requests, (b) their local epoch timeout is exceeded, (c) not enough progress by other leaders is observed, or (d) enough other replicas request an epoch-change. Hence, epochs have bounded-length.

Protocol goals. FnF-BFT achieves scalable and byzantine fault-tolerant *state machine replication (SMR)*. In SMR, a group of replicas decide on a growing log of client requests. Clients are provided with cryptographically secure certificates which prove the commitment of their request. The protocol ensures:

1. **Safety:** If any two correct replicas commit a request with the same sequence number, they both commit the same request.
2. **Liveness:** If a correct client broadcasts a request, then every correct replica eventually commits the request.

Thus, FnF-BFT will eventually make progress, and valid client requests cannot be censored. Additionally, FnF-BFT guarantees low overhead in reaching consensus. Unlike other protocols limiting the worst-case efficiency for a single request, we analyze the amortized authenticator complexity per request after GST. We find this to be the relevant throughput-limiting factor:

3. **Efficiency:** After GST, the amortized authenticator complexity of reaching consensus is $\Theta(n)$.

Furthermore, FnF-BFT achieves competitive performance under both optimistic and pessimistic adversarial scenarios:

4. **Optimistic Performance:** After GST, the best-case throughput is $\Omega(n)$ times higher than the throughput of sequential-leader protocols.
5. **Byzantine-Resilient Performance:** After GST, the worst-case throughput of the system is at least a constant fraction of its best-case throughput.

Hence, unlike many other BFT systems, FnF-BFT *guarantees that byzantine replicas cannot arbitrarily slow down the system when the network is stable.*

3 FnF-BFT Architecture

FnF-BFT executes client requests on a state machine replicated across n replicas. We advance FnF-BFT in epochs – identified by monotonically increasing *epoch numbers*. Replicas in the system act as leaders and backups concurrently. As a leader, a replica is responsible for ordering client requests within its jurisdiction. Each leader v is assigned a predetermined number of requests C_v to execute during an epoch. To deliver a client request, v starts by picking the next

available sequence number and shares the request with the backups. Leader v must collect $2f + 1$ signatures from replicas in the leader prepare and commit phase (Algorithm 1) to commit the request. We employ threshold signatures for the signature collection – allowing us to achieve linear authenticator complexity for reaching consensus on a request. Additionally, we use low and high watermarks for each leader to represent a range of request sequence numbers that each leader can propose concurrently to boost individual leaders' throughput.

Each epoch has a unique primary responsible for the preceding epoch-change, i.e., moving the system into the epoch. The primary changes every epoch and its selection is based on the system's history. A replica calls for an epoch-change in any of the following cases: (a) the replica has locally committed requests for all sequence numbers available in the epoch, (b) the maximum epoch time expired, (c) the replica has not seen sufficient progress, or (d) the replica has observed at least $f + 1$ epoch-change messages from other replicas.

FnF-BFT generalizes PBFT [9] and Mir-BFT [35] to the n leader setting. Additionally, we avoid PBFT's expensive all-to-all communication during epoch operation similar to Linear-PBFT [18]. Throughout this section, we discuss the various components of the protocol in further detail.

3.1 Client

Each client has a unique identifier. A client c requests the execution of an operation r by sending a $\langle \text{request}, r, t, c \rangle$ to all leaders. Here, timestamp t is a monotonically increasing sequence number used to order the requests from one client. By using watermarks, we allow clients to have more than one request in flight. Client watermarks, low and high, represent the range of timestamp sequence numbers which the client can propose concurrently. Thus, we require t to be within the low and high watermarks of client c. The client watermarks are advanced similarly to the leader watermarks (cf. Section 3.6). Upon executing operation r, replica u responds to the client with $\langle \text{reply}, e, d, u \rangle$, where e is the epoch number and d is the request digest (cf. Section 3.5)[3]. The client waits for $f + 1$ such responses from the replicas.

3.2 Sequence Number Distribution

We distribute sequence numbers to leaders for the succeeding epoch during the epoch-change. While we commit requests from each leader in order, the requests from different leaders are committed independently of each other in our protocol. Doing so allows leaders to continue making progress in an epoch, even though other leaders might have stopped working. Otherwise, a natural attack for byzantine leaders is to stop working and force the system to an epoch-change. Such attacks are possible in other parallel-leader protocols such as Mir-BFT [35].

To allow leaders to commit requests independently of each other, we need to allocate sequence numbers to all leaders during the epoch-change. Thus, we must also determine the number of requests each leader is responsible for before

[3] Instead of committing client request independently, the protocol could be adapted to process client requests in batches – a standard BFT protocol improvement [22,38,35].

Algorithm 1 Committing a request proposed by leader v

Leader prepare phase
  as replica u:
    upon receiving a valid $\langle \text{request}, r, t, c \rangle$ from client c:
      map client request to hash bucket
  as leader v:
    accept $\langle \text{request}, r, t, c \rangle$ assigned to one of v's buckets
    pick next assigned sequence number sn
    broadcast $\langle \text{pre-prepare}, sn, e, h(r), v \rangle$
Backup prepare phase
  as backup w:
    accept $\langle \text{pre-prepare}, sn, e, h(r), v \rangle$
    if the pre-prepare message is valid:
      compute partial signature $\sigma_w(d)$, where $d = h(sn \| e \| r)$
      send $\langle \text{prepare}, sn, e, \sigma_w(d) \rangle$ to leader v
  as leader v:
    compute partial signature $\sigma_v(d)$
    upon receiving $2f$ prepare messages:
      compute $(2f + 1, n)$ threshold signature $\sigma(d)$
      broadcast $\langle \text{prepared-certificate}, sn, e, \sigma(d) \rangle$
Commit phase
  as backup w:
    accept $\langle \text{prepared-certificate}, sn, e, \sigma(d) \rangle$
    compute partial signature $\sigma(\sigma_w(d))$
    $\langle \text{commit}, sn, e, \sigma_w(\sigma(d)) \rangle$ to leader v
  as leader v:
    compute partial signature $\sigma(\sigma_v(d))$
    upon receiving $2f$ commit messages:
      compute $(2f + 1, n)$ threshold signature $\sigma(\sigma(d))$
      broadcast $\langle \text{commit-certificate}, sn, e, \sigma(\sigma(d)) \rangle$

the epoch. The number of requests for leader v in epoch e is denoted by $C_v(e)$. It can be computed deterministically by all replicas in the network, based on the known history of the system (cf. Section 3.7).

When assigning sequence numbers, we first automatically yield to each leader $v \in [n]$ the sequence numbers of the $O_v(e)$ existing hanging requests from previous epochs in the assigned bucket(s). The remaining $C_v(e) - O_v(e)$ sequence numbers for each leader are distributed to them one after the other according to their ordering from the set of available sequence numbers. Note that $O_v(e)$ cannot exceed $C_v(e)$. For each leader v the assigned sequence numbers are mapped to local sequence numbers $1_{v,e}, 2_{v,e}, \ldots, C_v(e)_{v,e}$ in epoch e. These sequence numbers are later used to simplify checkpoint creation (cf. Section 3.6).

3.3 Hash Space Division

The client hash space is partitioned into buckets to avoid duplication. Each bucket is assigned to a single leader every epoch. We consider the client identifier to be the request input and hash the client identifier ($h_c = h(c)$) to map requests

into buckets. The hash space partition ensures that no two conflicting requests will be assigned to different leaders[4].

Thus, the requests served by different leaders are independent of each other. Additionally, the bucket assignment is rotated round-robin across epochs, preventing request censoring. The hash space is portioned into $m \cdot n$ non-intersecting buckets of equal size, where $m \in \mathbb{Z}^+$ is a configuration parameter. Each leader v is then assigned $m_v(e)$ buckets in epoch e according to their load $C_v(e)$ (cf. Section 3.7). Leaders can only include requests from their active buckets.

When assigning buckets to leaders, the protocol ensures that every leader is assigned at least one bucket, as well as distributing the buckets according to the load handled by the leaders. Precisely, the number of buckets leader v is assigned in epoch e is given by $m_v(e) = \left\lfloor \frac{C_v(e)}{\sum_{u \in [n]} C_u(e)} (m-1) \cdot n \right\rfloor + 1 + \tilde{m}_v(e)$, where $\tilde{m}_v(e) \in \{0, 1\}$ distributes the remaining buckets to the leaders – ensuring $\sum_{u \in [n]} m_u(e) = m \cdot n$. The remaining buckets are allocated to leaders v with the biggest value: $\left\lfloor \frac{C_v(e)}{\sum_{u \in [n]} C_u(e)} (m-1) \cdot n \right\rfloor + 1 - \frac{C_v(e)}{\sum_{u \in [n]} C_u(e)} \cdot m \cdot n$.

Note that the system will require a sufficiently long stability period for all correct leaders to be working at their capacity limit, i.e., $C_v(e)$ matching the performance of leader v in epoch e. Once correct leaders operate at capacity, the number of buckets they serve matches that. The hash buckets are distributed to leaders through a deterministic rotation such that each leader repeatedly serves each bucket under $f + 1$ unique primaries, i.e., preventing byzantine replicas from censoring specific hash buckets. For the remaining paper, we assume that there are always client requests pending in each bucket. Since we aim to optimize throughput, this assumption is in-sync with our protocol goals.

3.4 Primary Rotation

While all replicas are tasked with being a leader at all times, only a single replica, the primary, initiates an epoch. FnF-BFT assigns primaries periodically, exploiting the performance of good primaries and being reactive to network changes. The primary rotation consists of two core building blocks. First, FnF-BFT repeatedly rotates through the $2f + 1$ best primaries – exploiting their performance. Second, FnF-BFT explores every primary at least once within a sliding window. The sliding window consists of $g \in \mathbb{Z}$ epochs, and we set $g \geq 3f + 1$ to allow the exploration of all primaries throughout a sliding window. We depict a sample rotation in Figure 1.

Throughout the protocol, all replicas record the performance of each primary. We measure performance as the number of requests successfully committed under a primary in an epoch. Performance can thus be determined during the succeeding epoch-change by each replica (cf. Section 3.7). To deliver a reactive system, we update a replica's primary performance after each turn.

[4] Note that in case the requests are transactions with multiple inputs, the hash space division is more challenging to circumvent double-spending attacks. In such cases, we can employ well-known techniques [39,21] with no performance overhead as long as the average number of transactions' inputs remains constant [7].

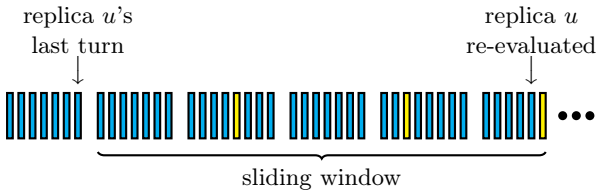

Fig. 1: FnF-BFT primary rotation in a system with $n = 10$ replicas. In blue, we show epochs led by primaries elected based on their performance. Epochs shown in yellow are led by replicas re-evaluated once their last turn as primary falls out of the sliding window.

We rotate through the best $2f + 1$ primaries repeatedly. After every $2f + 1$ primaries, the best $2f + 1$ primaries are redetermined and subsequently elected as primary in order of the time passed since their last turn as primary. The primary that has not been seen for the longest time is elected first. Cycling through the best primaries maximizes system performance. Simultaneously, basing performance solely on a replica's preceding primary performance strips byzantine primaries from the ability to misuse a good reputation. Every so often, we interrupt the continuous exploitation of the best $2f + 1$ primaries to revisit replicas that fall out of the sliding window. If replica u's last turn as primary occurred in epoch $e - g$ by the time epoch e rolls around, replica u would be re-explored as primary in epoch e. The exploration allows us to re-evaluate all replicas as primaries periodically and ensures that FnF-BFT is reactive to network changes.

The protocol starts by exploring all primaries ordered by their identifiers. Note that only one primary can fall out of the sliding window at any time after the first exploration. Thus, we always know which primary will be re-evaluated.

3.5 Epoch Operation

To execute requests, we use a leader-based adaption of PBFT, similar to Linear-PBFT [18]. Threshold signatures are commonly used to reduce the complexity of the backup prepare and commit phases of PBFT. The leader of a request is used as a collector of partial signatures to create a $(2f + 1, n)$ threshold signature in the intermediate stages of the backup prepare and commit phases. We visualize the schematic of the message flow for one request led by replica 0 in Figure 2 and summarize the protocol executed locally by replicas in Algorithm 1.

Leader prepare phase. Upon receiving $\langle \text{request}, r, t, c \rangle$ from a client, each replica computes the hash of the client identifier c. If the request falls into one of leader v's active buckets, v verifies $\langle \text{request}, r, t, c \rangle$. The request is discarded if either it has already been prepared or it is already pending. Once verified, leader v broadcasts $\langle \text{pre-prepare}, sn, e, h(r), v \rangle$, where sn is the sequence number, e the current epoch, $h(r)$ is the hash digest of request r and v represents the leader's signature. The cryptographic hash function h maps an arbitrary-length input to a fixed-length output. We can use the digest $h(r)$ as a unique identifier for a request r, as we assume the hash function to be collision-resistant.

Backup prepare phase. Backup w accepts $\langle \text{pre-prepare}, sn, e, h(r), v \rangle$ from leader v, if (a) the epoch number matches its local epoch number, (b) w has

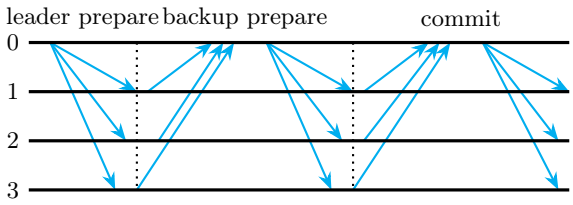

Fig. 2: Schematic message flow for one request.

not prepared another request with the same sequence number sn in epoch e, (c) leader v leads sequence number sn, (d) sn lies between the low and high watermarks of leader v, (e) r is in the active bucket of v, and (f) r was submitted by an authorized client. Upon accepting $\langle \text{pre-prepare}, sn, e, h(r), v\rangle$, w computes $d = h(sn \| e \| r)$ where h is a hash function. Additionally, w signs d by computing a verifiable partial signature $\sigma_w(d)$. Then w sends $\langle \text{prepare}, sn, e, \sigma_w(d)\rangle$ to leader v. Upon receiving $2f$ prepare messages for sn in epoch e, leader v forms a combined signature $\sigma(d)$ from the $2f$ prepare messages and its own signature. Leader v then broadcasts $\langle \text{prepared-certificate}, sn, e, \sigma(d)\rangle$ to all backups.

Commit phase. Backup w accepts the *prepared-certificate* and replies with $\langle \text{commit}, sn, e, \sigma_w(\sigma(d))\rangle$ to leader v. After collecting $2f$ commit messages, v creates a combined signature $\sigma(\sigma(d))$ using the signatures from the collected commit messages and its own signature. Once the combined signature is prepared, v continues by broadcasting $\langle \text{commit-certificate}, sn, e, \sigma(\sigma(d))\rangle$. Upon receiving the *commit-certificate*, replicas execute r after delivering all preceding requests led by v, and send replies to the client.

3.6 Checkpointing

Similar to PBFT [9], we periodically create checkpoints to prove the correctness of the current state. Instead of requiring a costly round of all-to-all communication to create a checkpoint, we add an intermediate phase and let the respective leader collect partial signatures to generate a certificate optimistically. Additionally, we expand the PBFT checkpoint protocol to run for n parallel leaders.

For each leader v, we repeatedly create checkpoints to clear the logs and advance the watermarks of leader v whenever the local sequence number $sn_{v,e,k}$ is divisible by a constant $k \in \mathbb{Z}^+$. Recall that when a replica u delivers a request for leader v with local sequence number $sn_{v,e,k}$, this implies that all requests led by v with local sequence number lower than $sn_{v,e,k}$ have been locally committed at replica u. Hence, after delivering the request with local sequence number $sn_{v,e,k}$, replica u sends $\langle \text{checkpoint}, sn_{v,e,k}, h(sn'_{v,e,k}), u\rangle$ to leader v. Here, $sn'_{v,e,k}$ is the last checkpoint and $h(sn'_{v,e,k})$ is the hash digest of the requests with sequence number sn_v in the range $sn'_{v,e,k} \leq sn_v \leq sn_{v,e,k}$. Leader v proceeds by collecting $2f+1$ checkpoint messages (including its own) and generates a *checkpoint-certificate* by creating a combined threshold signature. Then, leader v sends the checkpoint-certificate to all other replicas. If a replica sees the checkpoint-certificate, the checkpoint is *stable* and the replica can discard the corresponding messages from its logs, i.e., for sequence numbers belonging to leader v lower than $sn_{v,e,k}$.

We use checkpointing to advance low and high watermarks. In doing so, we allow several requests from a leader to be in flight. The low watermark L_v for leader v is equal to the sequence number of the last stable checkpoint, and the high watermark is $H_v = L_v + 2k$. We set k to be large enough such that replicas do not stall. Given its watermarks, leader v can only propose requests with a local sequence number between low and high watermarks.

Calling epoch-change. Replicas call an epoch-change by broadcasting an epoch-change message in four cases:

1. Replica u triggers an epoch-change in epoch e, once it has committed everyone's assigned requests locally.
2. Replica u calls for an epoch-change when its *epoch timer* expires. The value of the epoch timer T is set to ensure that after GST, correct replicas can finish at least $C_{\min}$ requests during an epoch. $C_{\min} \in \Omega(n^2)$ is the minimum number of requests assigned to leaders.
3. Replicas call epoch-changes upon observing inadequate progress. Each replica u has individual *no-progress timers* for all leaders. The no-progress timer is initialized with the same value T_p for all leaders. Initially, replicas set all no-progress timers for the first time after 5Δ in the epoch – accounting for the message transmission time of the initial requests. A replica resets the timer for leader v every time it receives a commit-certificate from v. In case the replica has already committed C_v requests for leader v, the timer is no longer reset. Upon observing no progress timeouts for $b \in [f+1, 2f+1]$ different leaders, a replica calls an epoch-change. Requiring at least $f+1$ leaders to make progress ensures that a constant fraction of leaders makes progress, and at least one correct leader is involved. On the other hand, we demand no more than $2f+1$ leaders to make progress such that byzantine leaders failing to execute requests cannot stop the epoch early. We let $b = 2f+1$ and set the no-progress timer such that it does not expire for correct leaders and simultaneously ensures sufficient progress, i.e., $T_p \in \Theta(T/C_{\min})$.
4. Finally, replica u calls an epoch-change if it sees that $f+1$ other replicas have called an epoch-change for an epoch higher than e. Replica u picks the smallest epoch in the set such that byzantine replicas cannot advance the protocol an arbitrary number of epochs.

After sending an epoch-change message, the replica will only start its epoch-change timer, upon seeing at least $2f+1$ epoch-change messages. We will discuss the epoch-change timer in more detail later.

3.7 Epoch-Change

At high level, in FnF-BFT's epoch-change protocol, we modify the PBFT view-change protocol as follows: we use threshold signatures to reduce the message complexity and extend the view-change message to include information about all leaders. Similar to Mir-BFT [35], we introduce a round of reliable broadcast to share information needed to determine the configuration of the next epoch(s). We determine the load assigned to each leader in the next epoch, based on their past performance, and also record the performance of the preceding primary.

Algorithm 2 Epoch-change protocol for epoch $e+1$

1: *Starting epoch-change*
2: **as** replica u:
3: broadcast $\langle \text{epoch-change}, e+1, \mathcal{S}, \mathcal{C}, \mathcal{P}, \mathcal{Q}, u \rangle$
4: **upon** receiving $2f$ epoch-change messages for $e+1$:
5: start epoch-change timer T_e
6: *Reliable broadcast*
7: **as** primary p_{e+1}:
8: compute $C_v(e+1)$ (Algorithm 3) for all leaders $v \in [n]$
9: perform 3-phase reliable broadcast sharing configuration details of epoch $e+1$ and the performance of primary p_e
10: **as** replica u:
11: participate in reliable broadcast initiates by p_{e+1}
12: *Starting epoch*
13: **as** primary p_{e+1}:
14: broadcast $\langle \text{new-epoch}, e+1, \mathcal{V}, \mathcal{O}, p_{e+1} \rangle$
15: enter epoch $e+1$
16: **as** replica u:
17: accept $\langle \text{new-epoch}, e+1, \mathcal{V}, \mathcal{O}, p_{e+1} \rangle$
18: enter epoch $e+1$

Starting epoch-change (Algorithm 2, steps 1-5). To move the system to epoch $e+1$, replica u sends $\langle \text{epoch-change}, e+1, \mathcal{S}, \mathcal{C}, \mathcal{P}, \mathcal{Q}, u \rangle$ to all replicas in the system. Here, $\mathcal{S}$ is a vector of sequence numbers sn_v of the last stable checkpoints S_v $\forall v \in [n]$ known to u for each leader v. $\mathcal{C}$ is a set of checkpoint-certificates proving the correctness of S_v $\forall v \in [n]$, while $\mathcal{P}$ contains sets $\mathcal{P}_v$ $\forall v \in [n]$. For each leader v, $\mathcal{P}_v$ contains a prepared-certificate for each request r that was prepared at u with sequence number higher than sn_v, if replica v does not possess a commit-certificate for r. Similarly, $\mathcal{Q}$ contains sets $\mathcal{Q}_v$ $\forall v \in [n]$. $\mathcal{Q}_v$ consists of a commit-certificate for each request r that was prepared at u with sequence number higher than sn_v.

Reliable broadcast (Algorithm 2, steps 6-11). The primary of epoch $e+1$ (p_{e+1}) waits for $2f$ epoch-change messages for epoch e. Upon receiving a sufficient number of messages, the primary performs a classical 3-phase reliable broadcast. During the broadcast, the primary informs leaders on the number of requests assigned to each leader in the next epoch and the identifiers of the replicas which send epoch-change messages. The number of requests assigned to a leader is computed deterministically (Algorithm 3). Through the reliable broadcast, we ensure that the primary cannot share conflicting information regarding the sequence number assignment and, in turn, the next epoch's sequence number distribution. In addition to sharing information about the epoch configuration, the primary also broadcasts the total number of requests committed during the previous epoch. This information is used by the network to evaluate primary performance and determine epoch primaries.

Starting epoch (Algorithm 2, steps 12-18). The primary p_{e+1} multicasts $\langle \text{new-epoch}, e+1, \mathcal{V}, \mathcal{O}, p_{e+1} \rangle$. Here, the set $\mathcal{V}$ contains sets $\mathcal{V}_u$, which carry the

valid epoch-change messages of each replica u of epoch e received by the primary of epoch $e+1$, plus the epoch-change message the primary of epoch $e+1$ would have sent. $\mathcal{O}$ consists of sets $\mathcal{O}_v$ $\forall v \in [n]$ containing pre-prepare messages and commit-certificates.

$\mathcal{O}_v$ is computed as follows. First, the primary determines the sequence number $S_{\min}(v)$ of the latest stable checkpoint in $\mathcal{V}$ and the highest sequence number $S_{\max}(v)$ in a prepare message in $\mathcal{V}$. For each sequence number sn_v between $S_{\min}(v)$ and $S_{\max}(v)$ of all leaders $v \in [n]$ there are three cases: (a) there is at least one set in $\mathcal{Q}_v$ of some epoch-change message in $\mathcal{V}$ with sequence number sn_v, (b) there is at least one set in $\mathcal{P}_v$ of some epoch-change message in $\mathcal{V}$ with sequence number sn_v and none in $\mathcal{Q}_v$, or (c) there is no such set. In the first case, the primary simply prepares a commit-certificate it received for sn_v. In the second case, the primary creates a new message $\langle \text{pre-prepare}, sn_v, e+1, d, p_{e+1} \rangle$, where d is the request digest in the pre-prepare message for sequence number sn_v with the highest epoch number in $\mathcal{V}$. In the third case, the primary creates a new pre-prepare message $\langle \text{pre-prepare}, sn_v, e+1, d^{null}, p_{e+1} \rangle$, where d^{null} is the digest of a special null request; a null request goes through the protocol like other requests, but its execution is a no-op. If there is a gap between $S_{\max}(v)$ and the last sequence number assigned to leader v in epoch e, these sequence numbers will be newly assigned in the next epoch.

Next, the primary appends the messages in $\mathcal{O}$ to its log. If $S_{\min}(v)$ is greater than the sequence number of its latest stable checkpoint, the primary also inserts the proof of stability (the checkpoint with sequence number $S_{\min}(v)$) in its log. Then it enters epoch $e+1$; at this point, it can accept messages for epoch $e+1$.

A replica accepts a new-epoch message for epoch $e+1$ if: (a) it is signed properly, (b) the epoch-change messages it contains are valid for epoch $e+1$, (c) the information in $\mathcal{V}$ matches the new request assignment, and (d) the set $\mathcal{O}$ is correct. The replica verifies the correctness of $\mathcal{O}$ by performing a computation similar to the one previously used by the primary. Then, the replica adds the new information contained in $\mathcal{O}$ to its log and decides all requests for which a commit-certificate was sent. Replicas rerun the protocol for messages with sequence numbers between $S_{\min}(v)$ and $S_{\max}(v)$ without a commit-certificate. They do not execute client requests again (they use their stored information about the last reply sent to each client instead). As request messages and stable checkpoints are not included in new-epoch messages, a replica might not have some of them available. In this case, the replica can easily obtain the missing information from other replicas in the system.

Hanging requests. While the primary sends out the pre-prepare message for all hanging requests, replicas in whose buckets the requests fall, are responsible for computing prepared- and commit-certificates of the individual requests. In the example shown in Figure 3, the primary of epoch $e+1$, replica 0, sends a pre-prepare message for a request in a bucket of replica 1, contained in the new-epoch message, to everyone. Replica 1 is then responsible for prepared- and commit-certificates, as well as collecting the corresponding partial signatures.

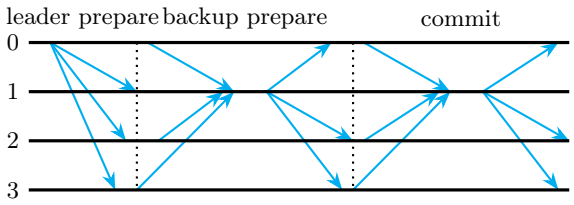

Fig. 3: Schematic of message flow for hanging requests. In this example, the primary is replica 0, and the request falls into the bucket of replica 1.

The number of request $C_v(e+1)$ assigned to leader v in epoch $e+1$ is determined deterministically based on its past performance (Algorithm 3). By $c_v(e)$ we denote the number of requests committed under leader v in epoch e. Each leader is re-evaluated during the epoch-change. If a leader successfully committed all assigned requests in the preceding epoch, we double the number of requests this leader is given in the following epoch. Else, it is assigned the maximum number of requests it committed within the last $f+1$ epochs.

Algorithm 3 Configuration adjustment

1: initially $C_v(1) = C_{\min}$ for all replicas v
2: **if** $c_v(e) < C_v(e)$:
3: $\quad C_v(e+1) = \max\left(C_{\min}, \max_{i \in \{0,\dots,f\}} c_v(e-i)\right)$
4: **else**:
5: $\quad C_v(e+1) = 2 \cdot c_v(e)$

Epoch-change timer. A replica sets an epoch-change timer T_e upon entering the epoch-change for epoch $e+1$. By default, we configure the timer T_e such that a correct primary can successfully finish the epoch-change after GST. If the timer expires without seeing a valid new-epoch message, the replica requests an epoch-change for epoch $e+2$. If a replica has experienced at least f unsuccessful consecutive epoch-changes previously, the replica doubles the timer's value. It continues to do so until it sees a valid new-epoch message. We only start doubling the timer after f unsuccessful consecutive epoch-changes to avoid having f byzantine primaries in a row, i.e., the maximum number of subsequent byzantine primaries possible, purposely increasing the timer value exponentially and, in turn, decreasing the system throughput significantly. As soon as replicas witness a successful epoch-change, they reduce T_e to its default again.

Assignment of requests Finally, the number of requests assigned to each leader is updated during the epoch-change. We limit the number of requests that can be processed by each leader per epoch to assign the sequence numbers ahead of time and allow leaders to work independently of each other.

We assign sequence numbers to leaders according to their abilities. As soon as we see a leader outperforming their workload, we double the number of requests they are assigned in the following epoch. Additionally, leaders operating below their expected capabilities are allocated requests according to the highest

potential demonstrated in the past $f+1$ rounds. By looking at the previous $f+1$ epochs, we ensure that there is at least one epoch with a correct primary in the leader set. In this epoch, the leader had the chance to display its capabilities. Thus, basing a leader's performance on the last $f+1$ rounds allows us to see its ability independent of the possible influence of byzantine primaries.

4 Analysis

We show that FnF-BFT satisfies the properties specified in Section 2. A detailed analysis can be found in Appendix A.

Safety. We prove FnF-BFT is safe under asynchrony. FnF-BFT generalizes Linear-PBFT [18], an adaptation of PBFT [9] that reduces its authenticator complexity during epoch operation. We thus rely on similar arguments to prove FnF-BFT's safety in Theorem 1.

Liveness. We show FnF-BFT makes progress after GST (Theorem 2). FnF-BFT's epoch-change uses the following techniques to ensure that correct replicas become synchronized (Definition 1) after GST: (1) A replica in epoch e observing epoch-change messages from $f+1$ other replicas calling for any epoch(s) greater than e issues an epoch-change message for the smallest such epoch e'. (2) A replica only starts its epoch-change timer for epoch e' after receiving $2f$ other epoch-change messages for epoch e', thus ensuring that at least $f+1$ correct replicas have broadcasted an epoch-change message for epoch e'. Hence, all correct replicas start their epoch-change timer for an epoch e' within at most 2 message delay. After GST, this amounts to at most 2Δ. (3) Byzantine replicas are unable to impede progress by calling frequent epoch-changes, as an epoch-change will only happen if at least $f+1$ replicas call it. A byzantine primary can hinder the epoch-change from being successful. However, there can only be f byzantine primaries in a row.

Efficiency. To demonstrate FnF-BFT's efficiency, we analyze the authenticator complexity for reaching consensus during an epoch. Like Linear-PBFT [18], using each leader as a collector for partial signatures in the backup prepare and commit phase, allows FnF-BFT to achieve linear complexity during epoch operation. We continue by calculating the authenticator complexity of an epoch-change. Intuitively speaking, we reduce PBFT's view-change complexity from $\Theta(n^3)$ to $\Theta(n^2)$ by employing threshold signatures. However, as FnF-BFT allows for n simultaneous leaders, we obtain an authenticator complexity of $\Theta(n^3)$ as a consequence of sharing the same information for n leaders during the epoch-change. Finally, we argue that after GST, there is sufficient progress by correct replicas to compensate for the high epoch-change cost (Theorem 3).

Optimistic Performance. We assess FnF-BFT's optimistic performance, i.e., we theoretically evaluate its best-case throughput, assuming all replicas are correct and the network is synchronous. We further assume that the best-case throughput is limited by the available computing power of each replica – mainly required for the computation and verification of cryptographic signatures – and that the available computing power of each correct replica is the

same. In this model, we demonstrate that FnF-BFT achieves higher throughput than sequential-leader protocols by the means of leader parallelization. To show the speed-up gained through parallelization, we first analyze the *optimistic epoch throughput* of FnF-BFT, i.e., the throughput of the system during stable networking conditions in the best-case scenario with $3f + 1$ correct replicas (Lemma 6). Later, we consider the repeated epoch changes and show that FnF-BFT's throughput is dominated by its authenticator complexity during the epochs. To that end, observe that for $C_{\min} \in \Omega(n^2)$, every epoch will incur an authenticator complexity of $\Omega(n^3)$ per replica and thus require $\Omega(n^3)$ time units. We show that after GST, an epoch-change under a correct primary requires $\Theta(n^2)$ time units (Lemma 7). We conclude our analysis by quantifying FnF-BFT's overall best-case throughput. Specifically, we prove that the speed-up gained by moving from a sequential-leader protocol to a parallel-leader protocol is proportional to the number of leaders (Theorem 4).

Byzantine-Resilient Performance. While many BFT protocols present practical evaluations of their performance that ignore byzantine adversarial behavior [9,18,38,35], we provide a novel, theoretical byzantine-resilience guarantee. We first analyze the impact of byzantine replicas in an epoch under a correct primary. We consider the replicas' roles as backups and leaders separately. On the one hand, for a byzantine leader, the optimal strategy is to leave as many requests hanging, while not making any progress (Lemma 9). On the other hand, as a backup, the optimal byzantine strategy is not helping other leaders to make progress (Lemma 10). In conclusion, we observe that byzantine replicas have little opportunity to reduce the throughput in epochs under a correct primary. Specifically, we show that after GST, the effective utilization under a correct primary is at least $\frac{8}{9}$ for $n \to \infty$ (Theorem 5).

Next, we discuss the potential strategies of a byzantine primary trying to stall the system. We first show that under a byzantine primary, an epoch is either aborted quickly or $\Omega(n^3)$ new requests become committed (Lemma 11). Then, we prove that rotating primaries across epochs based on primary performance history reduces the control of the byzantine adversary on the system. In particular, byzantine primaries only have one turn as primary throughout any sliding window in a stable network. Combining all the above, we conclude that FnF-BFT's byzantine-resilient utilization is asymptotically $\frac{8}{9} \cdot \frac{g-f}{g} > \frac{16}{27}$ for $n \to \infty$ (Theorem 7), where g is the fraction of byzantine primaries in the system's stable state, while simultaneously dictates how long it takes to get there after GST.

5 Related Work

Lamport et al. [23] first discussed the problem of reaching consensus in the presence of byzantine failures. Following its introduction, byzantine fault tolerance was initially studied in the synchronous network setting [30,13,12]. Dwork et al. [14] proposed the concept of partial synchrony and demonstrated the feasibility of reaching consensus in partially synchronous networks. Subsequently, Reiter [32,33] introduced Rampart, an early protocol tackling byzantine fault

tolerance for state machine replication in asynchrony. Then, with PBFT, Castro and Liskov [9] devised the first efficient protocol for state machine replication that tolerates byzantine failures. The leader-based protocol requires $\mathcal{O}(n^2)$ communication to reach consensus, as well as $\mathcal{O}(n^3)$ for leader replacement. While widely deployed, PBFT does not scale well when the number of replicas increases.

Kotla et al. [22] were the first to achieve $\mathcal{O}(n)$ complexity with Zyzzyva. The complexity of leader replacement in Zyzzyva remains $\mathcal{O}(n^3)$, and safety violations were later exposed [2]. Later, SBFT [18], improved the complexity of exchanging leaders to $\mathcal{O}(n^2)$. While reducing the overall complexity, both Zyzzyva and SBFT suffer from the single-leader bottleneck.

Developed by Yin et al. [38], leader-based HotStuff matches the $\mathcal{O}(n)$ complexity of Zyzzyva and SBFT. HotStuff rotates the leader with every request and is the first to achieve $\mathcal{O}(n)$ for leader replacement. However, HotStuff offers little parallelization due to its sequential proposal of requests, and experiments have revealed high complexity in practice [35]. Recently, Gelashvili et al. [16] improved on HotStuff's latency while adding an asynchronous fallback to enhance its performance during epoch synchronization. Although this work improves the overall performance of HotStuff, requests are still processed sequentially. In contrast, FnF-BFT enables n parallel leaders to propose requests simultaneously.

Parallel leaders. Leveraging parallel leaders to overcome the single-leader bottleneck was initially introduced by Mao et al. [26,28] with Mencius and BFT-Mencius. Mencius maps client requests to the closest leader, and in turn, requests can become censored. However, no de-duplication measures are in place to handle the re-submission of censored client requests. FnF-BFT addresses this problem by periodically rotating leaders over the client space.

Later, Stathakopoulou et al. [35] proposed Mir-BFT that significantly improved throughput compared to sequential-leader approaches. Mir runs instances of PBFT on a set of leaders, updating the leader set as soon as a single leader in the set stops making progress. Hence, we expect Mir's performance to drop significantly in the presence of byzantine replicas, as it allows byzantine leaders to repeatedly end epochs early. This is despite its high throughput demonstrated in the presence of faults. In a follow-up work, Stathakopoulou et al. [36] addressed Mir's temporary loss of throughput during epoch changes, but their protocol still offers no guarantees under attack, unlike FnF-BFT that maintains a constant fraction of its best-case throughput under byzantine attacks.

In parallel, Gupta et al. [19] proposed RCC protocol-agnostic approach to parallelize existing BFT protocols. While allowing multiple instances to each run an individual request, the protocol requires instances to unify after each request, creating a significant overhead. Further, RCC relies on failure detection, which is only possible in synchronous networks [24]. With FnF-BFT, we allow leaders to make progress independently of each other without relying on failure detection.

Another paradigm that has recently gained traction and enables replicas to operate in parallel to increase throughput is DAG-based consensus. The core idea is that the client requests are spread reliably as fast as the network permits and replicas accumulate them in a DAG. Subsequently, the replicas extract the total

ordering of the accumulated requests from their local DAG without exchanging additional messages. DAG-based protocols, initially introduced as consensus systems for data-center replication, precede blockchains [8,31,29,27,3]. These initial DAG protocols are, however, very complex and have high latency. Lately, several DAG-based consensus protocols have been proposed this time in the context of blockchains. Some are purely DAG-based [1] while others employ the DAG structure as transportation means for unconfirmed requests, e.g., [11], and on top execute a randomized BFT protocol [25,20,17,11,34]. The state-of-the-art Bullshark [34] achieves (minimum) constant latency with linear communication complexity in the partially synchronous model, similarly to FnF-BFT. While all these works achieve staggering throughput, none of them provide any provable guarantees on their throughput under Byzantine attacks.

Byzantine resilience. Byzantine resilience was initially explored by Clement et al. [10] with Aardvark. Aardvark is an adaptation of PBFT with frequent view-changes: a leader only stays in its position when displaying an increasing throughput level. This approach comes with significant performance cuts in networks without failures. Parallel leaders allow FnF-BFT to be byzantine-resilient without accepting significant performance losses in an ideal setting.

Prime, proposed by Amir et al. [4], aims to maximize performance in malicious environments. Besides adding delay constraints that further confine the partially synchronous network model, Prime restricts its evaluation to delay attacks, i.e., a byzantine leader adds as much delay to the protocol as possible. Similarly, Veronese et al. [37] only evaluated their proposed protocol, Spinning, in the presence of delay attacks – not fully capturing possible byzantine attacks. Consequently, the maximum performance degradation Spinning and Prime can incur under byzantine faults is at least 78% [6]. We analyze FnF-BFT theoretically to capture the entire spectrum of possible byzantine attacks.

Aublin et al. [6] further explored the performance of BFT protocols under byzantine attacks with RBFT. RBFT runs f backup instances on the same set of client requests as the master instance to discover whether the master instance is byzantine. Thus, RBFT incurs quadratic communication complexity for every request. To the contrary, FnF-BFT achieves a communication complexity of $\mathcal{O}(n)$ and further increases performance through parallelization – allowing byzantine-resilience without the added burden of detecting byzantine leaders.

Acknowledgments

The work was partially supported by the Austrian Science Fund (FWF) through the project CoRaF (grant agreement 2020388) and by the European Research Council (ERC) under the ERC Starting Grant (SyNET) 851809.

References

1. The swirdls hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf (Accessed: 2023-01-30)
2. Abraham, I., Gueta, G., Malkhi, D., Alvisi, L., Kotla, R., Martin, J.P.: Revisiting fast practical byzantine fault tolerance (2017)
3. Amir, Y., Dolev, D., Kramer, S., Malki, D.: Transis: a communication subsystem for high availability. In: [1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing. pp. 76–84 (1992). https://doi.org/10.1109/FTCS.1992.243613
4. Amir, Y., Coan, B., Kirsch, J., Lane, J.: Prime: Byzantine replication under attack. IEEE Transactions on Dependable and Secure Computing **8**(4), 564–577 (2010)
5. Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J., Zage, D.: Steward: Scaling byzantine fault-tolerant replication to wide area networks. IEEE Transactions on Dependable and Secure Computing **7**(1), 80–93 (2008)
6. Aublin, P., Mokhtar, S.B., Quéma, V.: Rbft: Redundant byzantine fault tolerance. In: ICDCS. pp. 297–306 (2013)
7. Avarikioti, G., Kokoris-Kogias, E., Wattenhofer, R.: Divide and scale: Formalization of distributed ledger sharding protocols (2019)
8. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: Proceedings of the Eleventh ACM Symposium on Operating Systems Principles. p. 123–138. SOSP '87, Association for Computing Machinery, New York, NY, USA (1987). https://doi.org/10.1145/41457.37515, https://doi.org/10.1145/41457.37515
9. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS) **20**(4), 398–461 (2002)
10. Clement, A., Wong, E.L., Alvisi, L., Dahlin, M., Marchetti, M.: Making byzantine fault tolerant systems tolerate byzantine faults. In: NSDI. pp. 153–168 (2009)
11. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: A dag-based mempool and efficient bft consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems. p. 34–50. EuroSys '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3492321.3519594, https://doi.org/10.1145/3492321.3519594
12. Dolev, D., Reischuk, R.: Bounds on information exchange for byzantine agreement. Journal of the ACM (JACM) **32**(1), 191–204 (1985)
13. Dolev, D., Strong, H.R.: Polynomial algorithms for multiple processor agreement. In: ACM STOC. pp. 401–407 (1982)
14. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) **35**(2), 288–323 (1988)
15. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
16. Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A., Xiang, Z.: Jolteon and & ditto: Network-adaptive efficient consensus with & asynchronous fallback. In: Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers. p. 296–315. Springer-Verlag, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-031-18283-9_14, https://doi.org/10.1007/978-3-031-18283-9_14
17. Gągol, A., Leundefinedniak, D., Straszak, D., undefinedwiundefinedtek, M.: Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In:

Proceedings of the 1st ACM Conference on Advances in Financial Technologies. p. 214–228. AFT '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3318041.3355467, https://doi.org/10.1145/3318041.3355467
18. Gueta, G.G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M.K., Seredinschi, D.A., Tamir, O., Tomescu, A.: Sbft: a scalable decentralized trust infrastructure for blockchains (2018)
19. Gupta, S., Hellings, J., Sadoghi, M.: Rcc: resilient concurrent consensus for high-throughput secure transaction processing. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE). pp. 1392–1403. IEEE (2021)
20. Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All you need is dag. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. p. 165–175. PODC'21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3465084.3467905, https://doi.org/10.1145/3465084.3467905
21. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: Omniledger: A secure, scale-out, decentralized ledger via sharding. In: IEEE SP. pp. 19–34 (2018)
22. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. SIGOPS Operating Systems Review **41**(6), 45–58 (2007)
23. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (1982)
24. Lynch, N.A.: Distributed algorithms. Elsevier (1996)
25. Malkhi, D., Szalachowski, P.: Maximal extractable value (mev) protection on a dag (2022). https://doi.org/10.48550/ARXIV.2208.00940, https://arxiv.org/abs/2208.00940
26. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: Building efficient replicated state machines for wans. In: USENIX OSDI. p. 369–384 (2008)
27. Melliar-Smith, P., Moser, L., Agrawala, V.: Broadcast protocols for distributed systems. IEEE Transactions on Parallel and Distributed Systems **1**(1), 17–25 (1990). https://doi.org/10.1109/71.80121
28. Milosevic, Z., Biely, M., Schiper, A.: Bounded delay in byzantine-tolerant state machine replication. In: IEEE SRDS. pp. 61–70 (2013)
29. Moser, L.E., Melliar-Smith, P.M.: Byzantine-resistant total ordering algorithms. Inf. Comput. **150**(1), 75–111 (1999). https://doi.org/10.1006/inco.1998.2770, https://doi.org/10.1006/inco.1998.2770
30. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM (JACM) **27**(2), 228–234 (1980)
31. Peterson, L.L., Buchholz, N.C., Schlichting, R.D.: Preserving and using context information in interprocess communication. ACM Trans. Comput. Syst. **7**(3), 217–246 (aug 1989). https://doi.org/10.1145/65000.65001, https://doi.org/10.1145/65000.65001
32. Reiter, M.K.: Secure agreement protocols: Reliable and atomic group multicast in rampart. In: ACM CCS. pp. 68–80 (1994)
33. Reiter, M.K.: The rampart toolkit for building high-integrity services. In: Theory and Practice in Distributed Systems, pp. 99–110 (1995)
34. Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: Dag bft protocols made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 2705–2718. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3559361, https://doi.org/10.1145/3548606.3559361

35. Stathakopoulou, C., David, T., Vukolić, M.: Mir-bft: High-throughput bft for blockchains (2019)
36. Stathakopoulou, C., Pavlovic, M., Vukolić, M.: State machine replication scalability made simple. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 17–33 (2022)
37. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C.: Spin one's wheels? Byzantine fault tolerance with a spinning primary. In: IEEE SRDS. pp. 135–144 (2009)
38. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: ACM PODC. pp. 347–356 (2019)
39. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: Scaling blockchain via full sharding. In: ACM CCS. pp. 931–948 (2018)

A Analysis

We show that FnF-BFT satisfies the properties specified in Section 2. In particular, we prove the safety and liveness of FnF-BFT, argue that it is efficient, and evaluate its resilience to byzantine attacks in stable network conditions.

A.1 Safety

FnF-BFT generalizes Linear-PBFT [18], which is an adaptation of PBFT [9] that reduces its authenticator complexity during epoch operation. We thus rely on similar arguments to prove FnF-BFT's safety in Theorem 1.

Theorem 1. *If any two correct replicas commit a request with the same sequence number, they both commit the same request.*

Proof. We start by showing that if $\langle$prepared-certificate, $sn, e, \sigma(d)\rangle$ exists, then $\langle$prepared-certificate, $sn, e, \sigma(d')\rangle$ cannot exist for $d' \neq d$. Here, $d = h(sn\|e\|r)$ and $d' = h(sn\|e\|r')$. Further, we assume the probability of $r \neq r'$ and $d = d'$ to be negligible. The existence of $\langle$prepared-certificate, $sn, e, \sigma(d)\rangle$ implies that at least $f + 1$ correct replicas sent a pre-prepare message or a prepare message for the request r with digest d in epoch e with sequence number sn. For $\langle$prepared-certificate, $sn, e, \sigma(d')\rangle$ to exist, at least one of these correct replicas needs to have sent two conflicting prepare messages (pre-prepare messages in case it leads sn). This is a contradiction.

Through the epoch-change protocol we further ensure that correct replicas agree on the sequence of requests that are committed locally in different epochs. The existence of $\langle$prepared-certificate, $sn, e, \sigma(d)\rangle$ implies that there cannot exist $\langle$prepared-certificate, $sn, e', \sigma(d')\rangle$ for $d' \neq d$ and $e' > e$. Any correct replica only commits a request with sequence number sn in epoch e if it saw the corresponding commit-certificate. For a commit-certificate for request r with digest d and sequence number sn to exist a set R_1 of at least $f + 1$ correct replicas needs to have seen $\langle$prepared-certificate, $sn, e, \sigma(d)\rangle$. A correct replica will only accept a pre-prepare message for epoch $e' > e$ after having received a new-epoch message for epoch e'. Any correct new-epoch message for epoch $e' > e$ must contain epoch-change messages from a set R_2 of at least $f + 1$ correct replicas. As there are $2f + 1$ correct replicas, R_1 and R_2 intersect in at least one correct replica u. Replica u's epoch-change message ensures that information about request r being prepared in epoch e is propagated to subsequent epochs, unless sn is already included in the stable checkpoint of its leader. In case the prepared-certificate is propagated to the subsequent epoch, a commit-certificate will potentially be propagated as well. If the new-epoch message only includes the prepared-certificate for sn, the protocol is redone for request r with the same sequence number sn. In the two other cases, the replicas commit sn locally upon seeing the new-epoch message and a correct replica will never accept a request with sequence number sn again. □

A.2 Liveness

One cannot guarantee safety and liveness for deterministic BFT protocols in asynchrony [15]. We will, therefore, show that FnF-BFT eventually makes progress after GST. In other words, we consider a stable network when discussing liveness. Furthermore, we assume that after an extended period without progress, the time required for local computation in an epoch-change is negligible. Thus, we focus on analyzing the network delays for liveness.

Definition 1. *Two replicas are called synchronized, if they start their epoch-change timer for an epoch e within at most 2Δ.*

Similar to PBFT [9], FnF-BFT's epoch-change employs three techniques to ensure that correct replicas become synchronized (Definition 1) after GST:

1. A replica in epoch e observing epoch-change messages from $f+1$ other replicas calling for any epoch(s) greater than e issues an epoch-change message for the smallest such epoch e'.
2. A replica only starts its epoch-change timer after receiving $2f$ other epoch-change messages, thus ensuring that at least $f+1$ correct replicas have broadcasted an epoch-change message for the epoch (or higher). Hence, all correct replicas start their epoch-change timer for an epoch e' within at most 2 message delay. After GST, this amounts to at most 2Δ.
3. Byzantine replicas are unable to impede progress by calling frequent epoch-changes, as an epoch-change will only happen if at least $f+1$ replicas call it. A byzantine primary can hinder the epoch-change from being successful. However, there can only be f byzantine primaries in a row.

Lemma 1. *After GST, correct replicas eventually become synchronized.*

Proof. Let u be the first correct replica to start its epoch-change timer for epoch e at time t_0. Following (2), this implies that u received at least $2f$ other epoch-change messages for epoch e (or higher). Of these $2f$ messages, at least f originate from other correct replicas. Thus, together with its own epoch-change message, at least $f+1$ correct replicas broadcasted epoch-change messages by time t_0. These $f+1$ epoch-change messages are seen by all correct replicas at the latest by time $t_0+\Delta$. Thus, according to (1), at time $t_0+\Delta$ all correct replicas broadcast an epoch-change message for epoch e. Consequently, at time $t_0+2\Delta$ all correct replicas have received at least $2f$ other epoch-change messages and will start the timer for epoch e. □

Lemma 2. *After GST, all correct replicas will be in the same epoch long enough for a correct leader to make progress.*

Proof. From Lemma 1, we conclude that after GST, all correct replicas will eventually enter the same epoch if the epoch-change timer is sufficiently large. Once the correct replicas are synchronized in their epoch, the duration needed for a correct leader to commit a request is bounded. Note that all correct replicas

will be in the same epoch for a sufficiently long time as the timers are configured accordingly. Additionally, byzantine replicas are unable to impede progress by calling frequent epoch-changes, according to (3). □

Theorem 2. *If a correct client c broadcasts request r, then every correct replica eventually commits r.*

Proof. Following Lemmas 1 and 2, we know that all correct replicas will eventually be in the same epoch after GST. Hence, in any epoch with a correct primary, the system will make progress. Note that a correct client will not issue invalid requests. It remains to show that an epoch with a correct primary and a correct leader assigned to hash bucket $h(c)$ will occur. We note that this is given by the bucket rotation, which ensures that a correct leader repeatedly serves each bucket in a correct primary epoch. □

A.3 Efficiency

To demonstrate that FnF-BFT is efficient, we first analyze the authenticator complexity for reaching consensus during an epoch. Like Linear-PBFT [18], using each leader as a collector for partial signatures in the backup prepare and commit phase allows FnF-BFT to achieve linear complexity during epoch operation.

Lemma 3. *The authenticator complexity for committing a request during an epoch is $\Theta(n)$.*

Proof. During the leader prepare phase, the authenticator complexity is at most n. The primary computes its signature to attach it to the pre-prepare message. This signature is verified by no more than $n-1$ replicas.

Furthermore, the backup prepare and commit phase's authenticator complexity is less than $3n$ each. Initially, at most $n-1$ backups, compute their partial signature and send it to the leader, who, in turn, verifies $2f$ of these signatures. The leader then computes its partial signature, as well as computing the combined signature. Upon receiving the combined signature, the $n-1$ backups need to verify the signature.

Overall, the authenticator complexity committing a request during an epoch is thus at most $7n + o(n) \in \Theta(n)$. □

Next, we analyze the authenticator complexity of an epoch-change. Intuitively speaking, we reduce PBFT's view-change complexity from $\Theta(n^3)$ to $\Theta(n^2)$ by employing threshold signatures. However, as FnF-BFT allows for n simultaneous leaders, we obtain an authenticator complexity of $\Theta(n^3)$ as a consequence of sharing the same information for n leaders during the epoch-change.

Lemma 4. *The authenticator complexity of an epoch-change is $\Theta(n^3)$.*

Proof. The epoch-change for epoch $e+1$ is initiated by replicas sending epoch-change messages to the primary of epoch $e+1$. Each epoch-change message holds n authenticators for each leader's last checkpoint-certificates. As there are at most $2k$ hanging requests per leader, further $\mathcal{O}(n)$ authenticators for prepared-

and commit-certificates of the open requests per leader are included in the message. Additionally, the sending replica also includes its signature. Each replica newly computes its signature to sign the epoch-change message, the remaining authenticators are already available and do not need to be created by the replicas. Thus, a total of no more than n authenticators are computed for the epoch-change messages. Note that epoch-change messages contain $\Theta(n)$ authenticators. Thus, the number of authenticators received by each replica is $\Theta(n^2)$.

After the collection of $2f + 1$ epoch-change messages, the primary performs a classical 3-phase reliable broadcast. The primary broadcasts the same signed message to start the classical 3-phase reliable broadcast. While the primary computes 1 signature, at most $n - 1$ replicas verify this signature. In the two subsequent rounds of all-to-all communication, each participating replica computes 1 and verifies $2f$ signatures. Thereby, the authenticator complexity of each round of all-to-all communication is at most $(2f + 1) \cdot n$. Thus, the authenticator complexity of the 3-phase reliable broadcast is bounded by $(4f + 3) \cdot n \in \Theta(n^2)$.

After successfully performing the reliable broadcast, the primary sends out a new-epoch message to every replica in the network. The new-epoch message contains the epoch-change messages held by the primary and the required pre-prepare messages for open requests. There are $\mathcal{O}(n)$ such pre-prepare messages, all signed by the primary. Finally, each new-epoch message is signed by the primary. Thus, the authenticator complexity of the new-epoch message is $\Theta(n^2)$. However, suppose a replica has previously received and verified an epoch-change from replica u whose epoch-change message is included in the new-epoch message. In that case, the replica no longer has to check the authenticators in u's epoch-change message again. For the complexity analysis, it does not matter when the replicas verify the signature. We assume that all replicas verify the signatures contained in the epoch-change messages before receiving the new-epoch messages. Thus, the replicas only need to verify the $\mathcal{O}(n)$ new authenticators contained in the new-epoch message.

Overall, the authenticator complexity of an epoch-change is $\Theta(n^3)$. □

Finally, we argue that after GST, there is sufficient progress by correct replicas to compensate for the high epoch-change cost.

Theorem 3. *After GST, the amortized authenticator complexity of committing a request is* $\Theta(n)$.

Proof. To find the amortized authenticator complexity of committing a request, we consider an epoch and the following epoch-change. After GST, the authenticator complexity of committing a request for a correct leader is $\Theta(n)$. The timeout value is set such that a correct worst-case leader creates at least $C_{\min}$ requests in each epoch initiated by a correct primary. Thus, there are $\Theta(n)$ correct replicas, each committing $C_{\min}$ requests. By setting $C_{\min} \in \Omega(n^2)$, we guarantee that at least $\Omega(n^3)$ requests are created during an epoch given a correct primary.

Byzantine primaries can ensure that no progress is made in epochs they initiate, simply by withholding the new-epoch messages. However, at most a

constant fraction of epochs lies in the responsibility of byzantine primaries. We conclude that, on average, $\Omega(n^3)$ requests are created during an epoch.

Following Lemma 4, the authenticator complexity of an epoch-change is $\Theta(n^3)$. Note that the epoch-change timeout T_e is set so that correct primaries can successfully finish the epoch-change after GST. Not every epoch-change will be successful immediately, as byzantine primaries might cause unsuccessful epoch-changes. Specifically, byzantine primaries can purposefully summon an unsuccessful epoch-change to decrease efficiency.

In case of an unsuccessful epoch-change, replicas initiate another epoch-change – and continue doing so – until a successful epoch-change occurs. However, we only need to start $\mathcal{O}(1)$ epoch-changes on average to be successful after GST, as the primary rotation ensures that at least a constant fraction of primaries is correct. Hence, the average cost required to reach a successful epoch-change is $\Theta(n^3)$.

We find the amortized request creation cost by adding the request creation cost to the ratio between the cost of a successful epoch-change and the number of requests created in an epoch, that is, $\Theta(n) + \frac{\Theta(n^3)}{\Omega(n^3)} = \Theta(n)$. □

A.4 Optimistic Performance

Throughout this section, we make the following optimistic assumptions: all replicas are considered correct, and the network is stable and synchronous. We employ this model to assess the optimistic performance of FnF-BFT, i.e., theoretically evaluating its best-case throughput. Note that this scenario is motivated by practical applications, as one would hope to have functioning hardware at hand, at least initially. Additionally, we assume that the best-case throughput is limited by the available computing power of each replica – predominantly required for the computation and verification of cryptographic signatures. We further assume that the available computing power of each correct replica is the same, which we believe is realistic as the same hardware will often be employed for each replica in practice. Without loss of generality, each leader can compute/verify one authenticator per time unit. As *throughput*, we define the number of requests committed by the system per time unit. Finally, we assume that replicas only verify the authenticators of relevant messages. For example, a leader receiving $3f$ prepare messages for a request will only verify $2f$ authenticators. Similarly, pre-prepare messages outside the leaders' watermarks will not be processed by backups. Note that we will carry all assumptions into the next section. There they will, however, only apply to correct replicas.

Sequential-Leader Protocols We claim that FnF-BFT achieves higher throughput than sequential-leader protocols due to its leader parallelization. To support this claim, we compare FnF-BFT's throughput to that of a generic sequential-leader protocol. The generic sequential-leader protocol serves as an asymptotic characterization of several sequential-leader protocols, e.g., [9,18,38].

A *sequential-leader protocol* characteristically relies on a unique leader at any point in time. During its reign, the leader is responsible for serving all client requests. The leader can be rotated repeatedly or only upon failure.

Lemma 5. *A sequential-leader protocol requires at least $\Omega(n)$ time units to process a client request.*

Proof. In sequential-leader protocols, a unique replica is responsible for serving all client requests at any point in time. This replica must verify $\Omega(n)$ signatures to commit a request while no other replica leads requests simultaneously. Thus, a sequential-leader protocol requires $\Omega(n)$ time units to process a request. □

Fig. 7: Sequential leader example with four leaders taking turns in serving client requests. Leader changes are indicated by vertical lines.

FnF-BFT Epoch With FnF-BFT, we propose a *parallel-leader protocol* that divides client requests into $m{\cdot}n$ independent hash buckets. Each hash bucket is assigned to a unique leader at any time. The hash buckets are rotated between leaders across epochs to ensure liveness (cf. Section 3.3). Within an epoch, a leader is only responsible for committing client requests from its assigned hash bucket(s). Overall, this parallelization leads to a significant speed-up.

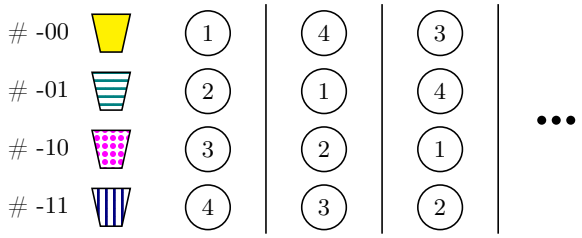

Fig. 8: Parallel leader example with four leaders and four hash buckets. In each epoch, leaders are only responsible for serving client requests in their hash bucket. Epoch-changes are indicated by vertical lines.

To show the speed-up gained through parallelization, we first analyze the *optimistic epoch throughput* of FnF-BFT, i.e., the throughput of the system during stable networking conditions in a best-case scenario with $3f + 1$ correct replicas. Furthermore, we assume the number of requests included in a checkpoint to be sufficiently large, such that no leader must ever stall when waiting for a checkpoint to be created. Finally, we analyze the effects of epoch-changes and compute the overall best-case throughput of FnF-BFT in the aforementioned optimistic setting.

Lemma 6. *After GST, the best-case epoch throughput with $3f+1$ correct replicas is* $\dfrac{k \cdot (3f+1)}{k \cdot (19f+3) + (8f+2)}$.

Proof. In the optimistic setting, all epochs are initiated by correct primaries, and thus all replicas will be synchronized after GST.

In FnF-BFT, n leaders work on client requests simultaneously. Similar to sequential-leader protocols, each leader needs to verify at least $\mathcal{O}(n)$ signatures to commit a request. A leader needs to compute 3 and verify $4f$ authenticators precisely to commit a request it proposes during epoch operation. Thus, leaders need to process a total of $4f+3 \in \Theta(n)$ signatures to commit a request. With the help of threshold signatures, backups involved in committing a request only need to compute 2 and verify 3 authenticators. We follow that a total of $4f+3+5 \cdot 3f = 19f+3$ authenticators are computed/verified by a replica for one of its own requests and $3f$ requests of other leaders.

After GST, each correct leader v will quickly converge to a C_v such that it will make progress for the entire epoch-time, hence, working at its full potential. We achieve this by rapidly increasing the number of requests assigned to each leader outperforming its assignment and never decreasing the assignment below what the replica recently managed.

Checkpoints are created every k requests and add to the computational load. A leader verifies and computes a total of $2f+2$ messages to create a checkpoint, and the backups are required to compute 1 partial signature and verify 1 threshold signature. The authenticator cost of creating $3f+1$ checkpoints, one for each leader, is, therefore, $8f+2$ per replica.

Thus, the best-case throughput of the system is

$$\frac{k \cdot (3f+1)}{k \cdot (19f+3) + (8f+2)}.$$

□

Note that it would have been sufficient to show that the epoch throughput is $\Omega(1)$ per time unit, but this more precise formula will be required in the next section. Additionally, we would like to point out that the choice of k does not influence the best-case throughput asymptotically.

FnF-BFT Epoch-Change As FnF-BFT employs bounded-length epochs, repeated epoch-changes have to be considered. In the following, we will show that FnF-BFT's throughput is dominated by its authenticator complexity during the epochs. To that end, observe that for $C_{\min} \in \Omega(n^2)$, every epoch will incur an authenticator complexity of $\Omega(n^3)$ per replica and thus require $\Omega(n^3)$ time units.

Lemma 7. *After GST, an epoch-change under a correct primary requires $\Theta(n^2)$ time units.*

Proof. Following Lemma 4, the number of authenticators computed and verified by each replica for all epoch-change messages is $\Theta(n^2)$. Each replica also processes $\Theta(n)$ signatures during the reliable broadcast, and $\mathcal{O}(n)$ signatures for the

new-epoch messages. Overall, each replica thus processes $\Theta(n^2)$ authenticators during the epoch-change. Subsequently, this implies that the epoch-change requires $\Theta(n^2)$ time units, as we require only a constant number of message delays to initiate and complete the epoch-change protocol. Recall that we assume the throughput to be limited by the available computing power of each replica. □

Theoretically, one could set $C_{\min}$ even higher such that the time the system spends with epoch-changes becomes negligible. However, there is a trade-off for practical reasons: increasing $C_{\min}$ increases the minimal epoch-length, allowing a byzantine primary to slow down the system for a longer time. Note that the guarantee for byzantine-resilient performance (cf. Section A.5) would still hold.

FnF-BFT Optimistic Performance Ultimately, it remains to quantify FnF-BFT's overall best-case throughput.

Lemma 8. *After GST, and assuming all replicas are correct,* FnF-BFT *requires $\mathcal{O}(n)$ time units to process n client requests on average.*

Proof. Under a correct primary, each correct leader will commit at least $C_{\min} \in \Omega(n^2)$ requests after GST. Hence, FnF-BFT will spend at least $\Omega(n^3)$ time units in an epoch, while only requiring $\Theta(n^2)$ time units for an epoch-change (Lemma 7). Thus, following Lemma 6, FnF-BFT requires an average of $\mathcal{O}(n)$ time units to process n client requests. □

Following Lemmas 5 and 8, the speed-up of a parallel-leader protocol over a sequential-leader protocol is proportional to the number of leaders.

Theorem 4. *If the throughput is limited by the (equally) available computing power at each replica, the speed-up for equally splitting requests between n parallel leaders over a sequential-leader protocol is at least $\Omega(n)$.*

A.5 Byzantine-Resilient Performance

While many BFT protocols present practical evaluations of their performance that neglect truly byzantine adversarial behavior [9,18,38,35], we provide a novel, theory-based byzantine-resilience guarantee. We first analyze the impact of byzantine replicas in an epoch under a correct primary. Next, we discuss the potential strategies of a byzantine primary trying to stall the system. And finally, we conflate our observations into a concise statement.

Correct Primary Throughput To gain insight into the byzantine-resilient performance, we analyze the optimal byzantine strategy. In epochs led by correct primaries, we will consider their roles as backups and leaders separately. On the one hand, for a byzantine leader, the optimal strategy is to leave as many requests hanging, while not making any progress (Lemma 9).

Lemma 9. *After GST and under a correct primary, the optimal strategy for a byzantine leader is to leave $2k$ client requests hanging and commit no request.*

Proof. Correct replicas will be synchronized as a correct primary initiates the epoch. Thus, byzantine replicas' participation is not required to make progress.

Byzantine leaders can follow the protocol accurately (at any chosen speed), send messages that do not comply with the protocol, or remain unresponsive.

Hanging requests reduce the throughput as they increase the number of authenticators shared during the epoch and the epoch-change. Hence, byzantine leaders leave the maximum number of requests hanging, i.e., $2k$ requests as all further prepare messages would be discarded by correct replicas.

While byzantine replicas cannot hinder correct leaders from committing requests, committing any request can only benefit the throughput of FnF-BFT. To that end, note that after GST, each correct leader v will converge to a C_v such that it will make progress during the entire epoch-time; hence, prolonging the epoch-time is impossible. The optimal strategy for byzantine leaders is thus to stall progress on their assigned hash buckets.

Finally, note that we assume the threshold signature scheme to be robust and can, therefore, discard any irrelevant message efficiently. □

On the other hand, as a backup, the optimal byzantine strategy is not helping other leaders to make progress (Lemma 10).

Lemma 10. *Under a correct primary, the optimal strategy for a byzantine backup is to remain unresponsive.*

Proof. Byzantine participation in the protocol can only benefit the correct leaders' throughput. Invalid messages can simply be ignored, while additional authenticators are not verified and thus do not reduce the system throughput. □

In conclusion, we observe that byzantine replicas have little opportunity to reduce the throughput in epochs under a correct primary.

Theorem 5. *After GST, the effective utilization under a correct primary is at least* $\frac{8}{9}$ *for* $n \to \infty$.

Proof. Moving from the best-case scenario with $3f + 1$ correct leaders to only $2f + 1$ correct leaders, each correct leader still processes $4f + 3$ authenticators per request, and 5 authenticators for each request of other leaders. We know from Lemma 9 that only the $2f + 1$ correct replicas are committing requests and creating checkpoints throughout the epoch. The authenticator cost of creating $2f + 1$ checkpoints, one for each correct leader, becomes $6f + 2$ per replica.

Byzantine leaders can open at most $2k$ new requests in an epoch. Each hanging request is seen at most twice by correct replicas without becoming committed. Thus, each correct replica processes no more than $8k$ authenticators for requests purposefully left hanging by a byzantine replica in an epoch. Thus, the utilization is reduced at most by a factor $\left(1 - \frac{8kf}{T}\right)$, where T is the maximal epoch length. While epochs can finish earlier, this will not happen after GST as soon as each correct leader v works at its capacity C_v.

Hence, the byzantine-resilient epoch throughput becomes

$$\frac{k \cdot (2f+1)}{k \cdot (14f+3) + (6f+2)} \cdot \left(1 - \frac{8kf}{T}\right).$$

By comparing this to the best-case epoch throughput from Lemma 6, we obtain a maximal throughput reduction of

$$\frac{(2f+1)(k \cdot (19f+3) + (8f+2))}{(3f+1)(k \cdot (14f+3) + (6f+2))} \cdot \left(1 - \frac{8kf}{T}\right).$$

Observe that the first term decreases and approaches $\frac{8}{9}$ for $n \to \infty$:

$$\frac{(2f+1)(k \cdot (19f+3) + (8f+2))}{(3f+1)(k \cdot (14f+3) + (6f+2))} \overset{n\to\infty}{=} \frac{16+38k}{18+42k} \geq \frac{8}{9}.$$

We follow that the epoch time is $T \in \Omega(n^3)$, as we set $C_{\min} \in \Omega(n^2)$ and each leader requires $\Omega(n)$ time units to commit one of its requests. Additionally, we know that $8kf \in \mathcal{O}(n)$, and thus: $\left(1 - \frac{8kf}{T}\right) \overset{n\to\infty}{=} 1$.

For $n \to \infty$, the throughput reduction byzantine replicas can impose on the system during a synchronized epoch is therefore bounded by a factor $\frac{8}{9}$. □

Byzantine Primary Throughput A byzantine primary, evidently, aims to perform the epoch-change as slow as possible. Furthermore, a byzantine primary can impede progress in its assigned epoch entirely, e.g., by remaining unresponsive. We observe that there are two main byzantine strategies to be considered.

Lemma 11. *Under a byzantine primary, an epoch is either aborted quickly or $\Omega(n^3)$ new requests become committed.*

Proof. A byzantine adversary controlling the primary of an epoch has three options. Following the protocol and initiating the epoch for all $2f+1$ correct replicas will ensure high throughput and is thus not optimal. Alternatively, initiating the epoch for $s \in [f+1, 2f]$ correct replicas will allow the byzantine adversary to control the progress made in the epoch, as no correct leader can make progress without a response from at least one byzantine replica. However, slow progress can only be maintained as long as at least $2f+1$ leaders continuously make progress. By setting the no-progress timeout $T_p \in \Theta(T/C_{\min})$, $\Omega(n^3)$ new requests per epoch can be guaranteed. In all other scenarios, the epoch will be aborted after at most one epoch-change timeout T_e, the initial message transmission time 5Δ, and one no-progress timeout T_p.

Note that we do not increase the epoch-change timer T_e for f unsuccessful epoch-changes in a row. In doing so, we prevent f consecutive byzantine primaries from increasing the epoch-change timer exponentially; thus potentially reducing the system throughput significantly. □

FnF-BFT Primaries Primaries rotate across epochs based on their performance history to reduce the control of the byzantine adversary on the system.

Lemma 12. *After a sufficiently long stable time period, the performance of a byzantine primary can only drop below the performance of the worst correct primary once throughout the sliding window.*

Proof. The network is considered stable for a sufficiently long time when all leaders work at their capacity limit, i.e., the number of requests they are assigned in an epoch matches their capacity, and primaries have subsequently been explored once. As soon as all leaders are working at their capacity limit, we observe the representative performance of all correct primaries, at least.

FnF-BFT repeatedly cycles through the $2f + 1$ best primaries. A primary's performance is based on its last turn as primary. Consequently, a primary is removed from the rotation as soon as its performance drops below one of the f remaining primaries. We conclude that a byzantine primary will only be nominated beyond its single exploration throughout the sliding window if its performance matches at least the performance of the worst correct primary. □

As its successor determines a primary's performance, the successor can influence the performance slightly. However, this is bounded by the number of open requests – $\mathcal{O}(n)$ many – which we consider being well within natural performance variations, as $\Omega(n^3)$ requests are created in an epoch under a correct primary. Thus, we will disregard possible performance degradation originating at the succeeding primary.

From Lemma 12, we easily see that the optimal strategy for a byzantine primary is to act according to Lemma 11 – performing better would only help the system. In a stable network, byzantine primaries will thus only have one turn as primary throughout any sliding window. In the following, we consider a primary to be behaving byzantine if it performs worse than all correct primaries.

Theorem 6. *After the system has been in stability for a sufficiently long time period, the fraction of byzantine behaving primaries is* $\frac{f}{g}$.

Proof. Following from Lemma 12, we know that a primary can only behave byzantine once in the sliding window. There are a total of g epochs in a sliding window, and the f byzantine replicas in the network can only act byzantine in one epoch included in the sliding window. We see that the fraction of byzantine behaving primaries is $\frac{f}{g}$. □

The configuration parameter g determines the fraction of byzantine primaries in the system's stable state, while simultaneously dictating how long it takes to get there after GST. Setting g to a small value ensures that the system quickly recovers from asynchrony. On the other hand, setting g to larger values provides near-optimal behavior once the system is operating at its optimum.

FnF-BFT Byzantine-Resilient Performance Combining the byzantine strategies from Theorem 5, Lemma 11 and Theorem 6, we obtain the following.

Theorem 7. *After GST, the effective utilization is asymptotically* $\frac{8}{9} \cdot \frac{g-f}{g}$ *for* $n \to \infty$.

Proof. To estimate the effective utilization, we only consider the throughput within epochs. That is because the time spent in correct epochs dominates the time for epoch-changes, as well as the time for failed epoch-changes under byzantine primaries, as the number of replicas increases (Lemma 7). Without loss of generality, we consider no progress to be made in byzantine primary epochs. We make this assumption, as we cannot guarantee asymptotically significant throughput. From Theorem 5, we know that in an epoch initiated by a correct primary, the byzantine-resilient effective utilization is at least $\frac{8}{9}$ for $n \to \infty$. Further, at least $\frac{g-f}{g}$ of the epochs are led by correct primaries after a sufficiently long time period in stability and thus obey this bound (Theorem 6). In the limit for $n \to \infty$ the effective utilization is $\frac{8}{9} \cdot \frac{g-f}{g}$. □

B Implementation & Preliminary Evaluation

Features. FnF-BFT's proof-of-concept implementation is directly based on the code of HotStuff's open-source prototype implementation `libhotstuff`.[5] We implement the basic functionality of FnF-BFT including the epoch-change and watermarks, while only changing ≈ 2000 lines of code and maintaining the general framework and experiment setup. In addition, we extend both implementations to support BLS threshold signatures.[6]

Threshold Signatures. Note that while HotStuff is designed with threshold signatures in mind and relies on them for its theoretic performance analysis [38], `libhotstuff` uses sets of $2f + 1$ signatures instead of real threshold signatures. While this workaround maintains a complexity of $\mathcal{O}(n)$ for creation of such a "threshold signature", it comes at the expense of a verification complexity of $\mathcal{O}(n)$ as well. In HotStuff, this additional overhead affects mainly the non-primary replicas, which would otherwise be idle in the HotStuff protocol. However, the design of FnF-BFT ensures that all replicas' computational resources are utilized at all times. Since the originally used `secp256k1` cryptographic signatures appear to be more optimized than the BLS threshold signatures, and to ensure a fair comparison, we thus compare FnF-BFT's throughput and latency to HotStuff using the identical BLS threshold signature implementation.

Limitations. As in the theoretical analysis (see Section 3.1), we did not implement a batching process for client requests. Hence, each block contains only a single client request. For this reason, the expected throughput (and typically reported number for BFT protocols) in practical deployments is much higher.

While FnF-BFT's design inherently allows to utilize concurrent threads during epoch operation, e.g., to interact with all parallel leaders, our proof-of-concept implementation currently only supports single-threaded operation.

Setup & Methodology. We compare FnF-BFT's single-threaded implementation to both single- and multi-threaded HotStuff (with 12 threads per replica) with respect to best-case performance, i.e., throughput and latency when all n replicas operate correctly. Experiments are repeated for $n \in \{4, 7, 10, 16, 31, 61\}$ replicas. We deploy both protocols on Amazon EC2 using c5.4xlarge AWS cloud instances. Each replica is assigned to a dedicated VM instance with 16 CPU cores powered by Intel Xeon Platinum 8000 processors clocked at up to 3.6 GHz, 32 GB of RAM, and a maximum network bandwidth of 10 Gigabits per second.

We measure the average throughput and latency over multiple epochs to include the expected drop in perfomance for FnF-BFT during the epoch-change. For each experiment, we run both protocols for at least three minutes and measure their average performance accordingly. We divide the hash space into n buckets, resulting in one bucket per replica. For generating requests, we run the `libhotstuff` client with its default settings, meaning that the payload of each request is empty. Clients generate and broadcast four requests in parallel, and

[5] https://github.com/hot-stuff/libhotstuff
[6] https://github.com/herumi/bls

issue a new request whenever one of their requests is answered. For the throughput measurements, we launch sufficiently many clients until we observe that no buckets are idle. For the latency measurement, we run a single client instance such that the system does not operate at its throughput limit. In general, we use the same settings for both protocols wherever applicable (Table 1).

Parameter	Value
Requests per block	1
Threads per replica	1
Threads per client	4
Epoch timeout	30s

Parameter	Value
No progress timeout	2s
Blocks per checkpoint (K)	50
Watermark window size ($2 * K$)	100
Initial epoch watermark bounds	10000

Table 1: Experiment parameters used for FnF-BFT and HotStuff, if applicable.

Performance. Figure 9 depicts a best-case operation of FnF-BFT over five epochs and demonstrates its consistently high throughput.[7] As expected, the throughput of our protocol stalls during an epoch-change. However, in comparison to HotStuff (Figure 10), FnF-BFT's average throughput remains on top over multiple epochs (i.e., including the epoch-change gap). As HotStuff's throughput decreases for increasing number of replicas, FnF-BFT showcases its superior scalability. Specifically, FnF-BFT handles large amounts of requests up to 4.7× faster than multi-threaded and 16× faster than single-threaded HotStuff.

Figure 11 depicts the average latency of both FnF-BFT and HotStuff, showing that they scale similarly with the number of replicas. As latency expresses the time between a request being issued and committed, both protocols exhibit very fast finality for requests on average, even with many replicas. In combination, Figure 10 and Figure 11 demonstrate the high performance and competitiveness of FnF-BFT with HotStuff, especially when scaling to many replicas.

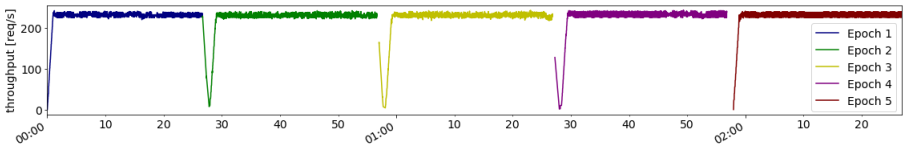

Fig. 9: Throughput of FnF-BFT with $n = 4$ replicas over 5 epochs.

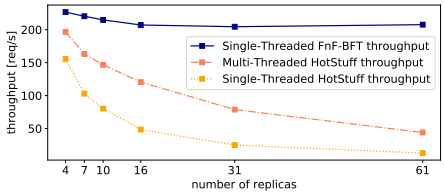

Fig. 10: Average Throughput Comparison

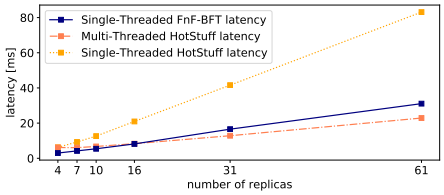

Fig. 11: Average Latency Comparison

[7] Note that a rate of 200 batches per second with a typical batch size of 500 commands per batch translates to a throughput of 100,000 requests per second.