Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback

Rati Gelashvili
Novi Research

Lefteris Kokoris-Kogias
Novi Research & IST Austria

Alberto Sonnino
Novi Research

Alexander Spiegelman
Novi Research

Zhuolun Xiang*
University of Illinois at Urbana-Champaign

Abstract

Existing committee-based Byzantine state machine replication (SMR) protocols, typically deployed in production blockchains, face a clear trade-off: (1) they either achieve linear communication cost in the happy path, but sacrifice liveness during periods of asynchrony, or (2) they are robust (progress with probability one) but pay quadratic communication cost. We believe this trade-off is unwarranted since existing linear protocols still have asymptotic quadratic cost in the worst case.

We design `Ditto`, a Byzantine SMR protocol that enjoys the best of both worlds: optimal communication on and off the happy path (linear and quadratic, respectively) and progress guarantee under asynchrony and DDoS attacks. We achieve this by replacing the view-synchronization of partially synchronous protocols with an asynchronous fallback mechanism at no extra asymptotic cost. Specifically, we start from HotStuff, a state-of-the-art linear protocol, and gradually build `Ditto`. As a separate contribution and an intermediate step, we design a 2-chain version of HotStuff, `Jolteon`, which leverages a quadratic view-change mechanism to reduce the latency of the standard 3-chain HotStuff.

We implement and experimentally evaluate all our systems. Notably, `Jolteon`'s commit latency outperforms HotStuff by 200-300ms with varying system size. Additionally, `Ditto` adapts to the network and provides better performance than `Jolteon` under faulty conditions and better performance than VABA (a state-of-the-art asynchronous protocol) under faultless conditions. This proves our case that breaking the robustness-efficiency trade-off is in the realm of practicality.

1 Introduction

The popularity of blockchain protocols has generated a surge in researching how to increase the efficiency and robustness of underlying consensus protocols used for agreement. On the efficiency front, the focus has been on decreasing the communication complexity in the steady state ("happy path"), first to quasilinear [19] and ultimately to linear [15, 38]. These protocols work in the eventually synchronous model and require a leader to aggregate proofs. However, handling leader failures or unexpected network delays requires quadratic communication and if the network is asynchronous, there is no liveness guarantee. On the robustness side, recent protocols [3, 24, 35] make progress by having each replica act as the leader and decide on a leader retroactively. This inherently requires quadratic communication even under good network conditions when the adversary is strongly adaptive [2].

We believe that in practice, we need the best of both worlds. An efficient happy path is beneficial for any production system, but for blockchains to support important (e.g. financial) infrastructure, robustness against asynchrony is also key. First, unpredictable network delays are a common condition when running in a large-scale network environment, e.g. over the Internet. Second, the possibility of targeted DDoS attacks on the leaders of leader-based protocols motivates the leaderless nature of the asynchronous solutions. Thus, we are interested if there are efficient systems that have a linear happy path and are robust against asynchrony.

This is an important question, posed as early as [22], and studied from a theoretical prospective [30, 34], but the existing blockchain systems still forfeit robustness for efficiency [15, 19, 38]. In this paper, we answer the above question with the first practical system, tailor-made to directly apply to the prominent HotStuff/DiemBFT [36, 38] family of protocols. Our protocol, `Ditto`, combines the optimistic (good network conditions) efficient happy path with pessimistic (worst-case network conditions) liveness guarantees with no extra asymptotic communication cost. `Ditto` is based on the key observation that when there is asynchrony or failures, the protocols with linear happy path still pay the quadratic cost, same as state-of-the-art asynchronous protocols (e.g., VABA [3] or Dumbo [24]) that provide significantly more robustness. Specifically, `Ditto` replaces the pacemaker of HotStuff/DiemBFT (a quadratic module that deals with

*Lead author, part of work was done at Novi Research

	Problem	Comm. Complexity	Rounds	Liveness
HotStuff [38]/DiemBFT [36]	partially sync SMR	sync $O(n)$	7	not live if async
VABA [3]	async BA	$O(n^2)$	E(16.5)	always live
Jolteon	partially sync SMR	sync $O(n)$	5	not live if async
Ditto	async SMR with fast sync path	sync $O(n)$, async $O(n^2)$	sync 5, async E(13.5)	always live

Table 1: Theoretical comparison of our protocol implementations. For HotStuff/DiemBFT and our protocols, sync $O(n)$ assumes honest leaders. Communication complexity measures the cost per consensus decision (committed block). Rounds measure the block-commit latency. $E(r)$ means r rounds in expectation. The async $E(13.5)$ latency of Ditto uses the MVBA protocol of [] which has an expected latency of 9.5 rounds.

view synchronization) with an asynchronous fallback.

In other words, instead of synchronizing views that will anyway fail (timeout) due to asynchrony or faults, we fall back to an asynchronous protocol that robustly guarantees progress at the cost of a single view-change. Furthermore, Ditto switches between the happy path and the fallback without overhead (e.g. additional rounds), and continues operating in the same pipelined fashion as HotStuff/DiemBFT.

Leveraging our observation further, we abandon the linear view-change [38] of HotStuff/DiemBFT, since the protocol anyway has a quadratic pacemaker [36]. We present Jolteon, a protocol that's a hybrid of HotStuff [38] and classical PBFT [9]. In particular, Jolteon preserves the structure of HotStuff and its linearity under good network conditions while reducing the steady state block-commit latency by 30% using a 2-chain commit rule. This decrease in latency comes at the cost of a quadratic view-change ($O(n)$ messages of $O(n)$ size). As the pacemaker is already quadratic, as expected, this does not affect performance in our experiments. We shared our findings with the Diem Team who is currently integrating Jolteon into the next release of DiemBFT.

Since Jolteon outperforms the original HotStuff in every scenario, we use it as the basis for Ditto. As shown experimentally under optimistic network conditions both Jolteon and Ditto outperform HotStuff block-commit latency. Importantly, Ditto's performance is almost identical to Jolteon in this case whereas during attack Ditto performs almost identically to our implementation of VABA [3]. Finally, the throughput of Ditto is 50% better than VABA in the optimistic path and much better than HotStuff and Jolteon under faulty (dead) leaders (30-50% better) or network instability (they drop to 0).

In Table 1 we show a theoretical comparison of our work to HotStuff and VABA, for which we implement and evaluate the performance experimentally in Section 5.2. More comprehensive comparisons and related work are given in Section 6.

2 Preliminaries

We consider a permissioned system that consists of an adversary and n replicas numbered $1, 2, \ldots, n$, where each replica has a public key certified by a public-key infrastructure (PKI). The replicas have all-to-all reliable and authenticated communication channels controlled by the adversary. We say a replica multicasts a message if it sends the message to all replicas. We consider a dynamic adversary that can adaptively corrupt up to f replicas, referred as *Byzantine*. The rest of the replicas are called *honest*. The adversary controls the message delivery times, but we assume messages among honest replicas are eventually delivered.

An execution of a protocol is *synchronous* if all message delays between honest replicas are bounded by Δ; is *asynchronous* if they are unbounded; and is *partially synchronous* if there is a global stabilization time (GST) after which they are bounded by Δ [11]. Without loss of generality, we let $n = 3f + 1$ where f denotes the assumed upper bound on the number of Byzantine faults, which is the optimal worst-case resilience bound for asynchrony, partial synchrony [11], or asynchronous protocols with fast synchronous path [4].

Cryptographic primitives and assumptions. We assume standard digital signature and public-key infrastructure (PKI), and use $\langle m \rangle_i$ to denote a message m signed by replica i. We also assume a threshold signature scheme, where a set of signature shares for message m from t (the threshold) distinct replicas can be combined into one threshold signature of the same length for m. We use $\{m\}_i$ to denote a threshold signature share of a message m signed by replica i. We also assume a collision-resistant cryptographic hash function $H(\cdot)$ that can map an input of arbitrary size to an output of fixed size. For simplicity of presentation, we assume the above cryptographic schemes are ideal and a trusted dealer equips replicas with these cryptographic materials. The dealer assumption can be lifted using the protocol of [20].

Any deterministic agreement protocol cannot tolerate even a single fault under asynchrony due to FLP [12]. Our asynchronous fallback protocol generates distributed randomness following the approach of [8]: the generated randomness of any given view is the hash of the unique threshold signature

on the view number.

BFT SMR. A Byzantine fault-tolerant state machine replication protocol [9] commits client transactions as a log akin to a single non-faulty server, and provides the following two guarantees:

- Safety. Honest replicas do not commit different transactions at the same log position.
- Liveness. Each client transaction is eventually committed by all honest replicas.

Besides these two requirements, a validated BFT SMR protocol must also satisfy *external validity* [7], requiring all committed transactions to be *externally valid*, i.e. satisfying some application-dependent predicate. This can be accomplished by adding validity checks on the transactions before the replicas propose or vote, and for brevity, we omit the details and focus on the BFT SMR formulation defined above. We assume that each client transaction will be repeatedly proposed by honest replicas until it is committed[1]. For most of the paper, we omit the client from the discussion and focus on replicas.

SMR protocols usually implement many instances of single-shot Byzantine agreement, but there are various approaches for ordering. We focus on the chaining approach, used in HotStuff [38] and DiemBFT [36], in which each proposal references the previous one and each commit commits the entire prefix of the chain.

2.1 Terminology

We present the terminology used throughout the paper. For brevity, we omit explicit format checks in the protocol description. The honest replica will discard messages that are not correctly formatted according to the protocol specification.

- *Round Number and View Number.* The protocol proceeds in rounds and views and each replica keeps track of the current round number r_{cur} and view number v_{cur}, which are initially set to 0. Each view can have several rounds and it is incremented by 1 after each asynchronous fallback. Each round r has a designated leader L_r that proposes a new block (defined below) of transactions in round r. Since DiemBFT and `Jolteon` (Section 3) do not have an asynchronous fallback, the view number is always 0, but it is convenient to define it here so that our `Ditto` protocol (Section 4) uses the same terminology.
- *Block Format.* A block is formatted as $B = [id, qc, r, v, txn]$ where qc is the quorum certificate (defined below) of B's parent block in the chain, r is the round number of B, v is the view number of B, txn is a batch of new transactions, and $id = H(qc, r, v, txn)$ is the unique hash digest of qc, r, v, txn. Note that when describing the protocol, it suffices to specify qc, r, and v for a new block, since txn, id follow the definitions. We will use $B.x$ to denote the element x of B.
- *Quorum Certificate.* A *quorum certificate (QC)* of some block B is a threshold signature of a message that includes $B.id, B.r, B.v$, produced by combining the signature shares $\{B.id, B.r, B.v\}$ from a quorum of $n - f = 2f + 1$ replicas. We say a block is *certified* if there exists a QC for the block. Blocks are chained by QCs to form a blockchain, or block-tree if there are forks The round and view numbers of QC for block B are denoted by $QC.r$ and $QC.v$, which equals $B.r$ and $B.v$, respectively. A QC or a block of view number v and round number r has rank $rank = (v, r)$, and QCs or blocks are compared lexicographically by their rank (i.e. first by the view number, then by the round number). The function $\max(rank_1, rank_2)$ returns the higher rank between $rank_1$ and $rank_2$, and $\max(qc_1, qc_2)$ returns the higher (ranked) QC between qc_1 and qc_2. In DiemBFT and `Jolteon` (Section 3) the view number is always 0, so QCs or blocks are compared by their round numbers; while in `Ditto` (section 4) they are compared by their rank. We use qc_{high} to denote the highest quorum certificate.
- *Timeout Certificate.* A timeout message of round r by a replica contains the replica's threshold signature share on r, and its qc_{high}. A timeout certificate (TC) is formed by a quorum of $n - f = 2f + 1$ timeout messages, containing a threshold signature on a round number r produced by combining $2f + 1$ signature shares $\{r\}$ from the `timeout` messages, and $2f + 1$ qc_{high}'s. A valid TC should only contain high-QCs with round numbers $< TC.r$, and this will be checked implicitly when a replica receives a TC.

Performance metrics. We consider message complexity and communication complexity per round and consensus decision. For the theoretical analysis of the latency, we consider block-commit latency, i.e., the number of rounds for all honest replicas to commit a block since it is proposed. For the empirical analysis, we measure the end-to-end latency, i.e., the time to commit a transaction since it is sent by a client.

Description of DiemBFT. The DiemBFT protocol (also known as LibraBFT) [36] is a production version of HotStuff [38] with a synchronizer implementation (Pacemaker). For better readability, Figure 1 presents the DiemBFT protocol in our terminology, which we will refer to as HotStuff or DiemBFT interchangeably throughout the paper There are two components of DiemBFT, a *Steady State protocol* that makes progress when the round leader is honest, and a *Pacemaker protocol* that advances round numbers either due to the lack of progress or due to the current round being completed. The leader L_r, upon entering round r, proposes a block B that extends a block certified by the highest QC it knows about, qc_{high}. When receiving the first valid round-r block from L_r, any replica tries to advance its current round number, update its highest locked round and its highest QC, and checks if any block can be committed. A block can be committed if it is the first block among 3 adjacent certified blocks with consecutive

[1] For example, clients can send their transactions to all replicas, and the leader can propose transactions that are not yet included in the blockchain, in the order that they are submitted. With rotating leaders of HotStuff/DiemBFT and random leader election of the asynchronous fallback, the assumption can be guaranteed.

Let L_r be the leader of round r. Each replica keeps the highest voted round r_{vote}, the highest locked round r_{lock} [a], current round number r_{cur}, and the highest quorum certificate qc_{high} (the current view number v_{cur} is not used and remains 0 throughout). Replicas initialize $r_{vote} = 0$, $r_{lock} = 0$, $r_{cur} = 1$, qc_{high} as the QC of the genesis block of round 0, and enter round 1.

Steady State Protocol for Replica i

- **Propose.** Upon entering round r, the leader L_r multicasts a block $B = [id, qc_{high}, r, v_{cur}, txn]$.
- **Vote.** Upon receiving the first proposal $B = [id, qc, r, v, txn]$ from L_r in round r, execute *Advance Round*, *Lock*, and then *Commit* (defined below). If $r = r_{cur}$, $v = v_{cur}$, $r > r_{vote}$ and $qc.r \geq r_{lock}$, vote for B by sending the threshold signature share $\{id, r, v\}_i$ to L_{r+1}, and update $r_{vote} \leftarrow r$.
- **Lock.** (2-chain lock rule) Upon observing any qc [b], let qc' be the QC contained in the block certified by qc (i.e., qc' is the parent of qc), the replica updates $r_{lock} \leftarrow \max(r_{lock}, qc'.r)$, and $qc_{high} \leftarrow \max(qc_{high}, qc)$.
- **Commit.** (3-chain commit rule) Whenever there exist three adjacent certified blocks B, B', B'' in the chain with consecutive round numbers, i.e., $B''.r = B'.r + 1 = B.r + 2$, the replica commits B and all its ancestors.

Pacemaker Protocol for Replica i

- **Advance Round.** The replica updates its current round $r_{cur} \leftarrow \max(r_{cur}, r)$, iff
  - the replica receives or forms a round-$(r-1)$ quorum certificate qc, or
  - the replica receives or forms a round-$(r-1)$ timeout certificate tc.
- **Timer and Timeout.**
  - Upon entering round r, the replica sends the round-$(r-1)$ TC to L_r if it has the TC, and resets its timer to count down for a predefined time interval (timeout τ).
  - When the timer expires, the replica stops voting for round r_{cur} and multicasts a `timeout` message $\langle \{r_{cur}\}_i, qc_{high} \rangle_i$ where $\{r_{cur}\}_i$ is a threshold signature share.
  - Upon receiving a `timeout` message or TC, execute *Advance Round*, *Lock*, and then *Commit*.
  - Upon receiving 2f+1 timeouts, form a TC.

[a] Corresponds to the *preferred round* in [36].

[b] May be formed from votes (by a leader) or contained in a proposal or a timeout messsage.

Figure 1: DiemBFT in our terminology.

round numbers. After the above steps, the replica votes for B by sending a threshold signature share to the next leader L_{r+1}, if the voting rules are satisfied. Then, when the next leader L_{r+1} receives $2f+1$ such votes, it forms a QC of round r, enters round $r+1$, proposes the block for that round, and the above process is repeated. When the timer of some round r expires, the replica stops voting for that round and multicast a timeout message containing a threshold signature share for r and its highest QC. When any replica receives $2f+1$ such timeout messages, it forms a TC of round r, enters round $r+1$ and sends the TC to the (next) leader L_{r+1}. When any replica receives a timeout or a TC, it tries to advance its current round number given the high-QCs (in the timeout or TC) or the TC, updates its highest locked round and its highest QC given the high-QCs, and checks if any block can be committed. For space limitation, we omit the correctness proof of the DiemBFT protocol, which can be found in [36].

3 `Jolteon` Design

In this section, we describe how we turn DiemBFT into `Jolteon` – a 2-chain version of DiemBFT. The pseudocode is given in Figure 2. As mentioned previously, the quadratic cost of view-synchronization in leader-based consensus protocols, due to faulty leaders or asynchronous periods, is inherent. While the linearity of HotStuff's view-change is a theoretical milestone, its practical importance is limited by this anyway quadratic cost of synchronization after bad views.

With this insight in mind, `Jolteon` uses a quadratic view-change protocol that allows a linear 2-chain commit rule in the steady state. The idea is inspired by PBFT [9] with each leader proving the safety of its proposal. In the steady state each block extends the block from the previous round and providing the QC of the parent is enough to prove safety, hence the steady state protocol remains linear. However, after a bad round caused by asynchrony or a bad leader, proving the safety of extending an older QC requires the leader to prove that nothing more recent than the block of that QC is committed. To prove this, the leader uses the TC formed for view-changing the bad round. Recall that a TC for round r contains $2f+1$ validators' qc_{high} sent in timeout messages for round r. The leader attaches the TC to its proposal in round $r+1$ and extends the highest QC among the QCs in the TC (see the **Propose** rule in Figure 2).

Replicas keep the same variables as DiemBFT in Figure 1.

Steady State Protocol for Replica i

Changes from DiemBFT in Figure 1 are marked in blue.

- **Propose.** Upon entering round r, the leader L_r multicasts a block $B = [id, qc_{high}, tc, r, v_{cur}, txn]$, where $tc = tc_{r-1}$ if L_r enters round r by receiving a round-$(r-1)$ tc_{r-1}, and $tc = \perp$ otherwise.
- **Vote.** Upon receiving the first proposal $B = [id, qc, tc, r, v, txn]$ from L_r, execute *Advance Round*, *Lock*, and then *Commit* (defined below). If $r = r_{cur}$, $v = v_{cur}$, $r > r_{vote}$ and ((1) $r = qc.r + 1$, or (2) $r = tc.r + 1$ and $qc.r \geq \max\{qc_{high}.r \mid qc_{high} \in tc\}$), vote for B by sending the threshold signature share $\{id, r, v\}_i$ to L_{r+1}, and update $r_{vote} \leftarrow r$.
- **Lock.** (1-chain lock rule) Upon seeing a qc (formed by votes or contained in proposal or timeouts), the replica updates $qc_{high} \leftarrow \max(qc_{high}, qc)$.
- **Commit.** (2-chain commit rule) Whenever there exists two adjacent certified blocks B, B' in the chain with consecutive round numbers, i.e., $B'.r = B.r + 1$, the replica commits B and all its ancestors.

Pacemaker Protocol for Replica i

Identical to DiemBFT in Figure 1.

Figure 2: `Jolteon`.

	Latency	steady state communication	view-change communication	view synchronization
DiemBFT	7 messages	linear	linear	quadratic
`Jolteon`	5 messages	linear	quadratic	quadratic

Table 2: Theoretical comparison between DiemBFT and `Jolteon`.

When a validator gets a proposal B, it first tries to advance its round number, then updates its qc_{high} with $B.qc$ and checks the 2-chain commit rule for a possible commit. Then, before voting, it verifies that at least one of the following two conditions is satisfied:

- $B.r = B.qc.r + 1$ or;
- $B.r = B.tc.r + 1$ and $B.qc.r \geq \max\{qc_{high}.r \mid qc_{high} \in B.tc\}$

In other words, either B contains the QC for the block of the previous round; or it contains at least the highest QC among the $2f+1$ QCs in the attached TC, which was formed to view-change the previous round.

Safety intuition. If the first condition is satisfied then B directly extends the block from the previous round. Since at most one QC can be formed in a round, this means that no forks are possible, and voting for B is safe.

The second condition is more subtle. Note that by the 2-chain commit rule, if a block B' is committed, then there exists a certified block B'' s.t. $B'.round + 1 = B''.round$. That is, at least $f+1$ honest replicas vote to form the QC for B'' and thus set their qc_{high} to be the QC for block B' ($qc_{B'}$). By quorum intersection and since replicas never decrease their qc_{high}, any future (higher round) TC contains a qc_{high} that is at least as high as $qc_{B'}$. The second condition then guarantees that honest replicas only vote for proposals that extend the committed block B'. Full proof can be found in Appendix A.

3.1 DiemBFT vs `Jolteon`.

Efficiency. Table 2 compares the efficiency of DiemBFT and `Jolteon` from a theoretical point of view. Both protocols have linear communication complexity per round and per decision *under synchrony and honest leaders*, due to the leader-to-all communication pattern and the threshold signature scheme[2]. The complexity of the Pacemaker to synchronize views (view synchronization in Table 2), for both protocols, under asynchrony or failures is quadratic due to the all-to-all timeout messages. The complexity of proposing a block after a bad round that requires synchronization (view-change communication in Table 2) is linear for DiemBFT and quadratic for `Jolteon`. This is because in DiemBFT such a proposal only includes qc_{high}, whereas in `Jolteon` it includes a TC containing $2f+1$ qc_{high}. The block-commit latency *under synchrony and honest leaders* is 7Δ and 5Δ for DiemBFT and `Jolteon`, respectively, due to the 3-chain (2-chain for `Jolteon`) commit rules. Each round in the commit chain requires two rounds trip times (upper bounded by Δ), plus the new leader multicast the last QC of the chain that allows all honest replicas to learn about the chain and commit the block.

Limitations. During periods of asynchrony, or when facing DDoS attacks on the leaders, both protocols have *no liveness guarantees* – the leaders' blocks cannot be received on time.

[2]The implementation of DiemBFT does not use threshold signatures, but for the theoretical comparison here we consider a version of DiemBFT that does.

As a result, replicas keep multicasting timeout messages and advancing round numbers without certifying or committing blocks. This is unavoidable [34]: communication complexity of any deterministic partially synchronous Byzantine agreement protocol is unbounded before GST, even in failure-free executions.

Fortunately, in the next section, we show that it is possible to boost the liveness guarantee of DiemBFT, `Jolteon`, by replacing the view-synchronization mechanism (pacemaker) with a fallback protocol that guarantees progress even under asynchrony. Furthermore, the asynchronous fallback can be efficient. The protocol we propose in the next section has quadratic communication cost for fallback, which is the cost DiemBFT and `Jolteon` pay to synchronize views anyway.

4 `Ditto` Design

To strengthen the liveness guarantees of existing partially synchronous BFT protocols such as DiemBFT [36] and `Jolteon`, we propose the protocol `Ditto`. `Ditto` has linear communication cost for the synchronous path, quadratic cost for the asynchronous path, and preserves liveness robustly in asynchronous network conditions. The `Ditto` protocol is presented in Figure 3.

Our `Ditto` protocol uses Multi-valued Validated Byzantine Agreement (MVBA) in a black-box way.

Multi-valued Validated Byzantine Agreement Multi-valued validated Byzantine agreement (MVBA) [7] is a Byzantine fault-tolerant agreement protocol where a set of protocol replicas each with an input value can agree on the same value satisfying a predefined external predicate $f(v) : \{0,1\}^{|v|} \rightarrow \{0,1\}$ globally known to all the replicas. An MVBA protocol with predicate $f(\cdot)$ should provide the following guarantees except for negligible probability.

- **Agreement**: All honest replicas output the same value.
- **External Validity**: If an honest replica outputs v, then v must be externally valid, i.e., $f(v) = 1$.
- **Termination**: If all honest replicas input an externally valid value, all honest replicas eventually output.

Our protocol uses MVBA with the state-aware predicate [39], where the predicate $f(v,e)$ can have an additional state-dependent variable e chosen by the replica when invoking the MVBA. With the state-aware predicate, a value v is externally valid to an honest replica, if and only if $f(v,e) = 1$ where e is the state-dependent variable chosen by the honest replica when invoking the MVBA. The guarantees of MVBA with the state-aware predicate become the following.

- **Agreement**: All honest replicas output the same value.
- **External Validity**: If an honest replica outputs v, then v must be externally valid to at least one honest replica.
- **Termination**: If all honest replicas input a value that is externally valid to all honest replicas, all honest replicas eventually output.

Our protocol `Ditto` can directly use existing MVBA protocols in a black-box manner, by plugging in the state-aware predicate as defined in Figure 3. We refer the reader to [39] for the argument as to why the agreement, termination, and external validity properties of MVBA still hold.

Protocol intuition. Our solution consists of a steady-state protocol, which is similar to that of `Jolteon`, and an asynchronous fallback protocol, which replaces the view-change of `Jolteon`. The idea behind our fallback protocol is that, after entering the fallback, all replicas will invoke MVBA to commit a new block extending the committed blocks. Then, replicas will try the steady state again from the block committed by MVBA.

Since this protocol has two paths, a synchronous fast path and an asynchronous fallback path, it is critical to ensure safety and liveness when the protocol transfers from one path to another. On a high level, our protocol ensures safety by always following the 1-chain lock and 2-chain commit rule from `Jolteon`, and by leveraging the agreement guarantee of MVBA. As for liveness, our protocol guarantees that either the sync path (same as `Jolteon`) makes progress, or enough replicas timeout the synchronous path and enter the asynchronous fallback.

Description of Steady State. All the steps are described in an event-driven manner, i.e., a step is executed whenever a condition is satisfied during the protocol execution. The protocol is similar to `Jolteon` with the following main differences. The blocks do not contain TC anymore. Each replica additionally keeps a boolean value `fallback` to record if it is in the fallback. When the replica reaches timeout for a round in view v, it stops the timer and stops proposing or voting for any block of view $\leq v$. The 1-chain lock rule and 2-chain commit rule still apply, but the two blocks in the 2-chain need to have the same view number.

Description of Fallback. Now we give a brief description of the Fallback protocol, which replaces the Pacemaker protocol in the `Jolteon` protocol (Figure 2).

Just like in `Jolteon`, when the timer expires, the replica tries to initiate the fallback (the equivalent of view-change) by broadcasting a timeout message containing the highest QC and a signature share of the current view number. When receiving $2f+1$ timeout messages of the same view v, the replica enters the fallback, updates its current view number, and multicasts a `proof` message attaching the view number and its high-QC. On receiving a `proof` message with qc, the replica threshold-signs the message if qc is no lower than its qc_{high}, otherwise sends back its qc_{high} with a higher rank. Then, the replica that sent `proof`, waits for either `proof` getting threshold-signed by $2f+1$ replicas, or receiving a higher-ranked QC. In the former case, the replica inputs a

In addition to the variables in Figure 2, each replica also keeps a boolean value `fallback`, initialized as $false$, to specify whether the replica is in a fallback.

Steady State Protocol for Replica i

- **Propose.** Upon entering round r, the leader L_r multicasts a block $B = [id, qc_{high}, r, v_{cur}, txn]$.
- **Vote.** Upon receiving the first proposal $B = [id, qc, r, v, txn]$ from L_r. If $r = r_{cur}$, $v = v_{cur}$, $r > r_{vote}$ and $r = qc.r + 1$, vote for B by sending the threshold signature share $\{id, r, v\}_i$ to L_{r+1}, and update $r_{vote} \leftarrow r$.
- **New QC.** Upon receiving a qc, replica i performs the following steps.
  - **Advance Round.** Replica i updates its current round $r_{cur} \leftarrow \max(r_{cur}, qc.r + 1)$.
  - **Lock.** (1-chain lock rule) Replica i updates $qc_{high} \leftarrow \max(qc_{high}, qc)$.
  - **Commit.** (2-chain commit rule) If there exists two adjacent certified blocks B, B' with the same view number, replica i commits B and all its ancestors.
- **Timeout.** Upon entering a new round, replica i resets its timer to τ.
  When the timer expires, replica i stops the timer as well as proposing or voting for any block of view $\leq v_{cur}$, and multicasts a `timeout` message $\langle \{v_{cur}\}_i, qc_{high} \rangle_i$ where $\{v_{cur}\}_i$ is a threshold signature share.

Fallback Protocol for Replica i

Replica i executes the following in event-driven manner.

- **Enter Fallback.** Upon receiving $2f + 1$ `timeout` messages of same view $v \geq v_{cur}$, replica i updates `fallback` $\leftarrow true$, $v_{cur} \leftarrow v + 1$, and multicasts a message $(\texttt{proof}, v_{cur}, qc_{high})$.
- **Ack.** When `fallback` $= true$, upon receiving the first message $(\texttt{proof}, v, qc)$ from replica j where $v = v_{cur}$,
  - if $qc.rank \geq qc_{high}.rank$, replica i sends $\{\texttt{proof}, v_{cur}, qc\}_i$ back to replica j;
  - if $qc.rank < qc_{high}.rank$, replica i sends $(\texttt{higherQC}, v_{cur}, qc_{high})$ back to replica j;
- **MVBA.** When `fallback` $= true$, and replica has not invoked $\texttt{MVBA}_{v_{cur}}$ yet,
  - upon aggregating a threshold signature σ on message $(\texttt{proof}, v_{cur}, qc_{high})$, or
  - upon receiving $(\texttt{higherQC}, v_{cur}, qc)$ where $qc.rank > qc_{high}.rank$, replica i updates $qc_{high} \leftarrow qc$ and sets $\sigma \leftarrow \perp$,

  replica i generates block $B = [id, qc_{high}, qc_{high}.r + 1, v_{cur}, txn]$ and invokes $\texttt{MVBA}_{v_{cur}}(B, \sigma)$ with predicate $f((B', \sigma'), qc_{high})$ defined below, where (B', σ') is the input of a replica.
- **Exit Fallback.** Upon $\texttt{MVBA}_v$ decides B where $v \geq v_{cur}$, replica i commits B and all its ancestors, multicasts the threshold signature share $\{B.id, B.r, B.v\}_i$, and updates $v_{cur} \leftarrow v, r_{vote} \leftarrow B.r,$ `fallback` $\leftarrow false$. Replica i then waits until B gets certified.

Predicate $f((B', \sigma'), qc_{high})$ of $\texttt{MVBA}_v$ (as replica i with qc_{high} in view v when invoking `MVBA`)

- returns 1, if $B'.v = v$, $B'.qc.v = v - 1$, $B'.r = B'.qc.r + 1$ and either (1) σ' is a valid threshold signature on the message $(\texttt{proof}, B'.v, B'.qc)$ or (2) $\sigma' = \perp$ and $B'.qc.rank \geq qc_{high}.rank$;
- returns 0 otherwise.

Figure 3: `Ditto`

block extending its qc_{high} together with the threshold signature as the proof to the `MVBA` instance of the current view. In the latter case, the replica inputs a block extending the received higher-ranked QC together with empty proof to the `MVBA` instance. In both cases, the replica invokes `MVBA` with the state-aware predicate which also has its qc_{high} as input.

When `MVBA` decides a block, the replica commits the block and its ancestors, multicasts a vote message to get the block certified, and exits the fallback. The replica waits until the block gets certified before executing other steps of the protocol.

Correctness of `Ditto`. The proof of safety and liveness for `Ditto` can be found in Appendix B.

Theorem 1 (Efficiency)**.** *During the periods of synchrony with honest leaders, the amortized communication complexity per block decision is $O(n)$, and the block-commit latency is* 5 *rounds. During periods of asynchrony, the expected communication complexity per block decision is $O(n^2)$, and the expected block-commit latency is* 13.5 *rounds.*

Proof. When the network is synchronous and leaders are honest, no honest replica will multicast timeout messages. In every round, the designated leader multicast its proposal of size $O(1)$ (due to the use of threshold signatures for QC), and all honest replicas send the vote of size $O(1)$ to the next leader. Hence the communication cost is $O(n)$ per round and per block decision. For the block latency, since `Jolteon` adopts 2-chain commit and need one more round for all replicas to receive the 2-chain proof, the latency is $2 \times 2 + 1 = 5$ rounds.

When the network is asynchronous and honest replicas enter the asynchronous fallback, each honest replica in the

fallback only broadcast $O(1)$ number of messages, and each message has size $O(1)$. Hence, each instance of the asynchronous fallback has communication cost $O(n^2)$, and will commit a new block with probability $2/3$. Therefore, the expected communication complexity per block decision is $O(n^2)$.

The latency to commit a block under fallback is 13.5 rounds, consisting of 1 round to exchange timeouts, 2 rounds to multicast and ack `proof` message, 9.5 rounds finish `MVBA`, and 1 round to get `MVBA` output certified. The latency can be further reduced to 12.5 rounds by removing the round to certify the `MVBA` output, if the `MVBA` already certifies its output. □

5 Implementation and Evaluation

[Note to the reader: The implementation and evaluation of 2-chain VABA, and the asynchronous fallback path of `Ditto` are outdated. The implementation and evaluation of `Jolteon`, and the synchronous path of `Ditto` are up to date.]

5.1 Implementation

We implement `Jolteon` and `Ditto` on top of a high-performance open-source implementation of HotStuff[3] [38]. We selected this implementation because it implements a Pacemaker [38], contrarily to the implementation used in the original HotStuff paper[4]. Additionally, it provides well-documented benchmarking scripts to measure performance in various conditions, and it is close to a production system (it provides real networking, cryptography, and persistent storage). It is implemented in Rust, uses Tokio[5] for asynchronous networking, ed25519-dalek[6] for elliptic curve based signatures, and data-structures are persisted using RocksDB[7]. It uses TCP to achieve reliable point-to-point channels, necessary to correctly implement the distributed system abstractions. We additionally use `threshold_crypto`[8] to implement random coins, Our implementations are between 5,000 and 7,000 LOC, and a further 2,000 LOC of unit tests. We are open sourcing our implementations of `Jolteon`[9] , and `Ditto` and 2-chain VABA[10]. We are also open sourcing all AWS orchestration scripts, benchmarking scripts, and measurements data to enable reproducible results [11].

`Ditto` with exponential backoff: From the protocol design of 2-chain VABA, we know that 2-chain VABA is exactly the asynchronous fallback of our `Ditto` with the timeout threshold τ set to be 0. Therefore, under asynchrony or leader attacks, the performance of 2-chain VABA would be better than `Ditto`, as 2-chain VABA immediately proceeds to the next view without waiting for the timer to expire. To improve the latency performance of `Ditto` under long periods of asynchrony or leader attacks, we adopt an exponential backoff mechanism for the asynchronous fallback as follows. We say a replica executes the asynchronous fallback consecutively x times if it only waits for the timer to expire for the first fallback, and skips waiting for the timer and immediately sends timeout for the rest $x-1$ fallbacks. Initially, replicas only execute asynchronous fallback consecutively $x = 1$ time. However, if a replica, within the timeout, does not receive from the steady state round-leader immediately after the fallback, it will multiply x by a constant factor (5 in our experiments); otherwise, the replica resets $x = 1$. Therefore, during long periods of asynchrony or leader attacks, the number of consecutively executed fallbacks would be exponentially increasing $(1, 5, 25, ...)$; while during periods of synchrony and honest leaders, the number of consecutively executed fallbacks is always 1.

5.2 Evaluation

We evaluate the throughput and latency of our implementations through experiments on Amazon Web Services (AWS). We particularly aim to demonstrate (i) that `Jolteon` achieves the theoretically lower block-commit latency than 3-chain DiemBFT under no contention and (ii) that the theoretically larger message size during view-change does not impose a heavier burden, making `Jolteon` no slower than 3-chain DiemBFT under faults (when the view-change happens frequently). Additionally we aim to show that `Ditto` adapts to the network condition, meaning that (iii) it behaves similarly to `Jolteon` when the network is synchronous (with and without faults) and (iv) close to our faster version of VABA (2-chain) when the adversary adaptively compromises the leader.

We deploy a testbed on Amazon Web Services, using m5.8xlarge instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (apsoutheast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast1). They provide 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, and 128GB memory and run Linux Ubuntu server 20.04.

We measure throughput and end-to-end latency as the performance metrics. Throughput is computed as the average number of committed transactions per second, and end-to-end latency measures the average time to commit a transaction from the moment it is submitted by the client. Compared with the block-commit latency in our theoretical analysis, end-to-end latency also includes the queuing delay of the transaction

[3] https://github.com/asonnino/hotstuff/tree/3-chain
[4] https://github.com/hot-stuff/libhotstuff
[5] https://tokio.rs
[6] https://github.com/dalek-cryptography/ed25519-dalek
[7] https://rocksdb.org
[8] https://docs.rs/threshold_crypto/0.4.0/threshold_crypto/
[9] https://github.com/asonnino/hotstuff
[10] https://github.com/danielxiangzl/hotstuff
[11] https://github.com/asonnino/hotstuff/tree/main/benchmark, https://github.com/danielxiangzl/hotstuff/tree/main/benchmark

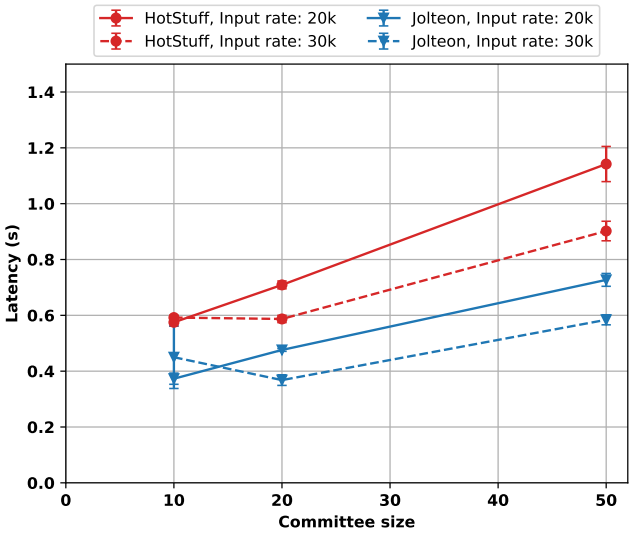

Figure 4: Comparative block-commit latency for 3-chain DiemBFT (HotStuff) and Jolteon. WAN measurements with 10, 20, or 50 replicas. No replica faults, 500KB mempool batch size and 512B transaction size.

when the clients' input rate is high which helps identify the capacity limit of our system.

In all our experiments, the transaction size is set to be 512 bytes and the mempool batch size is set to be 500KB. We deploy one benchmark client per node submitting transactions at a fixed rate for a duration of 5 minutes (to ensure we report steady state performance). We set the timeout to be 5 seconds for experiments with 10 and 20 nodes, and 10 seconds for 50 nodes, so that the timeout is large enough for not triggering the pacemaker of Jolteon and fallback of Ditto. In the following sections, each measurement in the graphs is the average of 3 runs, and the error bars represent one standard deviation.

5.3 Evaluation of Jolteon

In this section, we compare Jolteon with our baseline 3-chain DiemBFT implementation in two experiments. First in Figure 4 we run both protocol with a varying system size (10, 20, 50 nodes). In order to remove any noise from the mempool, this graph does not show the end-to-end latency for clients but the time it takes for a block to be committed. As the Figure illustrated Jolteon consistently outperforms 3-chain DiemBFT by about $200-300$ms of latency which is around one round-trip across the world and both systems scale similarly. In Figure 5 this effect is less visible due to the noise of the mempool (end-to-end latency of around 2 secs), but Jolteon is still slightly faster than 3-chain DiemBFT in most experiments.

Finally, in Figure 6 we run both protocols with 20 nodes and crashed 0, 1, or 3 nodes at the beginning of the experiment. This forces frequent view-changes due to the leader rotation that 3-chain DiemBFT uses in order to provide fairness. This is ideal for our experiment since we can see the overall impact of the more costly view-change slowing down Jolteon. As we can see, Jolteon again outperforms 3-chain DiemBFT in most settings due to the 2-chain commit enabling more frequent commits under faults. As a result, we can conclude that there is little reason to pay the extra round-trip of 3-chain DiemBFT in order to have a theoretically linear view-change. After sharing our findings with the Diem Team they are currently integrating an adaptation of Jolteon for their next mainnet release.

5.4 Evaluation of Ditto

Synchronous and fault-free executions. When all replicas are fault-free and the network is synchronous, we compare the performances of the three protocol implementations in Figure 5. As we can observe from the figure, the synchronous path performance of Ditto is very close to that of Jolteon, when the quadratic asynchronous fallback of Ditto and the quadratic pacemaker of Jolteon is not triggered. On the other hand, the performance of 2-chain VABA is worse than Jolteon and Ditto in this setting, due to its quadratic communication pattern – instead of every replica receiving the block metadata and synchronizing the transaction payload with only one leader per round in Jolteon and Ditto, in VABA every replica will receive and synchronize with $O(n)$ leaders per round.

Crash faults. In this experiment, we run the three protocol implementations with 20 nodes and crash 1 or 3 nodes from the beginning of the execution. The throughput-latency results are summarized in Figure 6. As we can observe from the figure, 2-chain VABA has the most robust performance under faults, where the latency only slightly increases before the throughput exceeds $30k$ tps. The reason is that by design 2-chain VABA can make progress as long as at least $2f+1$ nodes are honest, and crashing some of the nodes only add latency to the views when these nodes are elected as the leader of the views. The performance of 3-chain DiemBFT and Jolteon are more fragile under faults, where the peak performance degrades to $30k$ tps with about 10 seconds latency under 1 fault and $10k$ tps with about 15 seconds latency under 3 faults. The reason is that whenever a round-r leader is faulty, replicas need to wait for two timeouts, which is 10sec since the timeout is set to be 5sec, to enter round $r+1$ from round $r-1$. Ditto under faults performs better than Jolteon but worse than 2-chain VABA. Compared to 2-chain VABA, replicas in Ditto need to wait for a timeout of 5sec to enter the asynchronous fallback whenever the leader is faulty; compared to Jolteon, replicas only need to wait for one timeout (5sec) to enter the fallback, which makes progress efficiently even under faults (as the 2-chain VABA under faults suggests).

Attacks on the leaders. Figure 7 presents the measurement re-

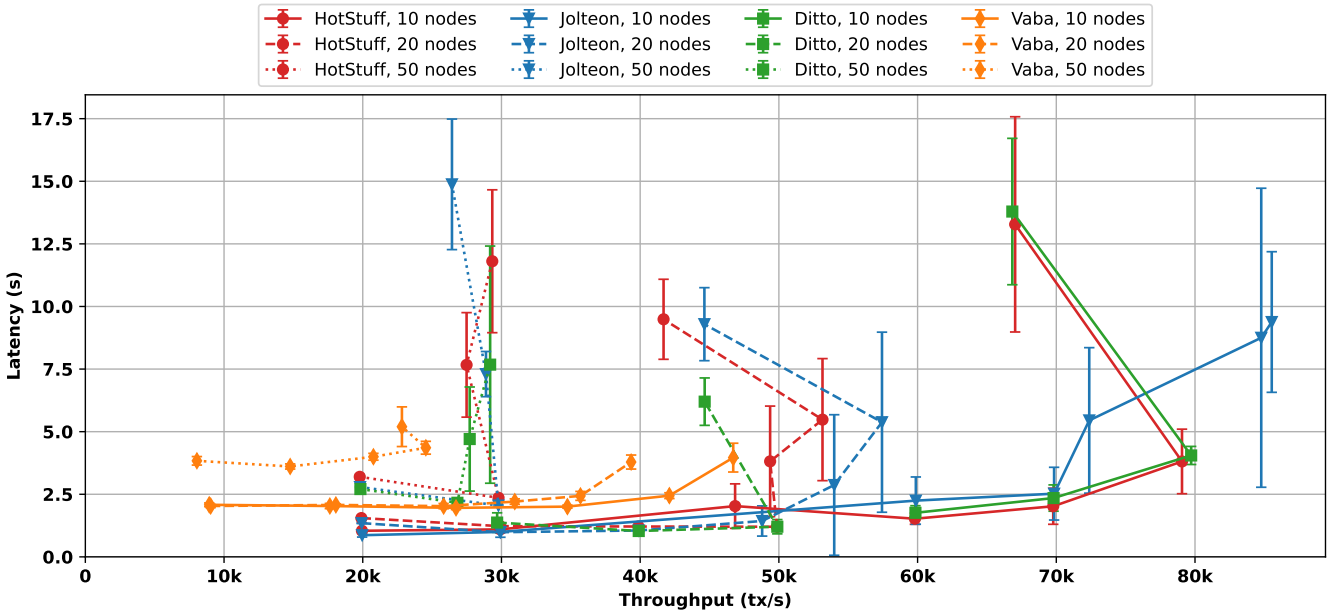

Figure 5: Comparative throughput-latency performance for 3-chain DiemBFT (HotStuff), `Jolteon`, `Ditto`, and 2-chain VABA WAN measurements with 10, 20, or 50 replicas. No replica faults, 500KB mempool batch size and 512B transaction size.

sults. When the eventual synchrony assumption does not hold, either due to DDoS attacks on the leaders or adversarial delays on the leaders' messages, 3-chain DiemBFT and `Jolteon` will have no liveness, i.e., the throughput of the system is always 0. The reason is that whenever a replica becomes the leader for some round, its proposal message is delayed and all other replicas will timeout for that round. On the other hand, `Ditto` and 2-chain VABA are robust against such adversarial delays and can make progress under asynchrony. The performance of the 2-chain VABA protocol implementation is not affected much by delaying a certain replica's proposal, as observed from the previous experiment with crashed replicas. Therefore, we use it as a baseline to compare with our `Ditto` protocol implementation. Our results, confirm our theoretical assumption as the asynchronous fallback performance of `Ditto` is very close to that of 2-chain VABA under 10 or 20 nodes, and slightly worse than 2-chain VABA under 50 nodes. This extra latency cost is due to the few timeouts that are triggered during the exponential back-off.

Crash faults and attacks on the leaders. Figure 8 presents the measurement results. The throughput of 3-chain DiemBFT and `Jolteon` is again 0 under leader attacks for the same reason mentioned above. 2-chain VABA makes progress even under leader attacks, and its performance is also robust against 1 or 3 node crashes. The performance of `Ditto` is very close to that of 2-chain VABA, since our implementation adopts the exponential backoff mechanism for the asynchronous fallback. One interesting artifact of the exponential backoff is that, compared to the case under just crash faults and no DDoS (Figure 6), `Ditto` has better performance under both crash faults and DDoS (Figure 8). The reason is that under long periods of leader attacks, the `Ditto` will skip waiting for the time to expire for most of the views, and directly send timeouts and enter the fallback.

Take away. To conclude, there is little reason not to use `Ditto` as our experiments confirm our theoretical bounds. `Ditto` adapts to the network behavior and achieves almost optimal performance. The only system that sometimes outperforms `Ditto` is 2-chain VABA during intermittent periods of asynchrony as it does not pay the timeout cost of `Ditto` when deciding how to adapt. This, however, comes at a significant cost when the network is good and in our opinion legitimizes the superiority of `Ditto` when run over the Internet.

6 Related Work

Eventually synchronous BFT. BFT SMR has been studied extensively in the literature. A sequence of efforts [5, 6, 9, 15, 19, 38] have been made to reduce the communication cost of the BFT SMR protocols, with the state-of-the-art being HotStuff [38] that has $O(n)$ cost for decisions, a 3-chain commit latency under synchrony and honest leaders, and $O(n^2)$ cost for view-synchronization. `Jolteon` presents another step forward from HotStuff as we realize the co-design of the pacemaker with the commit rules enables removing one round without sacrificing the linear happy path. Two concurrent theoretical works propose a 2-chain variation of the HotStuff as well [17, 31]. However, the work of Rambaud et al. [31]

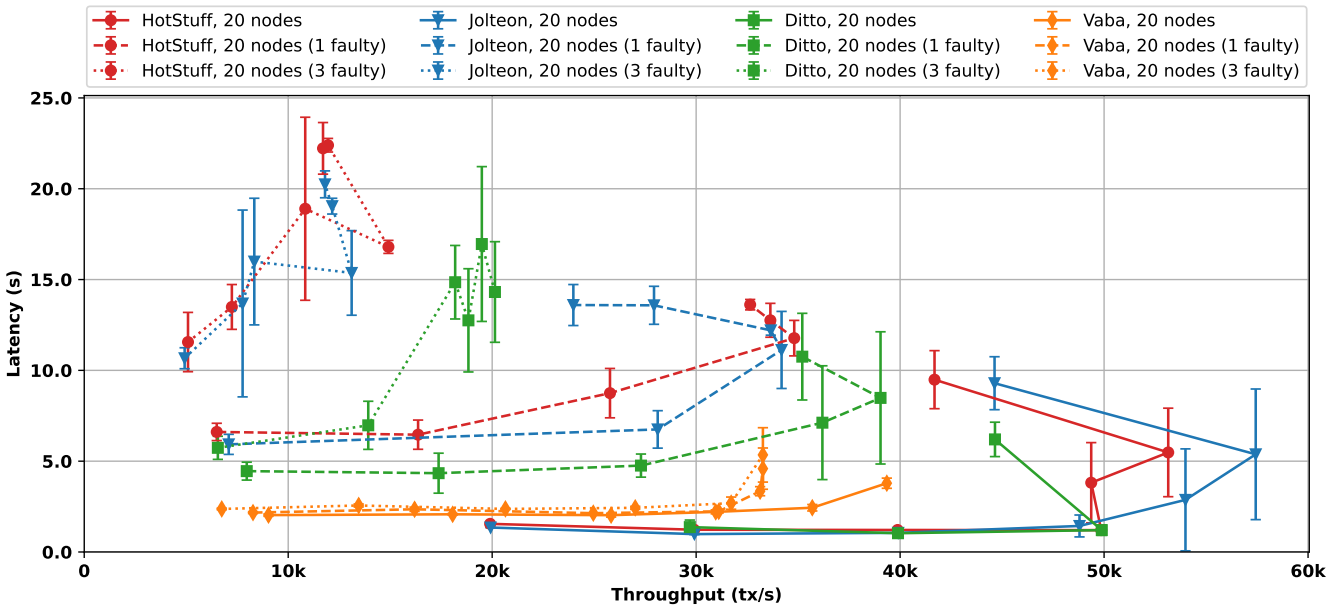

Figure 6: Comparative throughput-latency performance for 3-chain DiemBFT (HotStuff), `Jolteon`, `Ditto`, and 2-chain VABA WAN measurements with 20 replicas. 0, 1, and 3 faults, 500KB mempool batch size and 512B transaction size.

relies on impractical cryptographic primitives to preserve a linear view-change (assuming still a quadratic pacemaker) whereas neither protocol provides a comprehensive evaluation to showcase that the extra view-change costs (which also applies in [17]) does not cause significant overheads. Most importantly, both protocols fail to realize the full power of 2-chain protocols missing the fact the view-change can become robust and DDoS resilient.

Asynchronous BFT. Several recent proposals focus on improving the communication complexity and latency, including HoneyBadgerBFT [26], VABA [3], Dumbo-BFT [16], Dumbo-MVBA [24], ACE [35], Aleph [13], and DAG-Rider [18]. The state-of-the-art protocols for asynchronous SMR have $O(n^2)$ cost per decision [35], or amortized $O(n)$ cost per decision after transaction batching [13, 16, 18, 24].

BFT with optimistic and fallback paths. To the best of our knowledge, [22] is the first asynchronous BFT protocol with an efficient happy path. Their asynchronous path has $O(n^3)$ communication cost while their happy path has $O(n^2)$ cost per decision, which was later extended [30] to an amortized $O(n)$. A recent paper [34] further improved the communication complexity of asynchronous path to $O(n^2)$ and the cost of the happy path to $O(n)$. The latency of these protocols is not optimized, e.g. latency of the protocol in [34] is $O(n)$. Moreover, these papers are theoretical in nature and far from the realm of practicality.

Finally, a concurrent work named the Bolt-Dumbo Transformer (BDT) [23], proposes a BFT SMR protocol with both synchronous and asynchronous paths and provides implementation and evaluation. BDT takes the straightforward solution of composing three separate consensus protocols as black boxes. Every round starts with 1) a partially synchronous protocol (HotStuff), times-out the leader and runs 2) an Asynchronous Binary Agreement in order to move on and run 3) a fully asynchronous consensus protocol [16] as a fallback. Although BDT achieves asymptotically optimal communication cost for both paths this is simply inherited by the already known to be optimal back boxes. On the theoretical side, their design is beneficial since it provides a generally composable framework, but this generality comes at a hefty practical cost. BDT has a latency cost of 7 rounds (vs 5 of `Ditto`) at the fast path and of 45 rounds (vs 13.5 of `Ditto`) at the fallback, making it questionably practical. Finally, not opening the black-boxes stopped BDT from reducing the latency of HotStuff although it also has a quadratic view-change.

Our protocol, `Ditto`, has the asymptotically optimal communication cost in both the happy path and the asynchronous (fallback) path, but also good latency: in fact, due to the 2-chain design, the latency is even better than the state-of-the-art, (3-chain) HotStuff/DiemBFT and (3-chain) VABA protocols. `Ditto` also inherits the efficient pipelined design of these protocols (that are deployed in practice [36]). In contrast with BDT, there is no overhead associated with switching between synchronous and asynchronous paths. We provide an implementation that organically combines the different modes of operation and extensive evaluation to support our claim that `Ditto` enjoys the best of both worlds with practical performance.

Another related line of work on optimistic BFT protocols for other network models, including partially syn-

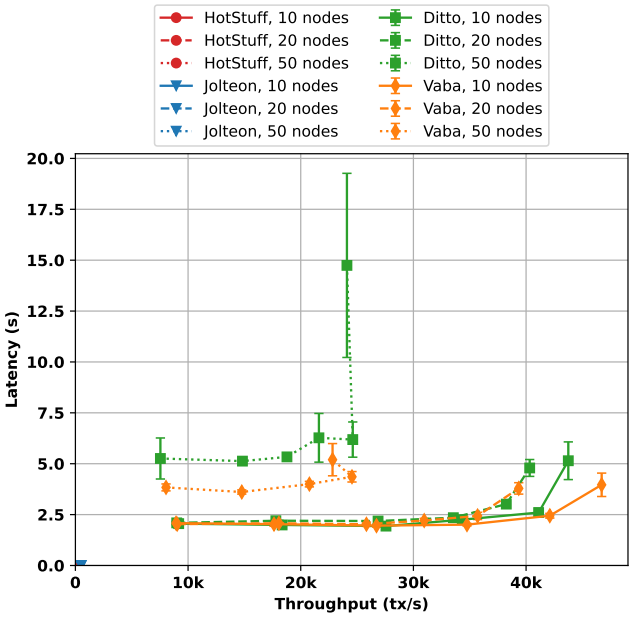

Figure 7: Comparative throughput-latency performance for 3-chain DiemBFT (HotStuff), `Jolteon`, `Ditto`, and 2-chain VABA. WAN measurements with 10, 20, or 50 replicas. No replica faults, 500KB mempool batch size, and 512B transaction size. Leader constantly under DoS attack.

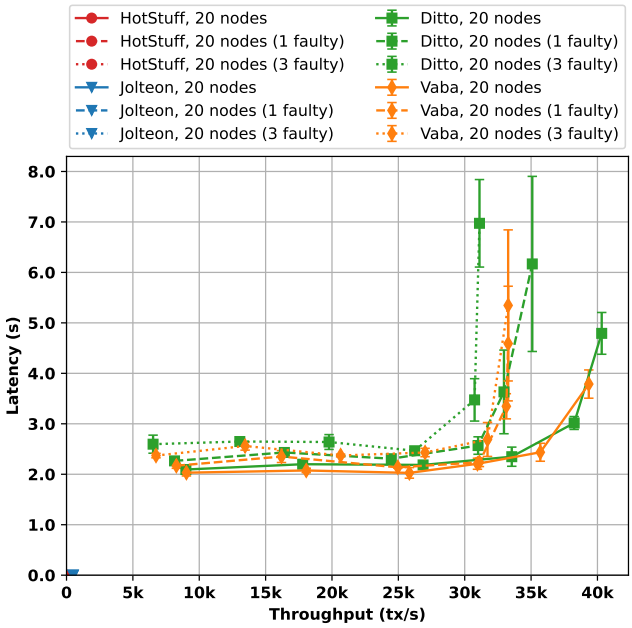

Figure 8: Comparative throughput-latency performance for 3-chain DiemBFT (HotStuff), `Jolteon`, `Ditto`, and 2-chain VABA. WAN measurements with 20 replicas. 0, 1, and 3 faults, 500KB mempool batch size and 512B transaction size. Leader constantly under DoS attack.

chrony [1, 14, 21] and synchrony [10, 27, 29, 33]. We leave the incorporation of an optimistic 1-chain commit path to `Ditto` to future work.

BFT protocols with flexible commit rules. Recently, a line of work studies and proposes BFT protocols with flexible commit rules for different clients of various beliefs on network and resilience assumptions, for the purpose of providing safety and liveness guarantees to the clients with the correct assumptions. As a comparison, our work offers multiple commit paths for replicas under synchrony and asynchrony tolerating the same resilience threshold (one-thirds) and achieves optimal communication cost for each path asymptotically. Flexible BFT [25] proposes one BFT SMR solution supporting clients with various fault and synchronicity beliefs, but the guarantees only hold for the clients with the correct assumptions. Strengthened BFT [37] shows how to strong commit blocks with higher resilience guarantees for partially synchronous BFT SMR protocols. Several recent works [28, 32] investigate how to checkpointing a synchronous longest chain protocol using partially synchronous BFT protocols, and offer clients with two different commit rules.

7 Conclusion and Future Work

We present `Ditto`, a practical byzantine SMR protocol that enjoys the best of both worlds: optimal communication on and off the happy path (linear and quadratic, respectively) and progress guarantees under the worst case asynchrony and DDoS attacks. As a secondary contribution, we design a 2-chain version of HotStuff, `Jolteon`, which leverages a quadratic view-change mechanism to reduce the latency of the standard 3-chain HotStuff. We implement and experimentally evaluate all our systems to validate our theoretical analysis.

An interesting future work would be to incorporate an optimistic 1-chain commit path in our `Jolteon` and `Ditto`, which can further improve the latency performance of our systems when there are no faults.

Acknowledgments

This work is supported by the Novi team at Facebook. We also thank the Novi Research and Engineering teams for valuable feedback, and in particular Mathieu Baudet, Andrey Chursin, George Danezis, Zekun Li, and Dahlia Malkhi for discussions that shaped this work.

We thank the authors of [] to point out flaws of the `Ditto` protocol and its byproduct 2-chain VABA in the previous version of the paper. The `Jolteon` protocol remains secure. For 2-chain VABA, please see [] for the attacks and the proposed protocol that fixes the flaw. For `Ditto`, we provide a fix to the protocol in this version, with higher expected latency ($E(13.5)$) under asynchrony.

References

[1] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the twentieth ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–74, 2005.

[2] Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 317–326, 2019.

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 337–346, 2019.

[4] Erica Blum, Jonathan Katz, and Julian Loss. Network-agnostic state machine replication. *arXiv preprint arXiv:2002.03437*, 2020.

[5] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[6] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[7] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[8] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating Systems Design and Implementation (NSDI)*, pages 173–186. USENIX Association, 1999.

[10] T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. Cryptology ePrint Archive, Report 2018/980, 2018.

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

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

[13] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT)*, pages 214–228, 2019.

[14] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, pages 363–376, 2010.

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

[16] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[17] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and resilient hotstuff protocol. *arXiv preprint arXiv:2010.11454*, 2020.

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

[19] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th Usenix Security Symposium (Usenix Security 16)*, pages 279–296, 2016.

[20] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1751–1767, 2020.

[21] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.

[22] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 204–215. Springer, 2005.

[23] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as pipelined bft. *arXiv preprint arXiv:2103.09425*, 2021.

[24] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 129–138, 2020.

[25] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM Conference on Computer and Communications Security (CCS)*, pages 1041–1053, 2019.

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

[27] Atsuki Momose, Jason Paul Cruz, and Yuichi Kaji. Hybrid-bft: Optimistically responsive synchronous consensus with optimal latency or resilience. Cryptology ePrint Archive, Report 2020/406, 2020.

[28] Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. *arXiv preprint arXiv:2009.04987*, 2020.

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

[30] HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

[31] Matthieu Rambaud. Malicious security comes for free in consensus with leaders. *IACR Cryptol. ePrint Arch.*, 2020:1480, 2020.

[32] Suryanarayana Sankagiri, Xuechao Wang, Sreeram Kannan, and Pramod Viswanath. Blockchain cap theorem allows user-dependent adaptivity and finality. *arXiv preprint arXiv:2010.13711*, 2020.

[33] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 839–857, 2020.

[34] Alexander Spiegelman. In search for a linear byzantine agreement. *arXiv preprint arXiv:2002.06993*, 2020.

[35] Alexander Spiegelman and Arik Rinberg. Ace: Abstract consensus encapsulation for liveness boosting of state machine replication. In *23rd International Conference on Principles of Distributed Systems (OPODIS)*, 2020.

[36] The LibraBFT Team. State machine replication in the libra blockchain, 2020. `https://developers.libra.org/docs/state-machine-replication-paper`.

[37] Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Strengthened fault tolerance in byzantine fault tolerant replication. In *The 41st IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2021.

[38] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–356, 2019.

[39] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. Long live the honey badger: Robust asynchronous {DPSS} and its applications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5413–5430, 2023.

A Correctness of `Jolteon`

A.1 Safety

We begin by formalizing some notation.

- We call a block byzantine (honest) if it was proposed by a byzantine (honest) replica.
- We say that a block B is *certified* if a quorum certificate QC_B exists.
- $B_i \longleftarrow QC_i \longleftarrow B_{i+1}$ means that the block B_i is certified by the quorum certificate QC_i which is contained in the block B_{i+1}.
- $B_i \longleftarrow^* B_j$ means that the block B_j *extends* the block B_i. That is, there is exists a sequence $B_i \longleftarrow QC_i \longleftarrow B_{i+1} \longleftarrow QC_{i+1} \cdots \longleftarrow QC_{j-1} \longleftarrow B_j$

Definition 1 (Global direct-commit)**.** *We say that a block B is* globally direct-committed *if $f+1$ honest replicas each successfully perform the **Vote** step on block B' proposal in round $B.r+1$, such that $B'.qc$ certifies B. These **Vote** calls invoke **Lock**, setting $qc_{high} \leftarrow B'.qc$, and return $f+1$ matching votes (that could be used to form a $QC_{B'}$ with f other matching votes).*

Lemma 1. *If an honest replica successfully performs the **Commit** step on block B then B is globally direct-committed.*

Proof. By the **Commit** condition, there exists a chain $B \longleftarrow QC_B \longleftarrow B' \longleftarrow QC_{B'}$ with $B'.r = B.r + 1$. The existence of $QC_{B'}$ implies that $f+1$ honest replicas did **Vote** for B'. □

The next lemma follows from the voting rules and the definition of global direct commit.

Lemma 2. *If a block B is globally direct-committed then any higher-round TC contains qc_{high} of round at least B.r.*

Proof. By Definition 1, $f+1$ honest replicas execute the **lock** step in round $B.r+1$ and set qc_{high} to $B'.qc$ that certifies B (so $B'.qc.r = B.r$, B' is the block proposed in round $B.r+1$). None of these honest replicas may have previously timed out in round $B.r+1$, and timing out stops voting in a round (but the replicas voted for B').

Since qc_{high} is never decreased, a timeout message prepared by any of the above $f+1$ honest replicas in rounds $> B.r$ contains a high qc of round at least $B.r$. By quorum intersection, timeout messages used to prepare the TC in any round $> B.r$ contain a message from one of these honest replicas, completing the argument. □

Due to quorum intersection, we have

Observation 1. *If a block is certified in a round, no other block can gather $f+1$ honest votes in the same round. Hence, at most one block is certified in each round.*

We can now prove the key lemma

Lemma 3. *For every certified block B' s.t. $B'.r \geq B.r$ such that B is globally direct-committed, $B \longleftarrow^* B'$.*

Proof. By Observation 1, $B \longleftarrow^* B'$ for every B' s.t. $B'.r = B.r$.

We now prove the lemma by induction on the round numbers $r' > B.r$.

Base case: Let $r' = B.r+1$. B is globally direct-committed, so by Definition 1, there are $f+1$ honest replicas that prepare votes in round $r' = B.r+1$ on some block B_{r+1} such that $B \longleftarrow QC_B \longleftarrow B_{r+1}$. By Observation 1, only B_{r+1} can be certified in round r'.

Step: We assume the Lemma holds up to round $r'-1 > B.r$ and prove that it also holds for r'. If no block is certified at round r', then the induction step holds vacuously. Otherwise, let B' be a block certified in round r' and let $QC_{B'}$ be its certificate. B is globally direct-committed, so by Definition 1, there are $f+1$ honest replicas that have locked high qc in round $B.r+1$. One of these replicas, v, must also have prepared a vote that is included in $QC_{B'}$ (as QC formation requires $2f+1$ votes and there are $3f+1$ total replicas).

Let $B'' \longleftarrow QC_{B''} \longleftarrow B'$ and denote $r'' = B''.r = QC_{B''}.r$. There are two cases to consider, $r'' \geq r$ and $r'' < r$. In the first case, by the induction assumption for round r'', $B \longleftarrow^* B''$ and we are done.

In the second case, $r'' < r < r'$ (the right inequality is by the induction step), i.e., the rounds for B'' and B' are not consecutive. Hence, B' must contain a TC for round $r'-1$. By Lemma 2, this TC contains a qc_{high} with round $\geq r$.

Consider a successful call by an honest replica to vote for B'. The only way to satisfy the predicate to vote is to satisfy (2), which implies $B''.r \geq qc_{high}.r \geq B.r$, which is a contradiction to $r'' < r$. □

As a corollary of Lemma 3 and the fact that every globally direct-committed block is certified, we have

Theorem 2. *For every two globally direct-committed blocks B, B', either $B \longleftarrow^* B'$ or $B' \longleftarrow^* B$.*

Let's call a successful invocation of the **Commit** step by a replica a local direct-commit. For every locally committed block, there is a locally direct-committed block that extends it, and due to Lemma 1, also a globally direct-committed block that extends it. Each globally committed block defines a unique prefix to the genesis block, so Theorem 4 applies to all committed blocks. Hence all honest replicas commit the same block at each position in the blockchain.

Furthermore, since all honest replicas commit the transactions in one block following the same order, honest replicas do not commit different transactions at the same log position.

A.2 Liveness

Lemma 4. *When an honest replica in round $< r$ receives a proposal for round r from another honest replica, it enters round r.*

Proof. Recall that a well-formed proposal sent by an honest replica contains either a TC or QC of round $r-1$. When an honest replica receives such a proposal message, it will advance the round and enter round r. □

Lemma 5. *If the round timeouts and message delays between honest replicas are finite, then all honest replicas keep entering increasing rounds.*

Proof. Suppose all honest replicas are in round r or above, and let v be an honest replica in round r.

We first prove that some honest replica enters round $r+1$. If all $2f+1$ honest replicas time out in round r, then v will eventually receive $2f+1$ timeout messages, form a TC and enter round $r+1$. Otherwise, at least one honest replica, v' – not having sent a timeout message for round r – enters round $r+1$. For this, v' must have observed qc of round r and updated its qc_{high} accordingly.

Since qc_{high} is never decreased and included in timeout messages, if v' times out in any round $> r$, then its timeout message will trigger v to enter a round higher than r. Otherwise, v' must observe a QC in all rounds $> r$. In this case, an honest leader sends a proposal in some round $> r$. That proposal will eventually be delivered to v, triggering it to enter a higher round by Lemma 4. □

In an eventually synchronous setting, the system becomes synchronous after the the global stabilisation time (GST). We assume a known upper bound Δ on message transmission delays among honest replicas (practically, a back-off mechanism can be used to estimate Δ) and let 4Δ be the local timeout threshold for all honest replicas in all rounds.

Rounds are consecutive, advanced by quorum or timeout certificates, and honest replicas wait for proposals in each round. We first show that honest replicas that receive a proposal without the round timer expiring accept the proposal, allowing the quorum of honest replicas to drive the system progress.

Lemma 6. *Let r be a round such that no QC has yet been formed for it and in which no honest replica has timed out. When an honest replica v receives a proposal of an honest leader of round r, v will vote for the proposal.*

Proof. The predicate in the **Vote** step for a proposal with block B in round r checks that (1) round numbers are monotonically increasing, and (2a) either the block extends the QC of the previous round ($B.qc.r + 1 = r$), or (2b) the round of extended qc ($B.qc.r$) isn't less than the maximum high qc round in the TC of the previous round.

For (1), by assumption none of the $2f+1$ honest replicas have timed out, so no TC could have been formed for round r. Also by assumption, no QC has been formed. Hence, no honest replica may have entered or voted in a round larger than r. Round r has an honest leader, so when a honest replica executes **Vote** step the round r proposal, it does so for the first time and with the largest voting round.

For (2), we consider two cases. If $B.tc = \perp$, then by well-formedness of honest leader's proposal, the $B.qc$ it extends must have round number $r-1$, rounds are consecutive, and condition (a) holds.

If $B.tc$ is not empty, then it is a TC for round $r-1$, formed based on $2f+1$ timeout messages. In this case, $B.qc.r \geq \max\{qc_{high}.r \mid qc_{high} \in B.tc\}$ predicate determines whether the replica votes for the proposal. Since the leader is honest, $B.qc$ is the qc_{high} of the leader when the proposal was generated. The predicate holds as the honest leader updates the qc_{high} to have round at least as large as the qc_{high} of each timeout messages it receives (separately or within a forwarded TC). □

We now show a strong synchronization for rounds with honest leaders.

Lemma 7. *Let r be a round after GST with an honest leader. Within a time period of 2Δ from the first honest replica entering round r, all honest replicas receive the proposal from the honest leader.*

Proof. When the first honest replica enters round r, if it is not the leader, it must have formed a TC for round $r-1$. Let v be the honest leader of round r. Since honest replicas forward TC to the leader of the next round, v will receive the TC and advance to round r within Δ time of the first honest replica entering round r.

Upon entering round r, v multicasts a proposal, which is delivered within Δ time to all honest replicas. □

Liveness follows from the following

Theorem 3. *Let r be a round after GST. Every honest replica eventually locally commits some block B with $B.r > r$.*

Proof. Since the leaders are determined by round-robin and the number of byzantine replicas is bounded by f, we can find round $r' > r$ such that rounds $r', r'+1, r'+2$ all have honest leaders.

By Lemma 5 honest nodes enter increasing rounds indefinitely. Due to Lemma 7, all honest replicas receive round r' proposal with block B from the leader within 2Δ time of starting their round r' timer (by Lemma 4 triggering ones that haven't yet entered round r' to do so). By Lemma 6, honest replicas accept the proposal and vote for it. Within time Δ their votes are delivered to the leader of round $r'+1$, who forms a QC extending B and sends a proposal with a block $B_{r'+1}$. This proposal will be received by honest replicas within another Δ time, by Lemma 4 triggering them to enter round $r'+1$ before the local timer of 4Δ for round r' expires.

By Lemma 6, every honest replica accepts the proposal, prepares a vote and sends it to round $r'+2$ leader, who is also honest. At this point, since $f+1$ honest replicas voted for B, by Definition 1, B is globally direct-committed.

Continuing the argument, the honest leader of round $r'+2$ receives the votes to form the round $r'+1$ QC after at most 5Δ time of the first honest validator entering round r'. It then prepares and sends round $r'+2$ proposal that extends $QC_{B_{r'+1}}$. After at most another Δ time all honest replicas receive this proposal, leading them to enter round $r'+2$ (before local timer for round $r'+1$ expires[12]) and locally direct-commit B. □

Since we assume that each client transaction will be repeatedly proposed by honest replicas until it is committed (see Section 2), eventually each client transaction will be committed by all honest replicas.

B Correctness of `Ditto`

B.1 Safety

We follow the same notation from Appendix A with the following additional notations.

- We say a block B is sync-committed if it is directly committed due to a 2-chain $B \longleftarrow QC \longleftarrow B' \longleftarrow QC'$ in step **Commit**.

[12] Analogous to round r'. Moreover, as the round $r'+1$ leader enters the round first, no replica spends more than 3Δ time in round $r'+1$.

- We say a block B is async-committed if it is directly committed due to being the output of MVBA in step **Exit Fallback**.

We redefine the notion of global direct-commit as follows.

Definition 2 (Global direct-commit)**.** *We say that a block B is* globally direct-committed *if $f+1$ honest replicas each successfully perform the **Vote** step on block B' proposal in round $B.r+1$ and view $B.v$, such that $B'.qc$ certifies B. These **Vote** calls invoke **Lock**, setting $qc_{high} \leftarrow B'.qc$, and return $f+1$ matching votes (that could be used to form a $QC_{B'}$ with f other matching votes).*

From the definition, any block that is sync-committed is also globally direct-committed.

Lemma 8. *For two certified blocks B, B', if $B.rank = B'.rank$ then $B = B'$.*

Proof. Consider any view v, according to the protocol specification, the lowest ranked block in each view is the output of MVBA. By the Agreement property of MVBA, only one block can be certified in step **Exit Fallback** of view v. For later rounds of view v, due to quorum intersection, only one block can be certified for each round. □

Lemma 9. *If a block B is globally direct-committed, then any certified block B' s.t. $B'.v = B.v$ and $B'.r \geq B.r$ must extend B, i.e., $B \longleftarrow^* B'$.*

Proof. Follows from the proof of Lemma 3. □

Lemma 10. *If a block B is globally direct-committed, then any block B' decided by $MVBA_{B.v+1}$ must extend B, i.e., $B \longleftarrow^* B'$.*

Proof. Let $v = B.v$. By Definition 2, $f+1$ honest replicas execute the **lock** step and set qc_{high} such that $qc_{high}.rank \geq B.rank$. None of these honest replicas may have previously timed out in round $B.r+1$, and timing out stops voting in a round (but the replicas voted for B'). Since qc_{high} is never decreased, any of the above $f+1$ honest replicas must have $qc_{high}.rank \geq B.rank$ when sending the timeout message of view v. By quorum intersection, any $2f+1$ timeout messages of view v contain a message from one of these honest replicas. Therefore, any honest replica that sets $\texttt{fallback} = true$ (by receiving $2f+1$ timeout messages of view v) must update its qc_{high} such that $qc_{high}.rank \geq B.rank$.

Let B' be the block decided by MVBA_{v+1}, so $B'.v = v+1$. For the sake of contradiction, suppose B' does not extend B. Let B'' be the parent block of B', we have B'' also does not extend B. By the External Validity property of MVBA, B' must satisfy $f((B', \sigma'), qc_{high}) = 1$ for at least one honest replica h where h has qc_{high} when invoking MVBA_{v+1}. According to the definition of f, we have $B''.v = v$. There are two cases.

- $B''.rank \geq qc_{high}.rank$. Since h has $qc_{high}.rank \geq B.rank$ when invoking MVBA_{v+1}, we have $B''.rank \geq B.rank$. Since $B''.v = B.v = v$, by Lemma 9, B'' must extend B, contradiction.
- There exists a threshold signature σ' on $(\texttt{proof}, B'.v, B'.qc)$. According to the protocol specification (**Ack**), at least $2f+1$ distinct replicas have their qc_{high} such that $B''.rank = B'.qc.rank \geq qc_{high}.rank$ when handling the $(\texttt{proof}, B'.v, B'.qc)$ message. Since honest replicas will not decrease their qc_{high}, and any honest replica has $qc_{high}.rank \geq B.rank$ when entering fallback, we conclude $B''.rank \geq qc_{high}.rank \geq B.rank$. Since $B''.v = B.v = v$, by Lemma 9, B'' must extend B, again contradiction.

□

Lemma 11. *If a block B is async-committed, then any certified block B' s.t. $B'.v = B.v$ and $B'.r \geq B.r$ must extend B, i.e., $B \longleftarrow^* B'$.*

Proof. Consider the view $B.v$, according to the protocol specification, the lowest ranked block in view $B.v$ is the decision of $\text{MVBA}_{B.v}$, i.e., block B. At least $2f+1$ honest replicas set their qc_{high} such that $qc_{high}.rank \geq B.rank$ when $\text{MVBA}_{B.v}$ decides.

Suppose for the sake of contradiction, there exists a certified block B' s.t. $B'.v = B.v$ and $B'.r \geq B.r$, but B' does not extend B. Let B'' denote the ancestor block of B' such that $B''.v = B.v$ and has the lowest rank. Let B''' denote the parent block of B'', then $B'''.v < B.v$. Since B is the lowest ranked block in view $B.v$, and honest replicas vote with increasing round numbers in the same view, by quorum intersection at least one honest replica vote for B and then vote for B''. When h votes for B, it updates its qc_{high} such that $qc_{high}.rank \geq B.rank$. Then, h will not vote for B'' since $B'''.rank < B.rank \leq qc_{high}.rank$, contradiction.

□

Lemma 12. *If a block B is async-committed, then any block B' decided by $MVBA_{B.v+1}$ must extend B, i.e., $B \longleftarrow^* B'$.*

Proof. At least $2f+1$ honest replicas set their qc_{high} such that $qc_{high}.rank \geq B.rank$ when $\text{MVBA}_{B.v}$ decides. By a similar proof of Lemma 10, any block B' decided by $\text{MVBA}_{B.v+1}$ must extend B. □

Lemma 13. *For every certified block B' s.t. $B'.rank \geq B.rank$ such that B is async-committed, $B \longleftarrow^* B'$.*

Proof. We prove by induction on the view numbers. For the base case of view $B.v$, by Lemma 11, any certified block of view $B.v$ with rank $\geq B.rank$ must extend B.

For the induction step, suppose the lemma holds up to view $v \geq B.v$, and we will prove it holds also for view $v+1$. By the induction assumption, the async-committed block B_v of view v extends B, since B'' is certified. By Lemma 12, the

async-committed block B_{v+1} of view $v+1$ extends B_v, which implies that B_{v+1} extends B. Then by Lemma 11, any certified block of view $v+1$ extends B_{v+1}, which also extends B.

By induction, the lemma holds. □

Lemma 14. *For every certified block B' s.t. $B'.rank \geq B.rank$ such that B is globally direct-committed, $B \longleftarrow^* B'$.*

Proof. By Lemma 9, any certified block of view $B.v$ and rank $\geq B.rank$ extends B. By Lemma 10, the async-committed block B'' of view $B.v+1$ extends B. Then, by Lemma 13, any certified block B' with $B'.v \geq B.v+1$ and $B'.rank \geq B.rank$ extends B'', which implies that B' extends B. □

Theorem 4. *For every two committed blocks B, B', either $B \longleftarrow^* B'$ or $B' \longleftarrow^* B$.*

Proof. Let B_1, B_2 be any two blocks each directly committed due to either a sync-commit or an async-commit. By Lemma 13 and 14, the fact that every sync-committed block is globally direct-committed, and the fact that every sync-committed or async-committed block is certified, we have B_1, B_2 extending one another. For any committed block, there exists a block extending it and gets either sync-committed or async-committed. Therefore, any two committed blocks also extend one another. □

Hence all honest replicas commit the same block at each position in the blockchain. Furthermore, since all honest replicas commit the transactions in one block following the same order, honest replicas do not commit different transactions at the same log position.

B.2 Liveness

Lemma 15. *If the network is synchronous and all replicas are honest, then all honest replicas keep committing new blocks with increasing round numbers.*

Proof. For the initial round (say round 0), the leader proposes a genesis block B_0. Since the network is synchronous and all replicas are honest, all replicas will receive and vote for the block, and the votes will be received by the next leader. Similarly, the next round-1 leader proposes the round-1 block B_1 extending B_0, gets voted by all honest replicas, and then the round-2 leader proposes the round-2 block B_2 extending B_1. By simple induction, all leaders keep proposing blocks and all blocks will be certified. Then, according to the commit rule, all replicas keep committing new blocks with increasing round numbers. □

Lemma 16. *If an honest replica h invokes $\texttt{MVBA}_v(B,\sigma)$, then for any honest replica h', its predicate $f((B,\sigma), qc_{high}) = 1$ where qc_{high} is the high-QC of h' when invoking $\texttt{MVBA}_v$.*

Proof. Let B' be the highest committed block with view $\leq v-1$. If B' is sync-committed due to a 2-chain $B' \longleftarrow B''.qc \longleftarrow B''$, then from the proof of Lemma 10, any honest replica has its $qc_{high}.rank \geq B'.rank$ when entering the fallback (setting $\texttt{fallback} = true$). Also, B'' is the highest certified block in view $B'.v$, otherwise B'' should be the highest committed block. Note that $B''.r = B'.r+1$. If B' is async-committed, then by the Agreement property of MVBA and step **Exit Fallback**, all honest replicas have their $qc_{high}.rank \geq B'.rank$ when entering the fallback (setting $\texttt{fallback} = true$). Similarly, either B' is the highest certified block in view $B'.v$, or there exists a higher certified block B'' in view $B'.v$ such that $B' \longleftarrow B''.qc \longleftarrow B''$. Note that $B''.r = B'.r+1$. Therefore, all honest replicas have their $qc_{high}.rank \geq B'.rank$ when entering the fallback (setting $\texttt{fallback} = true$), and any QC of view $B'.v$ has round $\leq B'.r+1$.

If the honest replica h invokes $\texttt{MVBA}_v(B,\sigma)$ with $\sigma \neq \perp$, it is externally valid for any honest replica according to the definition of predicate.

If the honest replica h invokes $\texttt{MVBA}_v(B,\sigma)$ with $\sigma = \perp$, according to the protocol specification (**MVBA**), we have $B.qc.rank > qc_{high}.rank \geq B'.rank$ where qc_{high} is the high-QC of h when h enters the fallback (sets $\texttt{fallback} = true$). As proved, any QC of view $B'.v$ has round $\leq B'.r+1$. Thus, $B.qc.rank \geq qc_{high}.rank$ where qc_{high} is the high-QC of any honest replica when invoking $\texttt{MVBA}_v$. According to the definition of predicate, $f((B,\perp), qc_{high}) = 1$ since $B.qc.rank \geq qc_{high}.rank$ for any honest replica with qc_{high} when invoking $\texttt{MVBA}_v$.

□

Lemma 17. *If the network is asynchronous, all honest replicas keep committing new blocks with increasing ranks.*

Proof. If at least one honest replica keeps committing new blocks in the **Commit** step, then the committed blocks have increasing round numbers and all honest replicas eventually receive and commit these blocks. Otherwise, let v be the highest view that any block is committed, and suppose block B is async-committed in view v ($\texttt{MVBA}_v$ decides this block). By the Agreement and Termination properties of MVBA, all honest replicas eventually decide B and enter view v. Since no honest replica commits new blocks, eventually all honest replicas will timeout view v, and enter fallback. Since the steps **Enter Fallback, Ack, MVBA** are non-blocking, all honest replicas eventually invoke $\texttt{MVBA}_{v+1}$. By Lemma 16, any honest replica's input is externally valid to all replicas. By the Termination property of MVBA, all honest replicas will eventually decide one block from $\texttt{MVBA}_{v+1}$, which has a higher rank than previously committed blocks. Hence, all honest replicas keep committing new blocks with increasing ranks. □

Theorem 5. *Each client transaction is eventually committed by all honest replicas.*

Proof. By Lemma 15 and 17, all honest replicas keep committing new blocks with increasing ranks. Since we assume that each client transaction will be repeatedly proposed by honest replicas until it is committed (see Section 2), eventually each client transaction will be committed by all honest replicas. □